March 20, 2006

EVA-Flo

Évaluation et Validation Automatiques pour le calcul Flottant
New automatic tools for validated floating-point computation

Contents

1 State-of-the-art and objectives

This project addresses the need for better floating-point code, obtained faster, and with guarantees on the quality of the result. Indeed, floating-point arithmetic is now used everywhere, from embedded systems to supercomputers. In safety-critical applications such as avionics and automotive, offering a guarantee on the results of some floating-point computations is often crucial. In other applications, the critical factors may be accuracy, time-to-market, cost, or performance (speed, code size, memory requirements. . .) of the final product.

1.1 Automatic generation and validation of floating-point code

The project addresses the development of floating-point programs which, without sacrificing performance, offer *quantified* and *guaranteed* quality. Quality criteria may be *accuracy* (a small and bounded error), *reliability* (e.g. guarantee that over/underflows will not occur), *portability*, among others. Another essential goal of this project is to *automate* both development and validation.

Floating-point arithmetic used to be a mere set of cooking recipes that sometimes worked well, and sometimes did not work at all. Floating-point numbers were approximations of real numbers, rounding errors were a fuzzy concept, and finally floating-point programs were approximations of algorithms over the reals. Many numerical analysis techniques were developed to manage the accuracy of floating-point programs in this context.

More recently, thanks to widespread standards such as IEEE-754, floating-point numbers and operations have become well-defined mathematical objects. This new view is compatible with the previous one, but is much more powerful. For example, it enables to compute on-the-fly the rounding error of a critical operation as a floating-point number, to be incorporated in later computations (compensated algorithms)[1]. It also makes it possible to analyse the actual code in the last details – not an approximation of this code. It even enables the derivation of a formal proof of bit-level properties of the code, to be checked by a formal proof assistant.

The challenge of this new approach is that it requires considerable attention to the details: so far, it has been successfully applied to the development of very small programs (a few operations). One reason is that such studies are usually performed "manually" and on a case-by-case basis for each program, using human expertise. This is both error-prone, time-consuming and not reusable, since the work must be done from scratch each time the program is modified.

The purpose of the EVA-Flo project is therefore to extend this approach to larger floating-point programs. Arbitrary programs are still out of reach, and our method will be to focus on classes of problems for which floating-point experts are already able to derive efficient programs and/or useful proofs. Our ambition is to automate enough of this expertise to allow considering problems of much larger size than those considered so far. This automation will have several aspects: specification and analysis of the mathematical object to implement in floating-point, generation of efficient floating-point code, validation of properties of this code with respect to the initial specification. The same tools, of course, may be used to validate existing code.

1.2 Expertise of projects members

This project builds up on previous work in the Arenaire team, concerning in particular algorithms for the evaluation of elementary functions in hardware and in software (the latter for processors with or without a floating-point unit), polynomial and rational approximation under constraints,

[1]Members of the Arénaire team have defined a portable algorithm that returns the error of an FMA *(fused multiply-and-add)* instruction as the sum of two floating-point numbers.

the use of arbitrary precision interval arithmetic for validated numerics, algorithmic complexity, computer algebra algorithms and tools, formal proofs and proof-checking for floating-point operators and small sequences of operators, automatic or interactive construction of formal proofs for larger codes.

With three former members of Arenaire in the DALI team, there is a strong collaboration between both teams. DALI brings useful expertise in numerical error analysis, compensated algorithms, and formal proofs. They will also target our tools to exotic architectures such as graphics coprocessors.

The TROPICS team will bring theoretical expertise as well as a practical tool for automatic differentiation. This will be used to study the sensitivity of numerical code to rounding errors, to compute accurate approximations, and to improve interval analysis.

The CEA LIST team has an established expertise, implemented in the Fluctuat tool, about static analysis of floating-point code and automatic invariant computation. This work has also given them practical experience concerning larger, industrial codes.

1.3 Other related works

This project gathers members of the very strong French community in computer arithmetic. Other members of this community relevant to this project include

- the CADNA project at LIP6, working on probabilistic estimation of rounding errors,
- the CerPAN project (ANR Projet Blanc 2005), focusing on formal proofs for real and floating-point arithmetic in a scientific computing framework, which should be considered as a back-end or target for the tools we propose to build,
- people from computational geometry (the Galapagos ANR proposal) who share the dual concern of generation and validation of floating-point code.

