Towards an industrial use of FLUCTUAT on safety-critical avionics software[⋆]

David Delmas[1], Eric Goubault[2], Sylvie Putot[2], Jean Souyris[1], Karim Tekkal[2], and Franck Védrine[2]

[1] Airbus France S.A.S., 316, route de Bayonne, 31060 TOULOUSE Cedex 9, France, `Firstname.Lastname@airbus.fr`

[2] 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. Most modern safety-critical control programs, such as those embedded in fly-by-wire control systems, perform a lot of floating-point computations. The well-known pitfalls of IEEE 754 arithmetic make stability and accuracy analyses a requirement for this type of software. This need is traditionally addressed through a combination of testing and sophisticated intellectual analyses, but such a process is both costly and error-prone. FLUCTUAT is a static analyzer developed by CEA-LIST for studying the propagation of rounding errors in C programs. After a long time research collaboration with CEA-LIST on this tool, Airbus is now willing to use FLUCTUAT industrially, in order to automate part of the accuracy analyses of some control programs. In this paper, we present the IEEE 754 standard, the FLUCTUAT tool, the types of codes to be analyzed and the analysis methodology, together with code examples and analysis results.

1 Introduction

For a decade, Airbus has been implementing formal techniques developed by academia into its own verification processes, for some avionics software products. So far, the most successful technique has been abstract interpretation based static analysis [5, 6]. It is currently used industrially on several avionics software products developed at Airbus to compute safe upper-bounds of stack consumption with AbsInt StackAnalyzer, and worst-case execution time with AbsInt aiT WCET [24, 23].

More static analyzers could be transferred soon. For instance, ASTRÉE [7] is a credible candidate for an industrial use in the near future [8, 22], in order to prove the absence of run-time errors on control programs. Indeed, such programs perform a lot of floating-point computations, so that the absence of floating-point overflow or other invalid operation has to be guaranteed. But

⋆ This work is supported by FP7 European project INTERESTED, ITEA 2 European project ES_PASS, a grant from the DIGITEO foundation, and the french DPAC (Direction des Programmes Aéronautiques Civils).

proving freedom from run-time errors is not enough: the issue of the precision of computations has to be addressed also. This is typically the kind of properties for which FLUCTUAT is designed. For this reason, Airbus is willing to use FLUCTUAT within its industrial process.

1.1 Numerical computations in control programs

All control programs are based on control theory, which describes the physical data that are manipulated, together with control algorithms, in the realm of (ideal) real numbers. Even if the control algorithms are correct by design in real number computations, we have to prove that the imprecision due to finite-precision implementation has negligible effects on the system, for instance, introduces negligible errors compared to the imprecision of the computer I/O, on which bounds are generally available. This view is complementary, but not equivalent to the view taken in particular in robust control theory. Robust control theory deals with control algorithms which are in some sense "robust" to perturbations of input signals and to uncertainties in parameters of the controlled system but not to "computation" perturbation, i.e. the fact that finite-precision machines do have subtle discrete semantics which perturb the control algorithm along the full history of computation.

Take for instance, a flight control computer. It reads inputs from pilot controls (side sticks and pedals) and other sensors and aircraft systems or functions (such as the autopilot), and computes commands for actuators of control surfaces. This has been achieved using fixed-point arithmetic until the 1990s, but the later fly-by-wire generations have switched to floating-point representation. The main difference between both formats is the nature of errors. Fixed-point numbers yield absolute errors, whereas floating-point yield relative errors. Besides, the IEEE 754 standard provides engineers with a precise specification of floating-point data formats and basic operations, which makes it easier to assess accuracy systematically. Moreover, more and more microprocessors implementing a native Floating-Point Unit can be embedded.

1.2 Accuracy and sensitivity analyses

Control software is usually developed in a model-based approach. Most of the source code is generated automatically from high-level synchronous data-flow specifications. The computations to be performed are described by system designers at model-level in a graphical stream language such as SCADE [9] or Simulink [16], by means of external basic blocks. These basic operators implement the elementary calculations. They are usually available in some external library including logical, temporal and numerical operators. This toolbox can be either provided together with the modelling tool, or implemented by the user in a lower-level programming language, in order to meet his specific needs exactly. The latter case occurs typically for safety-critical avionics software that have to be certified according to the DO-178B/ED-12B aeronautical international standard. In this context, assessing the accuracy and stability of all numerical library operators is a key point in the overall numerical precision analysis. We need to analyze very precisely, for each operator:

1. the potential loss of accuracy;
2. the potential propagation of errors from inputs to outputs (i.e. sensitivity).
3. the behavior of the underlying algorithm:
   a. bounding the number of iterations of an iterative algorithm when it may depend on the accuracy;
   b. proving that the algorithm actually computes outputs close to what is expected, both in real and floating-point numbers (functional proof).

This need has been addressed so far through a combination of testing and intellectual analyses. In this paper, we aim at describing a way to automate this analysis using the FLUCTUAT tool. Therefore, we first present the IEEE 754 standard and the FLUCTUAT tool, then we state the analysis method and demonstrate it on examples similar to some of Airbus's library operators[3]. We will show mostly the use of FLUCTUAT for points 1 and 2 above, but also for 3.a and 3.b in Section 4.3.

