

Certification of bounds on expressions involving rounded operators

Marc Daumas

LIRMM (UMR 5506 CNRS-UM2), visiting ÉLIAUS (ÉA 3679 UPVD)

and

Guillaume Melquiond

LIP (UMR 5668 CNRS-ÉNS Lyon-INRIA)

Gappa uses interval arithmetic to certify bounds on mathematical expressions that involve rounded as well as exact operators. Gappa generates a theorem with its proof for each bound treated. The proof can be checked with a higher order logic automatic proof checker, either Coq or HOL Light, and we have developed a large companion library of verified facts for Coq dealing with the addition, multiplication, division, and square root, in fixed- and floating-point arithmetics. Gappa uses multiple-precision dyadic fractions for the endpoints of intervals and performs forward error analysis on rounded operators when necessary. When asked, Gappa reports the best bounds it is able to reach for a given expression in a given context. This feature is used to quickly obtain coarse bounds. It can also be used to identify where the set of facts and automatic techniques implemented in Gappa becomes insufficient. Gappa handles seamlessly additional properties expressed as interval properties or rewriting rules in order to establish more intricate bounds. Recent work showed that Gappa is perfectly suited to the proof of correctness of small pieces of software. Proof obligations can be written by designers, produced by third-party tools or obtained by overloading arithmetic operators.

Categories and Subject Descriptors: G.4 [**Mathematical Software**]: Certification and Testing

General Terms: Interval Arithmetic, Floating Point, Proof System

Additional Key Words and Phrases: Forward error analysis, Dyadic fraction, Coq, PVS, HOL Light, Proof obligation

1. INTRODUCTION

Gappa is a simple and efficient tool to certify bounds in computer arithmetic [Revy 2006] and in the engineering of numerical software [de Dinechin et al. 2006; Melquiond and Pion 2007; Daumas and Giorgi 2007] and hardware [Michard et al. 2006]. Gappa bounds arithmetic expressions on real and rational numbers and their evaluations in computers on fixed- and floating-point data formats. Properties that are most often needed involve:

- ranges of rounded expressions to prevent exceptional behaviors (overflow, division by zero, and so on),
- ranges of absolute and/or relative errors to characterize the accuracy of results.

To the best of our knowledge, Gappa is a tool that was missing in computer arithmetic and related research areas. On one hand, Gappa is not the first tool able to certify static ranges and error bounds. Two other projects are currently mixing interval arithmetic and automatic proof checking [Gameiro and Manolios 2004; Daumas et al. 2005]. The first one uses ACL2 [Kaufmann et al. 2000] and the second one uses PVS [Owre et al. 1992]. Gappa is, however, the first tool able

to certify these bounds when the program relies on advanced numerical recipes like error compensation, iterative refinement, *etc.*

On the other hand, countless efficient algorithms use symbolic computation or interval arithmetic to produce bounds on expressions but seldom provide gateways to automatic proof checkers. The continuing work on interval arithmetic [Neumaier 1990; Jaulin et al. 2001] has created a huge set of useful techniques to deliver accurate answers in a reasonable time. Each technique is adapted to a specific class of problems and most evaluations yield accurate bounds only if they are handled by the appropriate techniques in the appropriate order. Blending interval arithmetic and properties on dyadic fractions has also been heavily used in computer arithmetic [Rump et al. 2005].

Integration is certainly the challenge that prevented the development of competitors to Gappa. Proofs generated by Gappa typically contain 4,800 of lines [de Dinechin et al. 2006] and related projects were not able to avoid the development of small programs, for example to generate a proof script about 9,935 intervals each requiring 3 theorems in PVS [Daumas et al. 2005]. The functionalities of Gappa presented here show its potential in tackling generic problems that are unreachable with other available tools. Our goal is to

- Provide *invisible formal methods* [Tiwari et al. 2003] in the sense that Gappa delivers formal certificates to users that are not expected to ever write any piece of proof in any formal proof system.
- Provide a tool that is able to consider and combine many techniques using interval arithmetic, dyadic fractions, and rewriting rules. Gappa performs an exhaustive search on its built-in set of facts and techniques. It is also able to follow hints given by users to take into account new techniques.
- Simplify a valid proof once it has been produced in order to reduce the certification time, as in-depth proof checking is and will remain much slower than simple C++ evaluation.
- Provide a tool appropriate to meet the highest Common Criteria Evaluated Assurance Level (EAL 7) [Schlumberger 2003; Rockwell Collins 2005] for numerical applications using floating- and fixed-point arithmetics.