The project will keep various links with this community (in particular through the AriNews working group organized twice a year) and with the French community in computer algebra. The Arénaire team is also involved in a sub-project (named SCEPTRE - Atelier du Futur) of the "Pôle de compétitivité mondial" MINALOGIC. Our industrial partner is the compilation group of STMicroelectronics in Grenoble. We will collaborate with STMicroelectronics on the aspects of code generation of this project.

To obtain guaranteed enclosures of a numerical result, one solution we will use and develop is interval arithmetic. The combined expertise in floating-point and interval arithmetic will yield results which are both tighter and really guaranteed, as illustrated by a collaboration with the COSY team at Michigan State University.

Other research teams with past or ongoing collaborations relevant to this project include those of Kahan at Berkeley (floating-point standardization), Rump at Hamburg Tech. Univ. (accurate summations algorithms), Shewchuk at Berkeley (accurate triangulation algorithms), and Yap at NYU (exact geometric computation). Industrial partners include STMicroelectronics, the NASA and EADS.

Formal proofs for floating-point operations are developed by constructors (Harrison at Intel, Russinov at AMD), that do not want to repeat the "Pentium bug" (an error in the floating-point division of the Intel processor, that cost them $475M). They have worked on the validation of individual operations, and on small sequences of operations, up to a complete elementary function, but always manually. When implementing elementary functions, they are also faced with some of the aspects of code generation relevant to our project (Tang at Intel, Markstein at HP, Toda at Sun, Ziv at IBM).

2 Description of the project and expected results

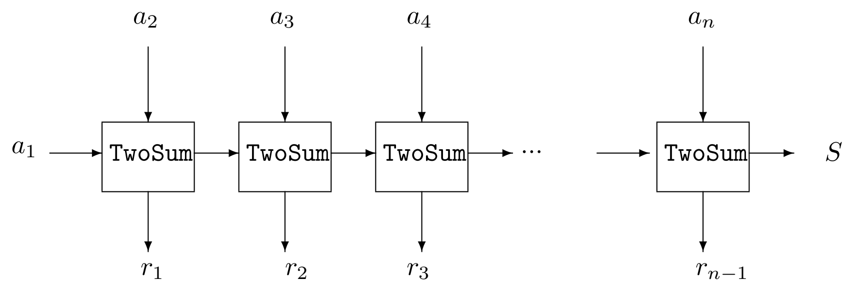
Floating-point arithmetic is widespread in numerical applications, and common programming languages require it to be present, either in hardware or emulated in software. Integer (or fixed-point) arithmetic often no longer suffices for numerical applications, and most general-purpose microprocessors and an increasing number of DSP chips have a floating-point unit. In this context, the behaviour of basic floating-point operators is governed by the IEEE-754 standard. Before going through the project in details, let us first see some central notions around this standard.

A few years ago, approaches for modelling floating-point arithmetic were "fuzzy approaches", describing it as a mere approximation to the arithmetic of real numbers. Such approaches allowed to get probabilistic or deterministic bounds to numerical errors, but were unable to catch the properties of subtle programs such as the one given by the following three C instructions:

```
s = (a1 + a2);   z = (s - a1);   r = (a2 - z);
```

If $|a_1| > |a_2|$, then on all systems of current commercial significance, this program—known in the literature as `FastTwoSum`—returns s equal to the floating-point number that is closest to $a+b$, and r equal to the (exact) error of the previous addition. These facts cannot be proven just by considering floating-point operations as mere approximations to the real operations: one has to use the specification of floating-point arithmetic provided by the IEEE-754 standard.