2 The IEEE-754 standard

The IEEE-754 standard [17] defines the format of floating-point numbers and the basic arithmetic operations on every processor supporting it. The standard defines the float format (8 bits for the exponent, 23 bits for the mantissa and 1 bit for the sign) and the double format (11 bits for the exponent, 52 bits for the mantissa and 1 bit for the sign). It also makes a distinction between several kinds of floating point numbers, which we describe for double precision numbers:

– the normalized numbers have a non-null, non-maximal exponent and, 53 relevant bits in the mantissa (the upper bit is implicitly 1) : $f = 2^{exp-2^{11-1}+1} \times (1.0 + m/2^{52})$, where exp is the exponent and m the mantissa,
– the denormalized numbers have a null exponent, a non-null mantissa and $1 + \lfloor \log_2(m) \rfloor$ relevant bits in the mantissa : $f = 2^{-2^{11-1}+2} \times (m/2^{52})$,
– $+0$ (resp. -0) has a null exponent, a null mantissa and a positive (resp. negative) sign,
– plus and minus infinities have a maximal exponent, a null mantissa,
– NaN (Not a Number) has a maximal exponent, a non-null mantissa. The upper bit of the mantissa differentiates “Signaling NaN” from “Quiet NaN”.

The standard also specifies four possible rounding modes: round to nearest (and in case of a tie, round to nearest even mantissa) which is the default mode, round to minus infinity, round to plus infinity, round to zero. The atomic arithmetic operations $+$, $-$, $\times$ $/$, and $\sqrt{}$ are exactly rounded, that is their result is the real result rounded to the nearest floating-point number, according to the chosen rounding mode. Some common pitfalls due to the use of finite precision

[3] For obvious confidentiality reasons, no real embedded code can be shown.

may induce high relative errors or problematic behavior : among them, we can cite representation errors (for example on seemingly innocent constants such as 0.1), absorption (when adding two numbers of very different amplitudes), cancellation (when subtracting two very close numbers), unstable tests (when the real and float control flows are different, with a discontinuity between the two flows), or a drift in computation that will eventually cause large errors. We give here a few examples of these :

Representation error and computation drift

```
float time = 0.0; int ct = 0; while (++ct < 20000) time += 0.1;
```

The user could expect that the successive errors should cancel out and so that `time` at the end of the loop is close to the real value 2000 . It is not the case, and `(time - 2000.0)/time` $= 2.7 \times 10^{-4}$, since the initial error on the representation of 0.1 is always the same, and then, for all `time` between any $[2^n, 2^{n+1}[$, all computations are rounded in the same direction.

Invariant and safety The following example comes from [20].

```
double modulo(double x, double mini, double maxi)
  { double delta = maxi-mini; double decl = x-mini;
    double q = decl/delta; return x - ((int) q)*delta; }
```

does not return a number $\in [mini, maxi]$, since when $decl/delta$ is rounded up to a power of 2, like 1.0, the result is $<$ `mini`. Such a case occurs one time every 2^{54} – e.g. `modulo(179.99999999999998, -180, 180) < -180` –, but it may be a source of crash for a system.

Absorption and cancellation Consider the function $f(x, y) = 333.75y^6 + x^2(11x^2y^2 - y^6 - 121y^4 - 2) + 5.5y^8 + x/(2y)$ proposed by Rump: computed in float $f_f(77617, 33096) = 1.172603...$, in double $f_d(77617, 33096) = 1.1726039400531...$ We would thus think that the computation is correct, however in real numbers $f(77617, 33096) = -0.82739$. This is due to a catastrophic cancellation during the computation.

Moreover, compilers can transform the code in order to optimize the final binary code, in such a way that source-level analysis might be unsound. We will not discuss this issue in detail here, but workarounds consist in, mainly, analyzing compilation patterns and parameterize the static analyzer or partially rewrite the C code so that it agrees with the binary (evaluation order etc.); using a certified compiler [18]; or analyze the binary directly (see for instance [19] for a description of a version of FLUCTUAT which directly analyzes relocatable assembly code) or conjointly with the source code, through a "compilation of invariants", see for instance [21]. However, in Airbus' process, no unsound optimisation is performed by the compiler, and the compiled code can be safely traced back to the source code, as imposed by the DO-178B standard.

3 The FLUCTUAT tool

3.1 General description

FLUCTUAT [15] is a static analyzer by abstract interpretation of ANSI C programs, that focuses on numerical properties. For that, it computes ranges of values that can be taken, for all possible executions, at all control points of the program, with two different semantics, the idealized one in real numbers, and the implemented one in finite precision numbers (here IEEE 754 floating-point numbers and machine integers). It bounds the difference between the values taken by variables with these two semantics, and decomposes it on its provenance on the control points of the program, allowing the user to determine which parts of the program mainly contribute to the rounding error.

A graphical interface (see Figure 4) allows in particular to visualize, at the end of the program and for each variable, the errors committed in the program as a graph, on which the user can quickly identify the main sources of errors.

3.2 Specific abstract domains based on affine arithmetic

FLUCTUAT relies on weakly-relational abstract domains that use affine forms [12, 14, 11] for the computation of values and errors on variables.

Abstract domain for idealized value in real numbers Affine arithmetic is a more accurate extension of interval arithmetic, introduced in 93 [3], that takes into account linear correlation between variables. The real value of a variable x is represented by an affine form $\hat{x}$:

$$\hat{x} = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n,$$

where $x_i \in \mathbb{R}$ and the *noise symbols* ε_i are independent symbolic variables with unknown value in $[-1, 1]$. The coefficients $x_i \in \mathbb{R}$ are the *partial deviations* to the center $x_0 \in \mathbb{R}$ of the affine form. Indeed, these deviations express uncertainties on the values of variables, for example when inputs are given in a range of values. The sharing of the same noise symbols between variables expresses *implicit dependency.*

The joint concretization of these affine forms is a center-symmetric polytope, that is a zonotope. These zonotope-based abstract domains provide an excellent trade-off between computational cost and accuracy. We refer the reader to [12, 14] for a full description of the abstract domain based on these forms.

In practice, these affine forms are themselves computed with floating-point coefficients, but the computation is made sound by over-approximating the rounding error committed on these coefficients and agglomerating this error in new noise terms.

