Fine-Tuning the Accuracy of Numerical Computations in Avionics Automatic Code Generators

Alexis Werey, David Delmas, Matthieu Martel

Fine-Tuning the Accuracy of Numerical Computations in Avionics Automatic Code Generators

Alexis Werey[†,*], David Delmas[*] and Matthieu Martel[†]

[*] Airbus Operations SAS, Toulouse, France
[†] University of Perpignan, Laboratoire de Mathématiques et Physique LAMPS, France

Abstract

Most of safety-critical embedded software, such as fly-by-wire control programs, performs a lot of floating-point computations. High level specifications are expressed in a formal model edited manually in SCADE through a graphical interface. It generally handles numerical variables and constants as if they were ideal real numbers. This work, for the purpose of numerical accuracy analysis, presents a new version of an Automatic Code Generator (ACG). This tool transforms high-level models into C codes and performs static computations by using multiple-precision arithmetic. This article describes a successful way of controlling computation accuracy of numerical constants in an Automatic Code Generator. An accuracy analysis on numerical constant values is presented in a case study.

Keywords: automatic code generator, industrial software, floating-point computation, numerical accuracy, constant propagation

1 Introduction

In this article, we focus on a method for tuning the accuracy of numerical computations in an avionic Automatic Code Generator (ACG), i.e. a software tool which aims at transforming a formal model into a target source code (generally in C). The generated codes are dedicated to be embedded into avionic systems. Nowadays, the accuracy of computations in avionic systems depends on floating-point arithmetic. Our motivation is to study the enhancement of floating-point computations in these codes by applying program transformation [11, 12]. For this purpose and as a first step, numerical error analysis is needed on static computation of constants during the code generation. Some constants are computed from the internal formal model by the ACG for code efficiency reasons due to CPU consumption. The analysis is performed by : a) observing algorithms accuracy manually or using static analysis by abstract interpretation [21, 20] ; b) checking systematically the accuracy of the resulting floating-point values after generation.

Assurance The embedded software are categorized at a Design Assurance Level depending of the safety impact on the system, in the sense of the avionic standard DO-178C/ED-12C [1]. In fact, the ACG has to be qualified as a development tool at the same assurance level as the generated software.

Numerical optimization In order to lower the number of computations and accept almost only arithmetic operations in the embedded code, the ACG provides simplified constants. A similar situation in compilers arises when static expressions are computed at compile-time (e.g `const y=1;x=y+1` replaced by `x=2` by the compiler.) The fact that almost any compiler makes the same processings than our ACG that makes the results presented in this article rather general.

Floating-point arithmetic errors The floating-point representation necessary leads to numerical errors created by roundings and propa-

gated by operations. In practice, they are highly negligible compared with the uncertainty of the input data (precision of the sensors/actuators). However as part of a long-term perspective, the evolution of the hardware precision and the sophistication of control algorithms could necessitate in the future a better accuracy concerning the computation of numerical operations.
We aim at verifying that a combination of constants of the high level model is not likely to degrade the quality of calculations. Indeed, constants are often called several times inside a loop. The objective in the present work is to make use of the state of the art to generate constants as accurate as possible and then to be able to measuring floating-point errors. A tunable approach using the rational representation [8] for arithmetic operations $(+, -, \times, \div)$ and the extended-precision [9] floating-point representation for elementary functions (such as $\sqrt{}$, tan) has been selected for the ACG reimplementation.

This article is organized as follows : in Section 2, we give an overview of the high critical real time software code generation. Section 3 introduces the floating-point arithmetic and its use for constant propagation. Then Section 4 describes the details of the reimplementation. Section 5 presents experimental results on a use case. Finally, Section 6 gives general perspectives and Section 7 concludes.

2 Embedded Code Generation

In this section, we describe briefly the development of embedded software products. Figure 2 illustrates the development process.

2.1 Model Specification

In the context of Airbus, the high-level formal specifications of avionics program behaviours are expressed by SCADE sheets [2] and translated into the Lustre [3] synchronous data-flow programming language.
The implementation of this specification is processed automatically through an Automatic Code Generator developed internally.
The model representation consists in data (real, integer or boolean as either constants or variables) and data-flow structures (symbols and their inputs/outputs). Examples of symbols include delays and cosines, as depicted in Figure 1.