The IEEE-754 standard defines precisely the format of the floating-point numbers (e.g. single and double-precision), and defines four rounding modes (to nearest, or towards $-\infty$, $+\infty$, or 0). It also mandates *correct rounding* for the basic operations $+$, $-$, $\times$, $\div$ and for $\sqrt{}$, that is, the system must behave as if the result was first computed *exactly*, with infinite precision, and then rounded. The latter requirement has a number of advantages that will play key roles in the project. Correct rounding first leads to *full compatibility* between computing systems: a given program will produce the same values on different computers. Correct rounding also allows the computation of lower or upper bounds on the exact result of a sequence of arithmetic operations, and more generally leads to efficient implementations of *interval arithmetic*. Further, algorithms can also be specifically designed for benefiting from the property, especially for improved accuracy. One can for instance derive very accurate yet reasonably fast methods for adding many numbers (a crucial operation in linear algebra, integration, etc.). This is illustrated by the figure below, where each `TwoSum` box represents a generalization of the previous `FastTwoSum` sequence.



Taking $S+(r_1+r_2+\cdots+r_{n-1})$ for the result of the summation allows, in conventional double precision arithmetic when all the a_i's have the same sign, to add up to 2^{40} terms and yet keep an error less than the weight of the last bit of the returned result.

This example typically shows the possibilities offered by a thorough specification that are the foundations of EVA-Flo. The specification will make possible highly accurate and efficient algorithms for expression evaluation. It will enable us to establish, and formally check, the corresponding proofs of accuracy.

2.1 Description of the project

Most numerical codes compute data whose main qualities are to approximate "ideal" values that are generally unknown, and to be computed at a reasonable cost using standard floating point operation libraries (compared to other types of arithmetics) on a wide variety of machines and architectures. For more than twenty years, the IEEE-754 standard for basic arithmetic operators (presented above) has provided extra *qualitative properties* on computed data that we propose to carry over to the *expression* or *critical code* level (§2.1.1). Our new expression and code evaluation approach will lead to gains of *performance* and *accuracy* (§2.1.2).

The originality of our approach is also to allow *validation*, and a much higher level of *confidence in codes* than in the past. Better qualitative properties and specifications (correct rounding, error bounds, portability, ...) of numerical data are first ingredients in this direction.

Another crucial ingredient for producing reliable codes is a development methodology based of proof assistant tools. Indeed, modern techniques for floating-point expression evaluation lead to sophisticated algorithms whose proofs are complex and highly sensitive to slight changes of the codes. A human-based only process is therefore expensive and leads to unsecure codes. We aim at producing *better specified codes* whose proofs are *automatically generated and checked* in interaction with the algorithm developer.

A third ingredient for validated codes concerns all the components of the project, and requires a strong support for development forces. We propose to introduce *automation in every branches of our work*. We mean: automatic tools for *algorithm conception and code generation*, for instance automatic approximation generation, or automatic algorithmic choices depending on the target architecture; automatic tools for *specifying and checking properties of numerical data*, such as for computing certified error bounds; automatic assisting tools for *proof elaboration*, and use of highly-trusted automatic *proof checkers*. This "ubiquitous" automation approach relies on—and is made possible by—our savoir-faire and collaboration. The essential and immediate benefits will be to reduce the risk of dramatic failure at execution, to reduce the overall development cost, and increase the speed of the development accordingly.

These various aspects of efficient expression evaluation and validation organize the project in four directions that are presented in Sections 2.1.1-2.1.4.

Terminology. The acronym EVA-Flo stands for *Évaluation et Validation Automatiques pour le calcul FLOttant* (Automatic Evaluation and Validation of Floating-point computations).

By *floating-point computations*, we mean expressions or portions of codes we want to evaluate in floating-point. Such numerical expressions may contain arithmetic and algebraic operations, and elementary functions (sin, exp, atanh...). In codes we may accept branching (tests) and loops. However, we only target critical pieces of code, not large code solving general mathematical problems. For example, partial differential equation solvers answer a question of a more mathematical nature than just the evaluation of an expression.

By *evaluation*, we mean to compute a floating-point value close to the exact mathematical value. This exact value may be defined by a mathematical expression or a sequence of instructions in a specific programming language. The choice of a pertinent input format is a research subject of the project. A first step may be to construct an approximant of the mathematical quantity that exhibits good floating-point properties: this implies that both approximation and rounding errors will be considered. Another step is to determine a good way for evaluating the given approximant in order to minimize the rounding error, or the computing time, or a tradeoff between both.