Abstract domain for floating-point value and difference with real value The abstract domain implemented in FLUCTUAT which we used for the experimentations presented here, extends these affine forms for the computation of the floating-point value of variables and for the difference between the real and floating-point computation, in the following model :

$$f^x = (\alpha_0^x + \bigoplus_i \alpha_i^x\, \varepsilon_i) + (e_0^x + \bigoplus_l e_l^x\, \eta_l + \bigoplus_i m_i^x\, \varepsilon_i),$$

where

- f^x is the floating-point abstract value for variable x,
- $\alpha_0^x + \bigoplus_i \alpha_i^x \varepsilon_i$ is the affine form that models the real value of x, i being the control points of the program,
- e_0^x is the center of the error,
- η_l are noise symbols that express the dependency between the errors, so that $e_l^x \eta_l$ expresses the rounding error committed at point l of the program, and its propagation through further computations,
- $m_i^x \varepsilon_i$ expresses the propagation of the uncertainty on value at point i, on the error term; it allows to model dependency between errors and values.

A new noise symbol is created for each new rounding error introduced by the computations. Thus, at the end of the program, for each visible variable, we can associate each error term of its affine error form to the location in the program that introduced it. This information is drawn on an error graph.

3.3 Use and main features of FLUCTUAT

We introduce in this section some features of FLUCTUAT that will be exemplified in the following sections.

Directives FLUCTUAT comes along with a language of directives to be added in the source code by the user. The first kind of directives allows to specify values of variables, possibly with initial errors. For example `double x = DBETWEEN(0,1);` will specify that x is a double precision variable which can take a value between 0 and 1, whereas `double x = DOUBLE_WITH_ERROR(0,1,-0.01,0.01);` will specify that is has in addition an error in [-0.01,0.01], for example due to the use of an imperfect sensor to measure that value. Also, one can, in a loop, bound the derivative between successive inputs, which is useful in some cases to get plausible behavior. Note also that we have recently made a further step in taking into account a model of a continuous environment in our analysis, using a guaranteed integrator to bound the behavior of the continuous environment. This is out of the scope of this paper, but more details can be found in [2]. Also note that in some cases, we use the `BUILTIN` directives to construct sub-domains on which to analyze a given function (as in Section 4.3) which allows for improving precision of the analysis by a “manual” disjunctive analysis. In that case, we collect back the values that variables can take on the full domain using directive `res=DCOLLECT(subres,res);` (`res` is the result of the analyzed function that we construct by collecting results on subdomains: `subres`).

The second kind of directives allows the user to print more information than just the values and graphs of errors at the end of the program. Specifying `DPRINT(x)` in the source code will make the analyzer print the value and error of double precision variable x each time it meets the directive, allowing the user to follow the evolution of x in the program. If specifying `DSENSITIVITY(x)` at some point of the program, the sensibility of variable x to inputs of the program will be displayed : indeed, the abstract domains of FLUCTUAT are particularly well suited to such sensitivity analysis, by constructing linearized forms on the inputs. Also, using heuristics based on these same linearized forms, the analyzer can generate scenarii that allow to reach a value close to the maximal (or minimal) bound on some variable, and then run it.

Parameters of the analysis Many parameters allow to tune the analysis, they can be set via the graphical interface. Among them, some allow to tune the trade-off between accuracy and computation in the fixpoint computation for loops, for instance the initial unfolding of loops, the number of cyclic unfoldings (useful in some examples to compute the fixpoint of a more contractant function), and the number of iterations before extrapolating the result (useful to reach a fixpoint in finite time in all cases).

When in presence of highly non-linear computations, it can be needed to subdivide some inputs for a more accurate computation, which can be done automatically.

Also, symbolic execution can be used, in some cases where the analysis gives large bounds, to confirm the behavior around some particular input values. The symbolic execution mode is based on the same abstract semantics as for the static analysis mode, but follows the abstract trace starting with input variables having as values the midpoint of their input ranges, and as errors, their full error range as specified through the corresponding directives. Symbolic execution can thus also be combined with subdivision (regular, for the time being) to have sample values in the input range.

4 Automating the accuracy analysis of basic operators with FLUCTUAT

4.1 Families of basic operators

Synchronous control programs, among which fly-by-wire, are built with several types of operators:

Pure boolean/integer operators All inputs and outputs have boolean or integer types, and outputs at time `t` only depend on inputs at time `t`. Algorithms use no remanent data, and perform no floating-point computations. Typical such operators are logic gates and boolean switches.

Pure temporal operators All inputs and outputs have boolean or integer types. Outputs at time `t` depend on inputs at ticks `0`, `1`, ..., `t-1`, `t` of the synchronous clock. Algorithms perform no floating-point computation, but use remanent data. Such operators include delays, timers, flip-flops, triggers, input confirmation operators, etc.

Pure numerical operators Most inputs and outputs are real-valued. Outputs at time `t` only depend on inputs at time `t`. Algorithms perform floating-point computations, but use no remanent data. Main types are sum, product, comparison, conversion and interpolation operators. Well-known examples are divisions, square roots, trigonometric and transcendental functions.

Both numerical and temporal operators Typical such recursive operators are digital filters, and signal integrators, derivators and speed limiters.

This paper focuses on numerical operators, be they temporal or not. Previous work has addressed functional verification of operators using theorem-proving techniques [1], but floats could not be handled through weakest precondition calculus.

In the examples we will give, note that the algorithms used for embedded operators are somewhat specific. For instance, the source codes for pure numerical operators contain hardly any loop at all, and no unbounded loop if any, as worst-case execution time and timing determinism are key constraints for safety-critical control programs. The same constraint applies for temporal numerical operators, although there is always an implicit main loop implementing the reactive nature of the control program, which will be emulated in the examples on recursive operators such as digital filters. These temporal numerical operators can be used on a very long period of time (i.e. on a large number of iterations) and we analyze them for any potential number of iterations, see Section 4.4.