Gappa is composed of two parts. First, a program written in C++, based on Boost interval arithmetic library [Brönnimann et al. 2003] and MPFR [Fousse et al. 2005], verifies numeric properties given by the user. Along these verifications, it generates formal certificates of their validity. Second, a companion library provides theorems with computable hypotheses. This set of theorems allows a proof assistant to interpret the formal certificates and hence to automatically check the validity of the numeric properties. The proof assistant we use is Coq [Huet et al. 2004], but ongoing work shows that Gappa can generate formal certificates for other proof assistants such as HOL Light [Harrison 2000].

We first describe the input language of Gappa and we detail its built-in rewriting rules. We then present the set of theorems and interval operators Gappa relies on to prove numeric properties and we describe how it interacts with proof checkers, extending [Daumas and Melquiond 2004]. We finish this report with perspectives, experiments, and concluding remarks.

2. DESCRIPTION OF THE INPUT LANGUAGE OF GAPPA

Consider for example that y is the result of a portion of code without loops and branches. The definition of y is an expression involving rounded operators and rounded constants. We may define Y (uppercase) as the exact answer without any rounding error. The expression Y is identical to y except that rounded operators are replaced by exact operators and rounded constants are replaced by exact constants. If some numeric terms were considered negligible and were optimized out of the implementation y , these terms are introduced in Y . So the expression y gives the effectively computed value while the expression Y gives the ideal value y tries to approximate.

In order to certify the correctness of this code, we will possibly need

- an interval containing all the possible values of y to guarantee that y does not overflow and produces no invalid value,
- an interval containing all the possible values of $y - Y$ or $(y - Y)/Y$ to guarantee that y is accurate and close to Y .

The grammar of the input language to Gappa is presented in Figure 1. It has been designed to efficiently express such needs. An input file is composed of three parts: a set of aliases (PROG, detailed in Section 2.2), the proposition to be proved (PROP, detailed in Section 2.1) and a set of hints (HINTS, detailed in Sections 2.3 and 2.4). When successful, Gappa produces a Coq or a HOL Light file with the proof of PROP. Its validity can be checked by Coq using the companion library and by HOL Light using a set of axioms until a companion library becomes available.

2.1 Formalizing the proposition (PROP) that Gappa proves

The proposition (PROP) that Gappa is expected to prove is written between brackets ($\{ \}$) as presented below and it may contain any conjunction (AND token: \wedge), disjunction (OR token: \vee), implication (IMPL token: \rightarrow) or negation (NOT token: `not`) of enclosures of expressions. Enclosures are either inequalities (LE or GE tokens: \leq or \geq) or bounded ranges (IN token: `in`) on expressions (REAL). Ranges may be left unspecified by using question marks (?) instead of intervals. Endpoints of intervals and bounds of inequalities are numerical constants (SNUMBER).

```
{ x - 2 in [-2,0] /\ (x + 1 in [0,2] -> y in [3,4])
  -> not x <= 1 \/ x + y in ? }
```

Expressions (REAL) may contain constants (SNUMBER), identifiers (IDENT), user-defined as well as built-in rounding operators (FUNCTION), and arithmetic operators (addition, subtraction, multiplication, division, absolute value, square root, negation, and fused multiply and add).