Validation (or certification, or verification: the vocabulary does not seem to be totally fixed) corresponds to some guarantee on the quality of the evaluation. Several directions will be considered, for instance the full error (approximation plus rounding errors) between the exact mathematical value and the computed floating-point result, or some guarantee on the range of a function. Validation also comprises a proof of this guarantee that is checked by a proof checker.

Automation is crucial since most development steps require specific expertise in floating-point computing that can neither be required from code developers nor be mobilised manually for every problems. An automated evaluation tool with guaranty should incorporate the knowledge accumulated by the members of this EVA-Flo project to offer high quality results at each step. Nevertheless, we anticipate that such a tool will not be totally automatic, but should rather interact with the author of the floating-point expression to be handled. How this interaction should take place? Which information is relevant for efficient automation? How to design an easy-to-use tool? These are the first questions we have to investigate.

2.1.1 Specification of qualitative aspects of floating-point codes

A first goal of EVA-Flo is better formalism and specifications for floating-point evaluation. Before developing this aspect, let us remark that since the *validation process* is performed once for a given code, then it can be (relatively) time-consuming compared to the execution time. Let us also note that even if we primarily aim at *a priori* validation, validation tools can also be applied *a posteriori*. Formalism and specification will especially concern the following quality aspects.

Specification. Typically, this will mean a proven error bound between the value computed by the program and a mathematical value specified by the user in some high-level format. For example with $\log(1 + e^x)$, this may concern the distance between the code output, and the mathematical value of $\log(1 + e^x)$ rather than the floating-point value specified by language standards. We will consider the proven error bound as being part of the *specification* of the evaluation program.

Tight error bound computation. The total error between a mathematical expression and the computed floating-point result is usually split as the sum of the approximation and the rounding errors. Formalisms such as *interval analysis* and *condition numbers* are useful for both types of errors, as well as automatic differentiation and its application to sensitivity analysis. For designing efficient estimation tools, new results on the *complexity of computing error bounds* should also be derived, based on concepts such as the complexity of computing derivatives.

Floating-point issues. Regarding the use of floating-point arithmetic, a frequent concern is the portability of code, and thus the reproducibility of computations. Successive roundings (with different intermediate precisions) can be sources of problems. A different concern is whether underflows or overflows occur. To detect the possible occurence of these problems, our automatic tool will need information on the input domains of variables, and also on the architecture and compiler.

Precision. Floating-point properties, such as the `FastTwoSum` (Introduction of §2) have led to algorithms that yield accurate results, as if extra computing precision were available. The choice of the method (compensated algorithm versus double-double versus quadruple precision for instance) that will yield the required accuracy at given or limited cost must be studied. An automatic tool will have either to be told or to decide when and how to rely on these methods.

Input domains and output ranges. The determination of input domain or output range also constitutes a specification/guarantee of a computation. Since the quality of an approximant depends on the input domain, this also enables optimizations during code generation. Interval arithmetic and Taylor models will be key tools here. Automatic differentiation will be used for Taylor expansions of order 1 whose evaluation using interval arithmetic give tighter enclosures of results. An automated tool for determining intput or output intervals has to be notified of what is known and what is sought.

Other arithmetics, dedicated techniques and algorithms for increased precision. For studying the quality of the results, most of conception phases will require *multiple-precision* or *exact* solutions to various algebraic problems. This will ask us further our understanding of the practical and theoretical costs of the corresponding arithmetics. Our automatic tool should dispose of a panel of arithmetics and dedicated algorithms.

The tools developed here for identifying, measuring, and checking properties will be in tight interaction with the proof process of the overall expression evaluation (addressed in §2.1.3).

2.1.2 Algorithms for the evaluation of floating-point expressions

Another goal of EVA-Flo is to *improve* and *automate* the evaluation of floating-point expressions. This will reduce to computing good approximants (i.e which both involve only machine-representable coefficients and minimize the errors), and evaluating them fast and accurately. Here, there are many challenges to the automation and we plan to proceed in three steps, as follows.

Good approximants with respect to the method error. First we will focus on the elaboration of automatic processes to get polynomials (or rational fractions or sums of cosines) whose coefficients fit the constraints imposed by the application and that approximate well the given function with respect to the absolute or relative error. Our aim is to be able to produce either very quickly good approximants inside an on-the-fly process of function evaluation (this is the main target here) or the best possible approximants (several hours or days for computing it are then acceptable: we aim here at computing it once and then use it millions, if not billions, of times). The tools involved include linear programming and lattice reduction.