4.2 The analysis process

Building the analysis project The user selects a set of C source files through the GUI. The source code should be compilable/linkable, but for directives useful to the analysis (see Section 3.3), that provide the tool with:

- hypotheses on the environment of the program:
  - ranges of input variables (real or floating-point) values;
  - ranges of errors on input variables;
  - bounds on inputs variables speed.
- union or intersection strategies to:
  - fine-tune the analysis process;
  - build irregular “custom” subdivisions for input ranges.
- requirements for printing relevant information.

Parametrizing the analysis Directives help tune the analysis locally, whereas analysis options allow for global parametrisation. The main choices for the purpose of analyzing basic operators are:

- abstract semantics: relational or non-relational static analysis versus symbolic execution;
- initial and cyclic unfolding of loops (mainly for recursive operators);
- refinement of bitwise operations when such operations are used;
- regular subdivisions of input variable ranges (in case we want very precise results on non-linear computations).

Exploitation of results The tool warns about possible:

- run-time errors, and their source in the analyzed program: the user has to make sure they cannot occur in his application, due to thresholds or closed-loop control. This can be for instance achieved performing a global analysis with ASTRÉE. When possible, the program should be fixed for the tool not to issue such warnings.
- unstable tests (when the floating-point and the real control flows may be different): the user must make sure they have no impact on the values computed by his program. In particular, he must check these tests cannot create discontinuities in the set of output values.

Static analyses may also yield infinite or abnormally large ranges or errors on output variables (the analyzer points to the main sources of errors). In this case, the user should switch to the symbolic execution mode of the tool, in order to make sure the analyzed program is not trivially unstable. More generally, the precision of the static analysis should be assessed through worst-case generation[4] and symbolic execution with (possibly many) subdivisions. The static analysis parameters should be tuned until results can be compared to that of symbolic execution.

4.3 Analyzing interpolation operators

An arctangent approximation Using rational functions to approximate the arctangent function is a good trade-off between efficiency and precision. For instance, one may choose the Padé approximant of order $(2, 2)$:

$$\frac{\arctan(x)}{x} \sim \frac{15+4x^2}{15+9x^2} = 1 - \frac{x^2}{3+\frac{9}{5}x^2}$$

Using this approximation on the interval $[0, 1]$ provides a method for approximating $\arctan(x)$ for all x in $\mathbb{R}$, via the identities:

$$\arctan(x) = -\arctan(-x) = \frac{\pi}{2} - \arctan(\frac{1}{x})$$

Let us derive a straightforward implementation for the arctangent operator:

```
const double Pi=3.141592653589793238;
double PADE_2_2(double x) {       double ARCTAN_POS(double x) {
 double x_2=x*x;                   if (x>1) return Pi/2-ARCTAN_0_1(1/x);
 return 1-x_2/(3+9./5*x_2);        else     return ARCTAN_0_1(x); }
}
                                  double ARCTAN(double x) {
double ARCTAN_0_1(double x) {      if (x<0) return -ARCTAN_POS(-x);
    return x*PADE_2_2(x);          else     return ARCTAN_POS(x); }
}
```

Now we may run an accuracy analysis of the `ARCTAN` operator for the complete range of double precision floating-point numbers:

[4] FLUCTUAT is able to deliver its best guess for input values of a program, so that to maximize or minimize some output variable, see [13].

```
double x = DBETWEEN(-DBL_MAX, DBL_MAX);
double y = ARCTAN(x);
```

The analysis takes 10 ms and 16 MB, and ensures $y \in [-1.57, 1.57]$. This interval is of course very satisfactory, as the expected output range for an arctangent approximation is $]-\frac{\pi}{2}, \frac{\pi}{2}[$. The computed over-approximation of the error interval for `y` is $[-4.83 \times 10^{-16}, 4.83 \times 10^{-16}]$. This error interval is an over-approximation of the difference between the algorithm result in the real field and the algorithm result in floating-point arithmetic. It does not take into account the difference between the implementation and the real arithmetic function *arctan*. However, this model error can often be tackled with the tool (see paragraph "a glimpse on functional proofs" at the end of this section for the sine implementation). A side-remark is the fact that the π constant cannot be represented exactly as it is declared in the code with `Pi`: the representation error is found by FLUCTUAT to be in $[1.214 \times 10^{-16}, 1.249 \times 10^{-16}]$.

Now we want to perform a sensitivity analysis on function `ARCTAN`. Because this analysis does not cope with unstable tests nor subdivisions, for the time being, we need to run separate analyses on the $]-\infty, -1[$, $[-1, 0]$, $[0, 1]$ and $]1, \infty[$ subintervals. Besides, sensitivity has little meaning for very large inputs, we thus restrict ourselves to $[-10^4, -1[\cup[-1, 0] \cup [0, 1]\cup]1, 10^4]$.

The analysis for the restricted range [0,1] takes 10 ms and 16 MB, and ensures that $|\frac{\Delta y}{\Delta x}| \leq 0.788$. For $]1, 10^4]$, we replace `DBETWEEN(0,1)` with `DBETWEEN(DSUCC(1), 1.e4 )` where `DSUCC(x)` (resp. `DPREC(x)`) stands for the next (resp. previous) floating-point number after (resp. before) `x`. The analysis costs are unchanged. FLUCTUAT ensures that $|\frac{\Delta y}{\Delta x}| \leq 3.12 \times 10^{-08}$. The results for $]-10^4, -1[$ (resp. $[-1, 0]$) are the same (resp. as for $[0, 1]$).

As a conclusion, FLUCTUAT helps us prove that the implementation in floating-point numbers of this approximation of the arctangent function:

1. introduces only negligible errors;
2. cannot amplify errors on inputs.