The goal of Gappa is to prove the whole logical proposition, assuming that undefined identifiers (x and y in the example above) are universally quantified over the set of real numbers. If question marks are used in some expression enclosures, Gappa suggests intervals for these enclosures such that the proposition can be proved. In the example above, Gappa suggests $x + y \in [3, 5]$, which happens to be the tightest interval such that the proposition holds true. Question marks are not allowed if they induce unspecified hypotheses once transformations of Section 3.1 have been applied.

```

0 $accept: BLOB $end
1 BLOB: PROG '{' PROP '}' HINTS
2 PROP: REAL LE SNUMBER
3     | REAL IN '[' SNUMBER ',' SNUMBER ']'
4     | REAL IN '?'
5     | REAL GE SNUMBER
6     | PROP AND PROP
7     | PROP OR PROP
8     | PROP IMPL PROP
9     | NOT PROP
10    | '(' PROP ')'
11 SNUMBER: NUMBER
12        | '+' NUMBER
13        | '-' NUMBER
14 FUNCTION_PARAM: SNUMBER
15                | IDENT
16 FUNCTION_PARAMS_AUX: FUNCTION_PARAM
17                      | FUNCTION_PARAMS_AUX ',' FUNCTION_PARAM
18 FUNCTION_PARAMS: /* empty */
19                 | '<' FUNCTION_PARAMS_AUX '>'
20 FUNCTION: IDENT FUNCTION_PARAMS
21 EQUAL: '='
22        | FUNCTION '='
23 PROG: /* empty */
24       | PROG IDENT EQUAL REAL ';'
25       | PROG '@' IDENT '=' FUNCTION ';'

26 REAL: SNUMBER
27     | IDENT
28     | FUNCTION '(' REALS ')'
29     | REAL '+' REAL
30     | REAL '-' REAL
31     | REAL '*' REAL
32     | REAL '/' REAL
33     | '|' REAL '|'
34     | SQRT '(' REAL ')'
35     | FMA '(' REAL ',' REAL ',' REAL ')'
36     | '(' REAL ')'
37     | '+' REAL
38     | '-' REAL
39 REALS: REAL
40       | REALS ',' REAL
41 DPOINTS: SNUMBER
42         | DPOINTS ',' SNUMBER
43 DVAR: REAL
44       | REAL IN NUMBER
45       | REAL IN '(' DPOINTS ')'
46 DVARs: DVAR
47       | DVARs ',' DVAR
48 HINTS: /* empty */
49        | HINTS REAL IMPL REAL ';'
50        | HINTS REALS '$' DVARs ';'
51        | HINTS '$' DVARs ';'
52        | HINTS REAL '~' REAL ';'

```

Fig. 1. Grammar of the input language to Gappa generated by byson

As Gappa stores interval endpoints as dyadic fractions, it produces an error message when a goal contains an interval so tight that it has to be replaced with an empty interval. For example, Gappa is unable to prove the goal $13/10$ in $[1.3, 1.3]$, as the empty set is the biggest representable subset of the set $\{1.3\}$.

The fact that bounds are numerical constants is not a strong limitation to the use of Gappa. For example, linear dependencies on intervals can be introduced by manipulating expressions: the enclosure $y - Y \in [-i \times 10^{-6}, i \times 10^{-6}]$ is not allowed, but the enclosure $(y - Y)/i \in [-10^{-6}, 10^{-6}]$ is.

2.2 Definitions of aliases to describe the behavior of programs (PROG)

Typing large expressions in the proposition (PROP seen Section 2.1) would not be practical for proof obligations generated from actual pieces of software. Aliases (IDENT) of expressions (REAL) are defined by constructions of the form `IDENT = REAL`. `IDENT` becomes available for later definitions, the proposition, and the hints. This construction is neither an equality nor an affectation but rather an alias. Gappa uses `IDENT` for its outputs and in the formal proof instead of machine generated names. An identifier cannot be aliased more than once, even if the right hand sides of both aliases are equivalent. Neither can it be aliased after having been used as an unbound variable. For example `b = a * 2; a = 1;` is not allowed.