Algorithms for evaluating approximants. Once we have a good approximant, how to *evaluate* it as *accurately and efficiently* as possible in IEEE-754 floating-point arithmetic? Such an approximant is typically a polynomial evaluated with Horner's method. However, several other evaluation algorithms exist, which are much faster. Our first goal is thus to study the accuracy of such algorithms. In particular, an expected conclusion is that some of these *fast algorithms* may be *also highly accurate.* Our second goal is, given a target accuracy like the one wanted for correctly rounded elementary functions, to design algorithms that, while achieving this accuracy, are faster than the algorithms in use today. To obtain such an improvement we shall combine three approaches: design *hybrid algorithms* that are halfway between Horner-like methods and the fast methods; exploit the known proven *properties* of the arithmetic model; exploit the *structure* of the approximant. This whole algorithmic design process will be only possible if one can routinely compute certified error bounds; it thus requires some of the *validation tools* of §2.1.1. Furthermore, we shall consider the impact of enriching the floating-point model with higher-level *proven properties*, such as FMA with correct rounding. Finally, the fast algorithms mentioned above usually assume a so-called *preconditioning* phase, which consists in solving systems of linear or polynomial equations over the rationals. Hence a need for further complexity results in exact and multi-precision arithmetics.

Automated expression evaluation. Once the two tasks above are completed, the ultimate goal is to be able to perform them automatically so as to produce a *best approximation code* with respect to both approximation and roundoff errors. This will be highly *context-dependent.* In particular, we will have to detect at compile-time for which expressions of a numerical program we should compute on-the-fly special expressions (typically, expressions that will be very frequently evaluated at run-time). When a compound function is computed, one could directly build approximations to that function, instead of using several consecutive approximations. This would improve accuracy and speed, and possibly usage of memory caches. We also need to estimate the *domain of the input* variables of these expressions and to exploit fully the *instruction set of the target* processor. This is obviously true for special purpose processors such as DSPs, but also for more "general" microprocessors such as PowerPCs and Itaniums, where an FMA instruction is available.

2.1.3 Interface with automatic proof checkers

The process presented in the preceding pages aims at providing a fully specified, well optimized and fairly robust piece of code. In most situations the code produced could be used with no further questioning. Yet, some users may want to obtain a higher level of certification as will be provided by the work described in this section. We propose to investigate two paths. Both paths add transparencies to a process that we want to present as an all-inclusive black box to anybody that is not highly trained in computer arithmetic or numerical analysis.

Formal proofs. We will provide formal proofs that can be checked by an independent highly-trusted automatic proof checker. This approach has been coined as *invisible formal methods* as we attain the certification level of formal methods without requiring any specific training from end users. Work initiated in previous projects will allow us to use the Coq and the PVS automatic proof checkers. Our burden to obtain a formal proof of correctness will be greatly reduced by the fact that we will produce codes with our own automatic tools. Where a project such as ANR CerPAN would have to infer many properties, as experts we will have no difficulty to provide tight and numerous code annotations about arithmetic properties required for proofs of correctness. Hence, a main goal is to obtain quickly an accurate and efficient set of annotations so that producing the proof becomes a much simpler task. This goal will be met by collaborations between tool and proof generator developers. Members EVA-Flo have already experienced this type of collaboration, especially for designing the Gappa tool. We may still have to pass some annotations from one tool to the next one, such a process has been presented as *invariant translation*. The annotation approach will be complemented by the use of static analysis with Fluctuat.

Semantics. We will also express some code manipulations related to floating-point arithmetics in a formal semantics framework, that is as semantics based program transformation. Providing semantics will help to promote our code transformations to people trained in that field as people in charge of compiler developments, for example for certified compilers like Scade. Our goal is twofold as we bring not only transparency but we also encourage the integration of some of our techniques into compilers. As a mean to settle *expert knowledge*, we will translate some expert practices into semantics based program transformations and heuristics.