An interpolated sine Embedded programs classically use interpolation tables to approximate trigonometric functions, such as the sine of an angle expressed in degrees. For instance, one may build an array of 361 doubles providing approximations of the sine function for every half degree:

$$\forall k = 0, 1, \ldots 360, \left| T[k] - \sin\left(\frac{k}{2}\right) \right| < 10^{-4},$$

and implement an approximation of sine by interpolating between the points of this table for angles between 0 and 180 degrees and using the identities

$$\forall x \in \mathbb{R},\ \forall k \in \mathbb{Z},\ \sin(x) = -\sin(-x) = \sin(x+360k) = \sin\left(x - 360\left\lfloor\frac{x+180}{360}\right\rfloor\right)$$

Let us, as for the arctangent, try to run an analysis for the full range of double numbers, for the following naïve implementation of the sine operator:

```
extern const double T[361];
double SIN_0_180(double x) {
 double dx, i_dx, v_inf;
 double v_sup; int i;
 dx=2*x; i=dx; i_dx=i;
 v_inf=T[i]; v_sup=T[i+1];
 return v_inf + (dx - i_dx)
        * (v_sup - v_inf); }

double SIN_180(double x) {
 if (x<0)
  return -SIN_0_180(-x);
 else
  return SIN_0_180(x); }

double SIN_POS(double x) {
 if (x>180) return SIN_180(x
     -360.*(int)((x+180.)/360.));
 else       return SIN_180(x); }

double SIN(double x) {
 if (x<0) return -SIN_POS(-x);
 else     return SIN_POS(x); }
```

The analysis takes only 1.52 seconds and 24 MB. FLUCTUAT issues infinite values and errors for the result of the sine function, and warns that the `(int)((x+180.)/360.)` cast may be undefined. Indeed this implementation uses a conversion from double to int, so it is only valid for inputs x such that $\left\lfloor \frac{|x|+180}{360} \right\rfloor$ can be represented in the type `int`. In our 32 bits case, this is the $]-773094113100, 773094113100[$ interval. This is not an issue, provided that call contexts for `SIN` in the whole control program are guaranteed to fall into this last interval, for instance through a global static analysis with ASTRÉE.

The analysis costs for the reduced domain are unchanged, but its result is still disappointing: the computed over-approximation of the value (and error) interval for the result of the sine algorithm in floating-point numbers is $[-3.09\times 10^{12}, 3.09\times 10^{12}]$, whereas it is $[-1.5, 1.5]$ in the real-number semantics. Besides, FLUCTUAT warns the user that the program test in function `SIN_180` is unstable. The warning also stresses that "real is bottom" in the `(x<0)` branch, which means that whenever the floating-point execution takes this branch, the real execution executes the `(x>=0)` branch. This requires careful attention.

At this point, we realize we may be faced with the pitfall described in Section 2 with the `modulo` example. As a matter of fact, whenever `x` is very close to (but smaller than) a number of the form $180 + 360 \times k$, for some $k \in \mathbb{N}$, then a rounding error occurs when evaluating expression `(x+180.)/360.` in floating-point arithmetic. The result is rounded to the `double` representing $k + 1$. In all such cases, the `x - 360.*(int)((x+180.)/360.)` expression has value close to (though below) 180 in the reals, but close to (though below) -180 in the floats.

In such a case, expression $T[i+1]$ in function `SIN_0_180` attempts to access array `T` out of bounds, as signalled by FLUCTUAT. Thus, we need to fix our naïve implementation of the sine operator. We add a 362^{nd} array element, such that $T[0] = T[360] = T[361] = \sin(0) = \sin(\pm 180) = 0$.

Now we have fixed the implementation, we subdivide the input ranges enough for FLUCTUAT to perform a very accurate analysis on every subinterval defined by the interpolation table. For instance, on the $[-180, 180]$ range, using 720 subdivisions, a four hour / 20 MB analysis ensures $y \in [-1.0004, 1.0004]$ with errors in $[-1.75 \times 10^{-2}, 1.75 \times 10^{-2}]$.

Such a large relative error is unacceptable, especially considering the interpolation table has been chosen to ensure a 10^{-4} accuracy. In order to decide whether this is due to an issue in the implementation or to a lack of precision of the static analysis, we switch to the symbolic execution mode of FLUCTUAT with same range and subdivisions, thus choosing a sample of inputs in the input range. As a result, we get a useful (though unsound for the whole range of values) estimate of the error range for y: $[-1.05 \times 10^{-16}, 1.05 \times 10^{-16}]$. This result is a good hint that the accuracy of the implementation is not at stake. We thus look for a more precise way to perform the static analysis: we write a new main function implementing a custom irregular subdivision, with a singleton for every interpolation point of the interpolation table:

$$[-180, 180.5[= \bigcup_{i=-360}^{360} \left\{\frac{i}{2}\right\} \cup \left]\frac{i}{2}, \frac{i+1}{2}\right[$$

```
double x=-180., y=0., xi, yi; int i;
for (i=-360; i<=360; i++) {
 xi = i*0.5;  yi = SIN(xi);
 x = DCOLLECT(x, xi);  y = DCOLLECT(y, yi);
 xi = DBETWEEN(DSUCC(i*0.5), DPREC((i+1)*0.5)) ;
 yi = SIN(xi); x = DCOLLECT(x, xi);  y = DCOLLECT(y, yi); }
```

The static analysis is run with no (tool-generated) subdivision, but unrolling the main loop 720 times. It takes 97.39 seconds and 36 MB, and ensures $x \in [-1.8 \times 10^2, 1.805 \times 10^2]$ and $y \in [-1.0, 1.0]$ with errors in $[-4.97 \times 10^{-16}, 4.97 \times 10^{-16}]$.

Over 99% of the error range originates from instruction `dx=2*x`. Next comes expression `v_inf + (dx - i_dx) * (v_sup - v_inf)`. The rest are negligible representation errors within the interpolation table. Such a result is satisfactory: the imprecision generated by the floating-point implementation is negligible compared to the accuracy of the interpolation table.