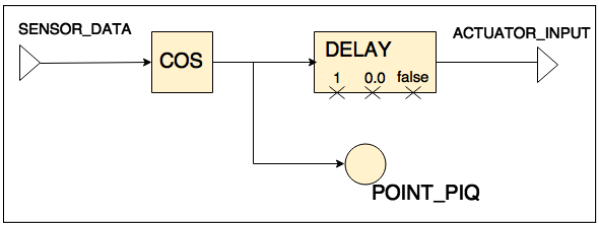


Figure 1 – An example of a SCADE sheet

Symbols may possibly hold constant parameters, e.g. the *cos* symbol in the Figure 1 does not contain constants while the *delay* symbol has 3 constants. Some other constants are global and not directly referenced in the sheet such as the clock.

2.2 Generated Code Structure

A symbol library is implemented by hand in C or assembly using macro-functions. The generated code, which has the same semantics as the initial formal specification, can therefore call these predefined macros. For hardware efficiency and safety reasons, macros are written as simple as possible with IEEE754 [4] well-defined operations with the double precision format. Macros and specification nodes can in addition differ on the number and the type of constant parameters since the ACG reduces the CPU-time by evaluating static expressions during the code generation.
A static expression in a symbol can be computed once for all by the ACG from either input and global constants in the model or by simplification of an algorithm thanks to mathematical properties. This method is similar, from a certain point of view, to the constant propagation optimization performed by compilers [5].

2.3 Numerical Error Analysis

In the process of qualification, the ACG is subject to analysis and in particular to floating-point accuracy. Note that during the step of optimization that we describe in the next section, the ACG computes values in floating-point double precision. Two complementary sorts of analysis are carried out :

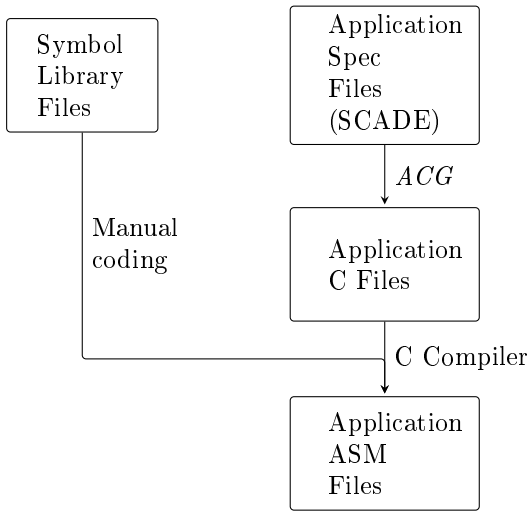


Figure 2 – The C development chain of the software

- A handwritten analysis followed by a static analysis [21] on floating-point accuracy to over-approximately estimate the errors generated by the algorithm.
- A set of tests to verify whether the ACG is near or far from the expected results.

Our objective is to provide a method to automatically analyse the current ACG with inputs from the formal model.
Note that some tools are already used for the analysis of control-command programs. *Astre* computes safe bounds on floating-point variables, *Fluctuat* computes error bounds between the exact and the floating-point semantic, both by abstract interpretation. *Compcert* is a certified compiler which use Coq libraries to handle the floating-point arithmetic [17].

3 Constant propagation of floating point expressions

3.1 The IEEE754 Standard

The IEEE-754 standard [4] describes the floating-point number format as well as the elementary operations $(+, -, \times, \div)$ and the square root function. It defines:

1. the simple precision on 32 bits with 8 bits for the exponent e, 23 bits for the mantissa m and 1 bit for the sign s
2. the double precision on 64 bits with 11 bits for the exponent, 52 bits for the mantissa and 1 bit for the sign
3. the double extended precision on a number of bits higher than 79 with, more than 15 bits for the exponent, more than 63 bits for the mantissa and 1 bit for the sign

It also specifies several kinds of floating-point numbers, e.g. in radix 2:

- normalized numbers represented by: $f = s \cdot (1.m_0...m_{p-1}) \cdot 2^e$ where $s \in \{-1, 1\}$, p the length of m and $E_{min} \leq e \leq E_{max}$ ($E_{min} = -1022$ and $E_{max} = 1023$ in double precision)
- denormalized numbers defined when $e = 0$ and represented by: $f = s \cdot (0.m_0...m_{p-1}) \cdot 2^{E_{min}}$
- signed zeros $+0$ and -0
- infinities $\pm\infty$
- NaN (Not A Number)

The standard also defines 4 rounding modes : rounding $\circ_\sim$ to nearest, rounding $\circ_{-\infty}$ towards $-\infty$, rounding $\circ_{+\infty}$ towards $+\infty$ and rounding $\circ_0$ towards zero. Arithmetical operations are exactly rounded, which means that assuming $\uparrow_\circ: \mathbb{R} \to \mathbb{F}$ the function returning the floating-point value c a real number with the chosen rounding mode $\circ \in \{\circ\ , \circ_{-\infty}, \circ_{+\infty}, \circ_0\}$, and an arithmetical operation $\diamond \in \{+, -, \times, \div\}$, then for all floating-point numbers $f_1, f_2 \in \mathbb{F}$:

$$f_1 \diamond_{\mathbb{F},\circ} f_2 = \uparrow_\circ (f_1 \diamond_{\mathbb{R}} f_2) \quad (1)$$

Despite this propriety, floating-point operations can lead to subtle traps as in logical as in material considerations [6, 10]. Among several reasons of numerical errors, we can cite : the representation errors, for example the rational number $\frac{1}{10}$ is not representable in floating-point radix 2 ; the accumulation and propagation of errors through operations ; the absorption (resp. cancellation), a loss of precision when adding a number with a smaller one

from a different order (resp. when subtracting two numbers approximately equal) ; unstable tests, the path executed in the control flow is different from the one expected in the real semantics.

3.2 Constant propagation

In the long term, we aim at improving the floating-point accuracy of the generated embedded code by reducing error propagation. A first step consists in analyzing the floating-point accuracy of the computation of constants during the code generation. Equation (1) gives an example of a first order low-pass filter algorithm, considering that $X(t)$ and $Y(t)$ are temporal values denoting respectively the input and the output at time t. The constants a, b and c are related to physical properties.

$$Y(t) = \frac{\frac{2b}{a}}{\frac{2b}{a}+1}\, Y(t-1) + \frac{1}{\frac{2b}{a}+1}\,(X(t)+X(t-1)) \quad (2)$$

The terms $c_1 = \dfrac{1}{1+\frac{2*b}{a}}$ and $c_2 = 1 - c_1$ can be calculated beforehand. The generated code then looks like:

$$Y(t) = c_2\; Y(t-1) + c_1\;(X(t) + X(t-1)) \quad (3)$$

The generated constants c_1 and c_2 appear several times in a function executed at every tick of a synchronous clock, their numerical accuracy is then worth to be considered. In addition, the embedded software environment constraints, for example the real time constraints, e.g. *Worst-Case Execution Time* (WCET) [7], are relevant at execution-time rather than at compile-time. In order to improve their accuracy, the floating-point operations can be computed by an arbitrary precision library such as the *Multi-Precision Floating-Point* library (*MPFR*) [9], which has in addition the benefit to be independent from the host machine. Indeed, static computations achieved by ACG, or more generally by compilers may be sensitive to the arithmetic of the processor as well to the dynamic libraries of the host machine and libraries get rid of this issue.

Example of floating-point rounding errors A first example of floating-point error that may arise is a call to the floor function (returning an integer) after some floating-point operations. If the floating-point arithmetic double precision is used to implement the real value $\lfloor\, x \times 1000 \rfloor$ when $x \in \mathbb{R}$ and $\lfloor . \rfloor : \mathbb{R} \to \mathbb{Z}$, the natural way is to write $floor(x \times 1000)$ when this time x is a floating-point variable, $\times$ is the standard floating-point multiplication and $floor$ the floating-point floor function returning an integer. Some values are error-prone, for $x = 1.001$, this implementation leads to an important relative error : 10^{-3}.

$$\begin{aligned} floor(1.001 * 1000) &= floor(1.000999999 * 1000) \\ &= floor(1000.999999) \\ &= 1000 \end{aligned} \quad (4)$$

3.3 Static Analysis

In this section, we illustrate the loss of accuracy resulting from the computation of constant parameters by a static analysis on the computation of the second order low-pass filter constants.