2.1.4 Integrated and interoperable automatic tools

Various automatic components have been somehow independently introduced in Sections 2.1.1-2.1.3, that correspond to the three main complementary steps of the design of a floating-point code for the evaluation of an expression:

i/ Tools for I/O domains, accuracy, and specification study.

ii/ Tools for approximation/evaluation algorithms and code generation.

iii/ Tools for proof design and check.

A main objective of EVA-Flo is to study how the tools—possibly seen as black boxes—will collaborate for providing an entire automatic approach taking in input an expression to evaluate (with possible annotations), and returning an executable validated code. The complete automation with optimal or at least good resulting performance seems to be far beyond the current knowledge. However, we see our objective as a major step for prototyping future compilers. We thus aim at developing a piece of software that automates the steps described in the previous pages. The result should be an easy-to-use integrated environment, and for this purpose we need the help and manpower of an engineer devoted to this development.

Especially, the project will propose interactive exchanges with the user, and code annotations. For proof generation, one possible approach is proof generation from the execution traces of other used tools (for instance traces of a rounding error calculator). Since many tools are planned to be involved in the general conception process, the proof elaboration process needs to be discussed and implemented jointly. A key motivation here is the sensitivity of the proof generation with respect to changes at the algorithm or the code level.

2.2 Qualification of the proponents and collaborative added value

Arénaire, Lyon. Arénaire team (LIP, Lyon) expertise is computer arithmetic. One of its research directions concerns the properties of floating-point arithmetic. Arénaire is also well known for its results on the evaluation of elementary functions with correct rounding, which requires the determination of a good approximant and an implementation that exploits the properties of the floating-point arithmetic. Other research directions of Arénaire include interval arithmetic or algorithmic complexity. Arénaire takes part in the "Pôle de compétitivité mondial" MINALOGIC where its role is to generate automatically, at compile-time, code for the floating-point evaluation of composite functions for embedded systems. EVA-Flo generalizes the latter to more general expressions and processors, together with more emphasis on validation.

DALI, Perpignan. DALI develops research on computer arithmetic and computer architecture. The goal of the team is to propose higher performance architectures and softwares to deliver more reliable results when rounding error cannot be avoided. The collaborative added value within EVA-Flo includes compensated algorithms (new algorithms with dynamic and validated error bounds, detailed performance analysis with and without processor dependence), formal proof checkers (invisible formal methods, semantic extensions), emerging architectures and exotic arithmetics (graphical processor units, non-IEEE 754 floating-point arithmetic). DALI takes part with CEA-LIST to the cooperating project OVALIE of the "Pôle de compétitivité mondial" Aerospace Valley (Midi-Pyrénées and Aquitaine) including industrial partners as Airbus, Alcatel, EADS, Siemens and PSA. OVALIE stands for "verification tools with static analysis of embedded software".

TROPICS, Sophia-Antipolis. The Tropics team develops an AD tool named "Tapenade", which differentiates programs written in Fortran (and soon in C). This tool provides directional derivatives, gradients, and Jacobian matrices. These derivatives actually help refining the intervals that represent the influence of truncation errors. Extension of Tapenade to provide second derivatives (Hessians) may help refine these intervals further. Tapenade can also be used to detect the fragments of a code that are most sensitive to truncation errors. From this work, Tropics expects a better insight of trucation error evaluation and new application domains for Tapenade. Tropics will also bring experience in static analysis of source programs, and hopes to find interesting new analyses to study and incorporate in its tool.

CEA LIST (Fluctuat tool), Saclay. The Fluctuat static analyzer automatically computes invariants related to the numerical precision of large size C and assembly codes. Concerning numerical algorithms, for example, it has been used to bound the maximal number of iterations of some Newton algorithms for any input in a large range. In this project Fluctuat will be used for validation, in interaction with Gappa. In particular, we aim at defining, jointly with the other partners of the project, how both tools should cooperate, how Fluctuat should be used and improved in the context of the certification of the implementation of mathematical functions.