Now we may also run sensitivity analyses for all $[\frac{i}{2}, \frac{i+1}{2}[$ subintervals of the $[-180, 180[$ range. Each (separate) analysis takes 20 MB and runs (at most) for 0.71 second. Merging the 720 results, we can guarantee $|\frac{\Delta y}{\Delta x}| \leq 0.0176$. This is of course a very satisfactory result, as we are approximating a real-valued sine function expressed in degrees, i.e. such that

$$\left|\frac{d}{dx}\left(\sin\left(\frac{180}{\pi}x\right)\right)\right| = \left|\frac{180}{\pi}\cos\left(\frac{180}{\pi}x\right)\right| \leq \frac{180}{\pi} \sim 0.0175$$

A glimpse on approximate functional proofs As previously indicated, we can use FLUCTUAT to check that the approximate operator satisfies, in real and in floating-point numbers, some properties close to classical properties of the real sin function. For example here:

$\sin(x) = -\sin(-x) = sin(180 - x) \quad \sin(x)^2 + \sin(x - 90)^2 = 1$

We basically replace the body of the loop calling the `SIN` function on subintervals of length $\frac{1}{2}$ by the following:

```
xi = DBETWEEN(DSUCC(i*0.5+90), DPREC((i+1)*0.5+90)) ;
s1 = SIN(xi); c1 = SIN(xi-90);
z1 = s1 + SIN(-xi); z4 = s1-SIN(180-xi);
z6 = s1*s1+c1*c1-1.0;
```

FLUCTUAT proves the following in 111 seconds and 40.96 Mb (for x in [90,180]): $z1 \in [-4.337 \times 10^{-18}, 4.337 \times 10^{-18}]$ in reals and $z1 \in [-5.44 \times 10^{-16}, 5.44 \times 10^{-16}]$ in floating-point numbers, $z4 \in [-4.34 \times 10^{-18}, 4.34 \times 10^{-18}]$ in reals and $z4 \in [-4.34 \times 10^{-16}, 4.34 \times 10^{-16}]$ in floating-point numbers, $z6 \in [-1.32 \times 10^{-4}, 1.21 \times 10^{-4}]$ with negligible error (within $[-8.44 \times 10^{-16}, 7.93 \times 10^{-16}]$). Moreover FLUCTUAT finds out that the worst-case for the value of `z6` is at iterate 74, and more precisely, it delivers as best guess for reaching the maximum of $z6$: $x = 127.5$. Asking FLUCTUAT to check this value, it gives $z6 = 1.21 \times 10^{-4}$ confirming the supremum bound found by static analysis. A simple analysis shows that this is due to an error less than 10^{-4} on the corresponding table entries. This shows that our algorithm for the sine function is actually most probably computing something quite close to the sine function, with the 10^{-4} absolute precision expected, mostly due to the implementation even in infinite precision (the interpolation table), and not to its finite precision implementation.

4.4 Analyzing recursive operators

Analyzing a set of order 2 filters conjointly Consider the code of Figure 1. It implements a linear filter of order 2, run `N` times with at each iteration a new unknown input `E` within [0,1], independent of the previous inputs.

```
double E, E0, E1, S0, S1, S;
int i;
E=DBETWEEN(0,1.0);
E0=DBETWEEN(0,1.0);
for (i=1;i<=N;i++) {
  E1 = E0;    E0 = E;
  E = DBETWEEN(0,1.0);
  S1 = S0;    S0 = S;
  S = 0.7*E-E0*1.3+E1*1.1
           +S0*1.4-S1*0.7;
  DPRINT(S);  }
DSENSITIVITY(S);
```

Fig. 1. A linear order 2 filter.

```
double E, E0, E1, S0, S1, S;
double A1, A2, A3, B1, B2; int i;
A1 = DBETWEEN(0.5, 0.8);
A2 = DBETWEEN(-1.5,-1);
A3 = DBETWEEN(0.8,1.3);
B1 = DBETWEEN(1.39,1.41);
B2 = DBETWEEN(-0.71,-0.69);
E=DBETWEEN(0,1.0); E0=DBETWEEN(0,1.0);
for (i=1;i<=N;i++) {
 E1 = E0;    E0 = E;
 E = DBETWEEN(0,1.0);
 S1 = S0;    S0 = S;
 S = A1*E+E0*A2+E1*A3+S0*B1+S1*B2;
 DPRINT(S); }
 DSENSITIVITY(S);
```

Fig. 2. A set of order 2 filters.

We find the results summarized in Table 1. The first columns of the table indicate respectively the number of cyclic unfolding and the number of iteration at which we begin to widen (instead of using the join operator). These are essential parameters to tune the accuracy of the analysis. We then give the time of analysis on a laptop PC (1Gb memory, Pentium Duo 1.66GHz), the maximal amount of memory used, and the ranges the analysis gives for the floating-point value of the output `S` of the filter, and for its imprecision error:

c. unfold	widen. thresh.	time	mem.	`S` (float)	`S` (error)
10	50	41 s	53 Mb	$[-6.30; 7.96]$	$[-5.13 \times 10^{-14}; 5.15 \times 10^{-14}]$
30	50	167 s	53 Mb	$[-5.18; 6.84]$	$[-3.23 \times 10^{-14}; 3.25 \times 10^{-14}]$
60	50	418 s	53 Mb	$[-5.12; 6.78]$	$[-3.11 \times 10^{-14}; 3.14 \times 10^{-14}]$

Table 1. Results on the order 2 filter of Fig. 1

Completely unfolding the loop on 100 iterations confirms that the bounds found by the analysis are fairly precise (see Figure 3): `S` is found to be in [-1.09, 2.76] with error in $[-1.15 \times 10^{-14}, 1.17 \times 10^{-14}]$. Still, there is room for improvement. We are currently experimenting a new "global" union which would find [-3.61,5.28] instead of [-6.30,7.96] in the case of a cyclic unfolding of 10, and [-3.00,4.67] instead of [-5.12,6.78] in the case of a cyclic unfolding of 60. Note that these results are just a bit less precise (for the floating-point range) than what the specialized abstract domain for filters of [10] delivers, while also giving precise bounds for the implementation error as well as their origins.