Low-pass filter transfer function

$$H(s) = \frac{1}{1 + 2\xi\frac{s}{\omega_0} + \frac{s^2}{\omega_0^2}}$$

This function specified in the formal model is implemented in a way that requires the following constant a :

$$a = \frac{\frac{2}{\tan^2(\omega_0 \frac{\Delta_t}{2})} - 2}{1 + \frac{2\xi}{\tan(\omega_0 \frac{\Delta_t}{2})} + \frac{1}{\tan^2(\omega_0 \frac{\Delta_t}{2})}} \quad (5)$$

Assuming that the clock Δ_t is 0.01 seconds, the analysis of a with *Fluctuat* and 1000 global subdivisions on ω_0 returns the results depicted in Figure 3.

The range of a is determined by the analysis from the input range of Δ_t, ω_0 and ξ and the relative error is the comparison of the real result and the floating-point one. Considering the errors, the generated floating-point constant a is sound and complying with the specifications.
Assuming that Δ_t is 0.02 seconds, Figure 4 show the relative error on the computation of a. Lower and upper bound of the relative error are drawn in the graph.

Constant	Range	Relative error
Δ_t	$[1.00000000 \cdot 10^{-2}; 1.00000001 \cdot 10^{-2}]$	$[-2.16840435 \cdot 10^{-17}; -2.03287906 \cdot 10^{-17}]$
ω_0	$[6.28318530 \cdot 10^{-1}; 9.42477797 \cdot 10^{1}]$	[0;0]
ξ	$[9.99999998 \cdot 10^{-2}; 1.20000001]$	[0;0]
a	$[5.95590847 \cdot 10^{-1}; 2.11425866]$	$[-6.52831414 \cdot 10^{-15}; 6.61998150 \cdot 10^{-15}]$

Figure 3 – Results of the static analysis from Equation 4

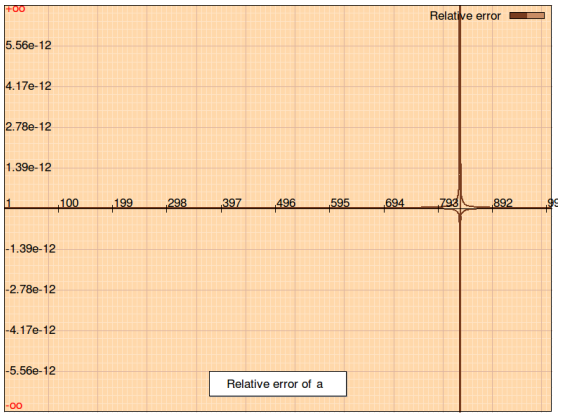


Figure 4 – Relative error bounds on a

Even though the absolute error stays stable, the relative error increases when $\omega_0 \simeq \frac{\pi}{2\Delta_t}$. A new implementation of the ACG would be an additive analysis tool to reinforce the assurance that constants are generated accurately.