Added value of the collaboration. Each team of the EVA-Flo project is expert on one or several aspects developed in §2.1 and has developed pieces of software that correspond to each field. The ongoing collaboration of some groups of EVA-Flo, with main efforts for collaborative softwares, has led to the CRlibm library. CRlibm is an efficient mathematical IEEE-754 compliant library with proven correct rounding. Its development has only been possible with an heavy use of the Gappa tool that is intended to help verifying and formally proving properties on numerical programs. This first practical and successful experience, with a strong interaction between the two different domains of floating-point arithmetic and of formal proving, is an important element for proposing our new research directions.

2.3 Dissemination of results

Academic partners We plan to publish our results regularly and to present them at conferences (we plan one conference or two per partner and per year), relevant in our field, such as Arith, RNC, SCAN, ISSAC, ICMS. We also plan to apply for the organization of the next SCAN conference (*GAMM - IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*) in 2008.

Industrial partners Our complementary partnership with ST Microelectronics is important for us[2]: they bring problems and applications, and we can try our solutions on their platform. Also, this project obviously requires tight relations with a good compilation team. Our first code generations solutions will be tried on the C compilers for the ST200 familly.

Other partners are concerned by our future tools: current users of Fluctuat in particular, people from control theory (for robustness issues even when the captors, and thus the input data, are not accurate), people form computational geometry (cf. the ANR "projet blanc" Galapagos proposal) etc

2.4 Methodology and schedule

The working methodology will consist in frequent and long meetings (2-3 days, 4 times per year) to elaborate together on the automation part mainly, i.e. on the interaction between each constitutive component (such as approximant generation or output range determination). Each component will be developed by the corresponding expert and our mutual effort will be on the definition of what should be communicated by one component to the other. Related issues, such as the input language for the mathematical expression, will also be handled collectively.

The planned schedule is the following.

- *First year:* starting from small examples (taken from embedded systems or from numerical codes analyzed by Fluctuat), we shall:
  - develop the underlying techniques necessary to reach our goals, which are not yet formalized: choice of a computing precision, of an evaluation algorithm... ;

[2]Recall that we collaborate on a joint project in the "Pôle de compétitivité mondial" MINALOGIC.

  - identify our needs in validation: for instance, do we need input domain, output range, both? and how they can be fulfilled;
  - define how automation can be reached: what each component requires as input, what it can produce as output, for instance "what kind of annotation should be provided by the bounding rounding error component to the proof checker".

- *Second year:* we shall then implement the techniques and put into practice the methodology elaborated during the first year. Also, we shall refine the interaction part. The PhD student(s) shall start here, since the directions of research will be clearly identified at that stage. The engineer shall start there, since the development needs will be identified at that stage.
- *Third year:* what will be started in the second year will be completed during the third year. The integration of the various components is achieved during this year and the effort on automation comes to a conclusion.
- *Fourth year:* this last year will be devoted to apply our results and tools to real life examples, testing and refinements.

3 Motivation of expected funding

The project's goals in terms of software development are ambitious: it should produce a working prototype of a complete tool, aimed at non-specialists, allowing them to write and validate expert floating-point code in reduced time.

It should be noted that several projects members are already involved in relevant software projects : Gappa, Linbox, MPFI and CRLibm at Arenaire, Fluctuat at CEA/List, Tapenade at TROPICS. These existing tools are usually complementary. The EVA-Flo project, however, is not simply their integration in an heterogeneous toolbox. Firstly, the relevant aspects of these various tools should be integrated in a consistent tool, with a consistent interface. Secondly, the project should fill the gaps that currently exist between the existing approaches.

These aspects of software development should be managed by a professional software engineer with a long-term view of the project: he/she should be recruited for 3 years. This engineer will be working with Arénaire in Lyon.

Some more theoretical aspects of the project will be managed by a PhD student in the DALI team, and under shared supervision of DALI and CEA/List. Funding is requested for this student who will focus on formal proof tools (Why, Gappa and Fluctuat interoperability, enhancements and extensions).

On the equipment side, EVA-Flo will fund workstations for the abovementionned engineer and PhD student, and one workstation for each participating team. In addition, it will fund the acquisition of development boards for exotic hardware targets, such as embedded processors and DSPs.

Finally, EVA-Flo will fund regular meetings of project members in France (roughly one three-day mission for each project member each year). Some of this funding will also allow the engineer to spend some time in the various teams in order to coordinate the software aspects. It will also fund each year one mission to an international conference for each team.