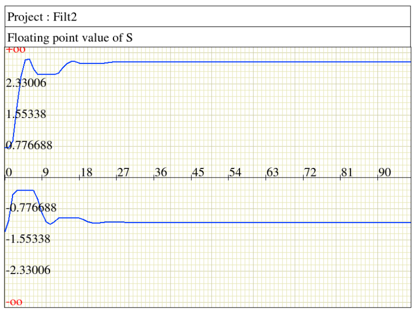
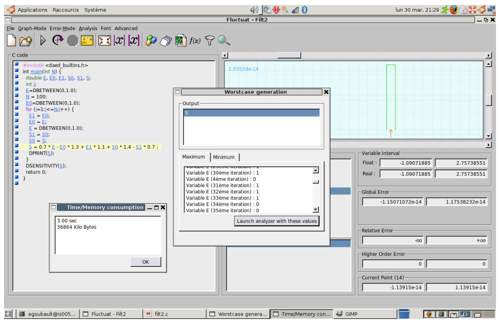


Fig. 3. Floating-point values along the iterations



```
#include <daed_builtins.h>
int main(int N) {
  double E, E0, E1, S0, S1, S;
  int i;
  E=DBETWEEN(0,1.0);
  N = 100;
  E0=DBETWEEN(0,1.0);
  for (i=1;i<=N;i++) {
    E1 = E0;
    E0 = E;
    E = DBETWEEN(0,1.0);
    S1 = S0;
    S0 = S;
    S = 0.7 * E - E0 * 1.3 + E1 * 1.1 + S0 * 1.4 - S1 * 0.7 ;
    DPRINT(S);
  }
  DSENSITIVITY(S);
  return 0;
}
```

```
Variable E (30ème iteration) : 1
Variable E (4ème iteration) : 0
Variable E (31ème iteration) : 1
Variable E (32ème iteration) : 1
Variable E (33ème iteration) : 1
Variable E (34ème iteration) : 0
Variable E (35ème iteration) : 0
```

```
3.00 sec
36864 Kilo Bytes
```

Variable Interval		
Float :	-1.09071885	2.75738551
Real :	-1.09071885	2.75738551
Global Error	-1.15071072e-14	1.17538232e-14
Relative Error	-oo	+oo
Higher Order Error	0	0
Current Point (14)	-1.13915e-14	1.13915e-14

Fig. 4. Fluctuat screen at the end of the analysis

Consider the code of Figure 2 now. It is the same order 2 filter, but with uncertain coefficients. These coefficients might look as narrow intervals, but a manual calculation reveals that the poles of the Z-transform of this filter have a module in [0.932,0.975], thus close to instability (the filter would be unstable if the norm could go above one).

FLUCTUAT finds the results shown in Table 2. On the first line, we see the results for the full static analysis for `N` unknown, the second and third line show the results of the loop completely unfolded on the first 200 iterations (hence the analysis is particularized to N=200), first without subdividing coefficients B1 and B2, and then, subdividing them. We can see that the results of the static analysis (i.e. [post]fixpoint calculation) are not too much over-approximated compared to the results of the unfolded loop, which, as could be seen on the evolution graph, are quite stable on the last iterations, so should not be far from the actual invariant[5]. As in Table 1, we indicate the number of cyclic unfolding, the widening threshold, time and memory usage, and floating-point/error range for the output S. First column indicates how much we subdivide coefficients B1 and B2 : FLUCTUAT can also deliver automatically estimates for coefficients A1, A2,

#B1&B2	c.	widen.	time	mem.	S (float)	S (error)
1, 1	200	20	1242 s	65 Mb	$[-15.1; 16.6]$	$[-7.9 \times 10^{-14}; 7.9 \times 10^{-14}]$
1, 1	200	no	21 s	57 Mb	$[-5.63; 7.13]$	$[-2.93 \times 10^{-14}; 2.93 \times 10^{-14}]$
10, 10	200	no	2943 s	57Mb	$[-4.81; 6.33]$	$[-2.61 \times 10^{-14}; 2.61 \times 10^{-14}]$

Table 2. Results on the order 2 filter of Fig. 2

A3, B1 and B2 in their input ranges such that the corresponding output reaches a value close to the bounds previously delivered by the static analysis (using here 10 subdivisions on B1 and 10 subdivisions on B2). For the upper bound, it finds B1=1.41, B2=-0.69, A1=0.8, A2=-1 and A3=1.3, along with input values for successive iterates of E, when we unfold the loop for 200 iterations. When the filter is executed with these values, S is found to be equal to 5.34.

For the lower bound, it finds B1=1.408, B2=-0.71, A1=0.5, A2=-1.5 and A3=0.8. In that case, S is found to be equal to -0.75. This gives an idea of the quality of the invariants found.

5 Conclusions and future work

In this paper, we have shown how FLUCTUAT can be used to automate part of the accuracy analysis of basic numerical operators of control programs. We have shown extremely precise results could be obtained in a quite systematic way, which should ease the work of engineers in charge of this task.

The next step, for such a research prototype, is its industrial use within operational development teams. This requires some extra work. First, an "industrial" version of the tool is needed: this phase has already been prepared through a DIGITEO[6] grant and plans are currently made for creating a service oriented spinoff company, together with partial transfer to publishing companies, such as ABSINT and ESTEREL (through the European project INTERESTED). Also,

[5] This could also be analyzed by a modified version of the ASTREE analyzer, implementing a specific extension of the abstract domain of [10], with comparable, but slighly less precise results.