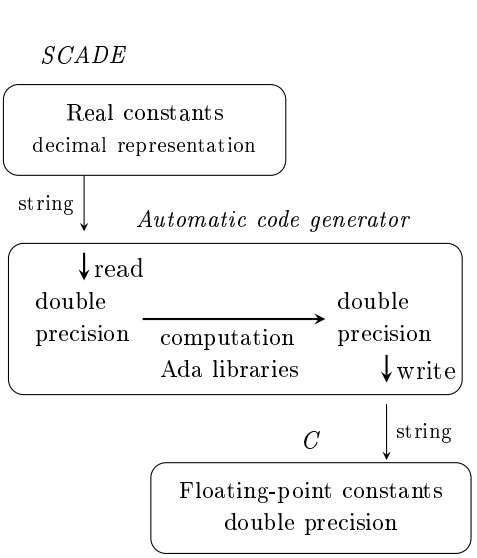
4 A Tunable Code Generator

4.1 Alternative Arithmetics

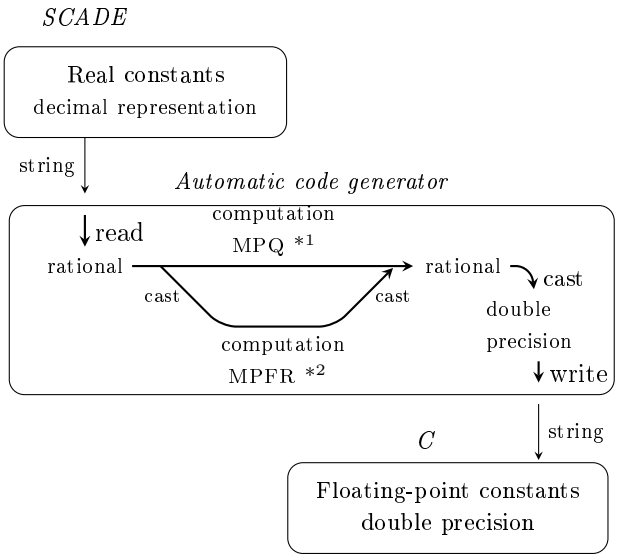
Several works has been done to estimate and to improve the numerical accuracy of programs [16]. We can cite interval arithmetic which consists of bounding the exact result by two floating-point numbers. Stochastic arithmetic [15] that consists of running the programs several times with random rounding modes. Doing so, the round-off errors are randomly propagated and the output is eventually statistically approximated.
The ones that would be considered for the reimplementation of the computations in the ACG is the rational and the extended-precision arithmetic. The rational representation is defined by two unbounded integers, a numerator and a denominator. The computations in this representation is exact and thus not subjected to any round-off errors, but valid only for the elementary operations $\{+, -, \times, \div\}$. Since the constants of the model are expressed in decimal and most of the operations in the ACG are elementary, this arithmetic is well-suited to this representation. The downside of this approach is that it is not stable in term of memory and time costs. Basically, after a certain number of operations on a variable x, the cost of a new operation is getting higher and higher due to the possible increasing length of the numerator and the denominator. However in our context, it does not have to be taken into account since most of the time few operations are involved in static computation of constants and not performed at run time.
In some circumstances, for example on a call to a tangent or a square root function, the floating-point representation with extended precision is more appropriate and can be configured with the selected number of bits.

4.2 Architectures

We reimplement the existing ACG written in Ada to improve the numerical accuracy of the constant computations. The current ACG uses the double precision complying with the precision of the IEEE754 double format. As shown in Figure 5a, the library Ada is used to compute the static expressions and both the reading and the writing process of constants generate rounding errors. In addition, there is some call to the tangent function which is not defined in the IEEE754 Standard but which is defined in the dynamic mathematical library of the host machine. Figure 5b describes the computation process of the new ACG using the multi-precision rational arithmetic library (MPQ) from GMP [8] and the *MPFR* library.



(a) Static computation in the current ACG



(b) Static computation in the optimized ACG

*1 : only with arithmetic operators
*2 : with elementary functions

5 Experimental Results

In order to have representative results of static computations, the former and the new ACG have been tested on a use case representative of the model for a large avionic system (several thousand nodes). They have been executed on a Sun Solaris platform on a SPARC architecture. MPFR has been configured to the to-nearest rounding mode and 200 bits of allocation for a floating-point number. We compare by analysis of the generated constants the numerical errors created by the ACGs. The analysis calculates for each constant the relative error between the two values. If c_1 is generated by the first ACG and c_2 by the second from the same static expression, the relative error corresponds to $\frac{|c_1-c_2|}{c_2}$ as we suppose that the second ACG computes almost exactly. Figure 6 depicts the results. N indicates the number of generated constants whose relative error from the analysis is greater than the error fixed in x-axis. For clarity, Figure 7 shows the index of the most significant bit in the mantissa of the absolute error $|c_1 - c_2|$ between the related constants c_1 and c_2 for the 1000 worst cases. Some conclusions have been raised from this experiment : The results are complying with the accuracy requirements for the former ACG. There are approximately 1000 constants whose relative error is above 10^{-10} and 500 above 10^{-9} on a total of 40000 generated constants. They all are in conformance with the internal error criterion expressed by an acceptable error bound.

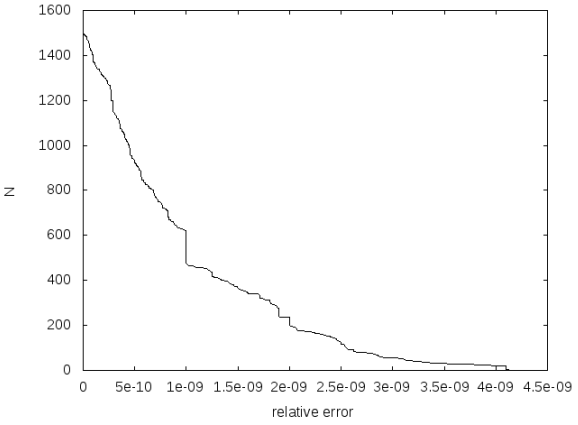


Figure 6 – Relative error between generated constants

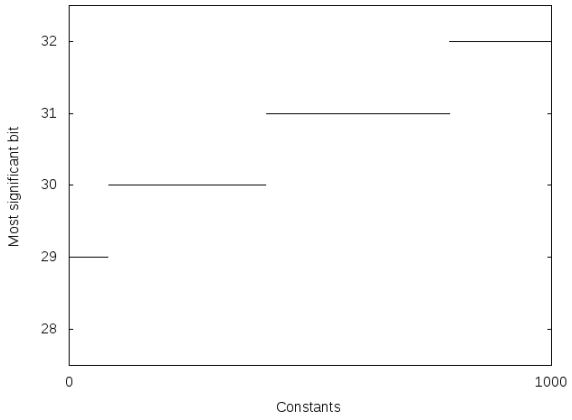


Figure 7 – Index of the most significant bit of the absolute error

6 Perspectives

The work introduced in this article, concerning the accurate evaluation of the static arithmetic expressions by ACGs, is a first step towards the automatic generation of fully optimized code with respect to the accuracy of floating-point computations. The next steps are the generation of accurate arithmetic expressions and, more generally, the generation of accurate programs. More precisely, in the floating-point arithmetic, the accuracy of expressions depends on how formulas are written and mathematically equivalent expressions give different results, more or less accurate, when evaluated by the machine. For example, $a+(b+c)$ is generally different from $(a+b)+c$ and $x+x^2$ is generally different from $x(x+1)$. The choice of the best formula depends on the ranges of the inputs which can be obtained by static analysis [20].

Automatic techniques enabling one to find a very accurate implementation of a formula for the IEEE754 arithmetic have been introduced in [12] and other work has been done to re-organize larger pieces of code containing several statements made of assignments, conditionals and loops [13]. For example, `x=a+b ; y=c+d; z=x+y` may be rewritten as `z=((a+c)+d))+b` if this choice is relevant, depending on the ranges of `a`, `b`, `c` and `d` computed by static analysis. These transformations still needs to be extended to the inter-procedural case. At mid-term, ACGs could take advantage of these techniques to generate accurate code, made of formulas mathematically equivalent to the ones of the high level model (e.g. *Scade*) but taylored to evaluate very accurately in the computer arithmetic.

The code transformations operated to improve the numerical accuracy may lead to programs which are different enough from the original ones while they still computing the same mathematical results. Another important research direction is to take care of certification. Traceability and a good level of confidence in the transformed codes are obviously mandatory. We aim at generating correctness certificates ensuring that the transformed codes are mathematically equivalent and more accurate than the original codes. These certificates will be expressed using formal proof assistants. In practice, we plan to use the Coq theorem prover. A related problem, more theoretical is to show that substituting a more accurate piece of code to an original piece of code inside a large program improves the accuracy of the whole application. Ongoing research in currently done in this direction.

7 Conclusion

In this article, we have shown how the static computations performed by ACGs are sensitive to the host machine on which the code generation is performed. Indeed, the resulting code may depend on the arithmetic of the host machine as well as on the dynamic libraries of its operating system. This problem is not limited to ACGs. It is general to all the modern compilers which perform constant propagation during their optimization passes. Our experimental results show that large industrial applications may be impacted by this transformation. In addition, even if the accuracy of the computations done at compile-time without using any specific high precision library like $MPFR$ is acceptable, reproducibility and maintenance questions still remain since compiling again the same program on another machine, possibly several years later to deliver a new version of the software, may yield a program embedding constants which are different from these of the original code.