[6] DIGITEO is a French research foundation devoted to complex software-intensive systems, regrouping such institutions as Ecole Polytechnique, CEA, INRIA, Supelec, Paris XI, ENS Cachan, Centrale etc. see http://www.digiteo.fr.

depending on the verification strategy of operational end-users, FLUCTUAT may need to be qualified as a verification tool, according to the DO-178B/ED-12B aeronautical international standard. If such is the case, Airbus will conduct qualification activities matching the targeted context. Detailed information on the principles and architecture of the tool will be needed from the tool provider, in order to anticipate DO-178C extended qualification objectives for such formal tools. There again, work has started thanks to the support from DIGITEO.

Further work on the accuracy analysis of basic operators could be to interoperate FLUCTUAT with FRAMA_C (http://frama-c.cea.fr) or ASTRÉE. The latter could be used to compute over-approximated ranges for input variables of all operators on the complete program, and the former could then restrict its accuracy analyzes to these sound input intervals. In the cases where the local value analysis with FLUCTUAT is more precise than the global analysis with ASTRÉE, invariants computed by the former may be re-injected to improve the precision of the analysis of the latter.

Beyond local accuracy analyses of basic operators, FLUCTUAT can also be used to perform very accurate analyses on C code generated from sets of SCADE sheets. First steps have been made in that direction with the successful study of a complex program relying on set of linear recursive filters of orders varying from 3 to 8. Outside the scope of the cooperation with Airbus, CEA-LIST has also had successful experiences with other industrialists such as Hispano-Suiza and IRSN, see [15], and a 33KLoC C program implementing a critical function for Astrium's Automated Transfer Vehicle, see [4]. This should be further investigated, in order to assess the accuracy of some critical independent system-level aircraft functions.

References

1. P. Baudin, D. Delmas, S. Duprat, and B. Monate. Proving temporal properties at code level for basic operators of control/command programs. In *Proceedings of ERTS 2008, SIA*, 2008.
2. O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Vedrine. Hybridfluctuat: a static analyzer of numerical programs within a continuous environment. In *Computed Aided Verification conference, CAV'09, Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 620–626, 2009.
3. J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *Anais do VI Simpósio Brasileiro de Computaccão Gráfica e Processamento de Imagens (SIBGRAPI'93)*, pages 9–18, October 1993.
4. 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.
5. P. Cousot. Abstract interpretation based formal methods and future challenges. In *Informatics*, pages 138–156, 2001.
6. P. Cousot and R. Cousot. Basic concepts of abstract interpretation. In *IFIP Congress Topical Sessions*, pages 359–366, 2004.
7. P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. The astrée analyzer. In *ESOP*, pages 21–30, 2005.

8. D. Delmas and J. Souyris. Astrée: From research to industry. In *Static Analysis, 14th International Symposium, SAS 2007, Kongens Lyngby, Denmark, August 22-24, 2007, Proceedings*, volume 4634 of *Lecture Notes in Computer Science*, pages 437–451. Springer, 2007.
9. F.-X. Dormoy. Scade 6 a model based solution for safety critical software development. In *Embedded Real-Time Systems Conference*, 2008.
10. J. Feret. Static analysis of digital filters. In *European Symposium on Programming (ESOP'04)*, number 2986 in LNCS. Springer-Verlag, 2004.
11. K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain taylor1+. In *Computed Aided Verification conference, CAV'09, Grenoble, France*, volume 5643 of *Lecture Notes in Computer Science*, pages 627–633, 2009.
12. E. Goubault and S. Putot. Static analysis of numerical algorithms. In *SAS'06, Seoul*, volume 4134 of *LNCS*, pages 18–34, 2006.
13. E. Goubault and S. Putot. Under-approximations of computations in real numbers based on generalized affine arithmetic. In *SAS*, volume 4634 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2007.
14. E. Goubault and S. Putot. Perturbed affine arithmetic for invariant computation in numerical program analysis. *CoRR*, abs/0807.2961, 2008.
15. E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: Principles and experiments. In *FMICS*, volume 4916 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2007.
16. Hunt, Lipsman, Rosenberg, Coombes, Osborn, and Stuck. *A Guide to MATLAB, 2e: for Beginners and Experienced Users.* Cambridge University Press, 2006.
17. *IEEE 754 standard for floating-point arithmetic.* Floating-Point Working Group of the Microprocessor Standards Subcommittee of the Standards Committee of the IEEE Computer Society. Work in Progress, 2004.
18. X. Leroy. Formal certification of a compiler back-end, or: programming a compiler with a proof assistant. In *33rd ACM symposium on Principles of Programming Languages*, pages 42–54. ACM Press, 2006.
19. M. Martel. Validation of assembler programs for dsps: a static analyzer. In *PASTE '04: Proceedings of the 5th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 8–13. ACM, 2004.
20. D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.
21. X. Rival. Symbolic transfer functions-based approaches to certified compilation. In X. Leroy, editor, *31st Symposium on Principles of Programming Languages*, pages 1–13. ACM, janvier 2004.
22. J. Souyris and D. Delmas. Experimental assessment of astrée on safety-critical avionics software. In *SAFECOMP*, pages 479–490, 2007.
23. J. Souyris, E. Le Pavec, G. Himbert, G. Borios, V. Jégu, and R. Heckmann. Computing the worst case execution time of an avionics program by abstract interpretation. In *5th Intl. Workshop on Worst-Case Execution Time (WCET) Analysis*, Dagstuhl, Germany, 2007.
24. S. Thesing, J. Souyris, R. Heckmann, F. Randimbivololona, M. Langenbach, R. Wilhelm, and C. Ferdinand. An abstract interpretation-based timing validation of hard real-time avionics software. In *DSN*, 2003.