The accuracy tunable ACG is intended to be used as an analysis tool and may be subject to a process of qualification with the corresponding libraries GMP MPQ and $MPFR$ in the case of a development application.

Another aspect of the problem of improving the accuracy concerns the execution-time. We wish the transformed codes, optimized for accuracy, be at least as efficient as the original codes. The transformations introduced in [12, 13, 14] neither slow the applications nor speed them up significantly. However, the transformation of the arithmetic expression is only guided by accuracy. The existing methods could be interestingly extended to search a compromise between time and accuracy, or, in other word, a rewriting of the computations which improves both accuracy and execution-time even if it not optimal for each criterion taken separately.

References

[1] DO-178C/ED-12C, Software Considerations in Airborne Systems and Equipment Certification, RTCA/EUROCAE, December 2011.

[2] F.-X. Dormoy. Scade 6 a model based solution for safety critical software development. In Embedded Real-Time Systems Conference, 2008.

[3] N. Halbwachs, P. Caspi and D. Pilaud. The synchronous dataflow programming language Lustre. Another Look at Real Time Programming, Proceedings of the IEEE, Special Issue, Sept. 1991.

[4] ANSI/IEEE, IEEE Standard for Binary Floating-point Arithmetic, Std 754-2008, ANSI/IEEE, 2008.

[5] G. A. Kildall, A Unified Approach to Global Program Optimization, 1973.

[6] D. Monniaux, The Pitfalls of Verifying Floating-Point Computations, ACM, 2008.

[7] J. Souyris, V. Wiels, D. Delmas, H. Delseny. Formal Verification of Avionics Software Product, 2009.

[8] T. Granlund and the GMP development team. The GNU Multiple Precision Arithmetic Library, http://gmplib.org/, 2012.

[9] L. Fousse, G. Hanrot, V. Lefèvre, Patrick Pélissier and Paul Zimmermann. A Multiple-Precision Binary Floating-Point Library with Correct Rounding, ACM Transactions on Mathematical Software 2007.

[10] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic, ACM Comput. Surv., 1991.

[11] M. Martel, Semantics-Based Transformation of Arithmetic Expressions, 14th International Symposium, SAS 2007.

[12] A. Ioualalen and M. Martel. A New Abstract Domain for the Representation of Mathematically Equivalent Expressions, 19th International Symposium, SAS, 2012.

[13] N. Damouche, M. Martel and A. Chapoutot. Intra-procedural Optimization of the Numerical Accuracy of Programs, Formal Methods for Industrial Critical Systems, 20th International Workshop, 2015.

[14] P. Panchekha, Alex Sanchez-Stern, James R. Wilcox and Zachary Tatlock. Automatically improving accuracy for floating point expressions, Proceedings of the 36th ACM SIGPLAN Conference on Programming Language Design and Implementation, ACM, 2015.

[15] F. Jézéquel and J. M. Chesneaux, CADNA: a library for estimating round-off error propagation, Computer Physics Communications, 2008.

[16] J. C. Bajard, O. Beaumont, J. M. Chesneaux, M. Daumas, J. Erhel, D. Michelucci, J. M. Muller, B. Philippe, N. Revol, J. L. Roche, J. Vignes. Qualité des calculs sur ordinateurs, vers des arithmétiques plus fiables ?, Masson, 1997.

[17] G. Melquiond, Floating-point arithmetic in the Coq system, Inf. Comput., 2012.

[18] S. Boldo, J. C. Filliâtre and G. Melquiond, Combining Coq and Gappa for Certifying Floating-Point Programs, Intelligent Computer Mathematics, 16th Symposium, MKM, 2009.

[19] Modern Compiler Implementation in C, Cambridge University Press, 1998.

[20] Static Analysis by Abstract Interpretation of Numerical Programs and Systems, and FLUCTUAT, Static Analysis - 20th International Symposium, SAS, 2013.

[21] David Delmas, Eric Goubault, Sylvie Putot, Jean Souyris, Karim Tekkal, and Franck Védrine. Towards an industrial use of fluctuat on safety-critical avionics software, FMICS, 2009.