Rounding operators are used in the arithmetic expressions describing the behavior of numerical codes. They are real functions yielding rounded values according to the target data format (`precision` and `minimum_exponent`, or `lsb_weight`) and a predefined rounding mode amongst the ones presented Table I. For modes that are not defined by IEEE 754 standard [Stevenson et al. 1987] and its forthcoming revision, see [Even and Seidel 1999; Boldo and Melquiond 2005] and references herein. Floating- and fixed-point rounding operators can be expressed with the following operators where rounding parameters (`FUNCTION_PARAMS`) are listed between angle brackets:

```
float< precision, minimum_exponent, rounding_direction >(...)
fixed< lsb_weight, rounding_direction >(...)
```

The syntax above can be abbreviated for the floating-point formats of Table II and for (fixed-point) integer arithmetic:

```
float< name, rounding_direction >(...)
int< rounding_direction >(...)
```

Aliases are permitted for rounding operators. Their definitions are prefixed by the '@' sign. Line 1 below defines the `rnd` function as rounding to nearest using IEEE 754 standard for 32 bit floating-point data. The example shows various ways of expressing rounded operations using the alternate constructions of `EQUAL`. When all the arithmetic operations on the right hand side of an alias are followed by the same rounding operator (as visible Line 2), this operator can be put once and for all on the left of the equal symbol (as presented Line 3). On this example, Gappa even complains that y and z are two different names for the same expression.

```
1 @rnd = float< ieee_32, ne>;
2 y = rnd(x * rnd(1 - x));
3 z rnd= x * (1 - x);
```

Table I. Rounding modes available in Gappa

| Alias | Meaning |
|-----------|--|
| zr | toward zero |
| aw | away from zero |
| dn | toward minus infinity (down) |
| up | toward plus infinity |
| od | to odd mantissas |
| ne | to nearest, tie breaking to even mantissas |
| no | to nearest, tie breaking to odd mantissas |
| nz | to nearest, tie breaking toward zero |
| na | to nearest, tie breaking away from zero |
| nd | to nearest, tie breaking toward minus infinity |
| nu | to nearest, tie breaking toward plus infinity |

Table II. Predefined floating-point formats available in Gappa

| Alias | Meaning |
|-----------------|------------------------------|
| ieee_32 | IEEE-754 single precision |
| ieee_64 | IEEE-754 double precision |
| ieee_128 | IEEE-754 quadruple precision |
| x86_80 | 80 bit extended precision |

Most truncated hardware operators [Texas Instruments 1997] and some compound operators cannot be described as if they were first computed to infinite precision and then rounded to target precision. For such operators we revert to under-specified functions that produce results with a known bound on the relative error.

```
{add|sub|mul}_rel < precision [, minimum_exponent] >(..., ...)
```

If a minimum exponent is provided, Gappa does not instantiate any assumption that involves a result with an exponent below the minimum exponent. Otherwise, the error bound always holds and the absolute error is 0 when the result is 0.

2.3 Rewriting expressions to suppress some dependency effects (first use of HINT)

Let Y be an expression and y an approximation of Y due to round-off errors, for example. The absolute error is $y - Y$ and the relative error is $(y - Y)/Y$. As soon as Gappa has computed some ranges for y and Y , it naively computes an enclosing interval of $y - Y$ and $(y - Y)/Y$ using theorems on subtraction and division of intervals.

Unfortunately, expressions y and Y are strongly correlated and error ranges computed that way are useless. To suppress some dependency effects and reproduce many of the techniques used in numerical analysis and in computer arithmetic [Kahan 1965; Higham 2002; Boldo and Daumas 2004; de Dinechin et al. 2004], Gappa manipulates error expressions through a set of built-in pattern-matching as well as user-defined rewriting rules.

We assume that $y = \text{rnd}(a + b)$ and $Y = A + B$. Gappa rewrites the absolute error $\text{rnd}(a + b) - (A + B)$ as $(\text{rnd}(a + b) - (a + b)) + ((a + b) - (A + B))$. It finds an enclosure of the first term using a theorem on the `rnd` rounding operator. For

Table III. Built-in rewriting rules available in Gappa

| Rule | Before | After | Condition |
|------------|------------------------------------|---|--|
| opp_mibs | $-a - -b$ | $-(a - b)$ | $a \neq b$ |
| opp_mibs | $(-a - -b) / -b$ | $(a - b) / b$ | $b \neq 0 \wedge a \neq b$ |
| add_xals | $a + b$ | $(a - A) + (A + b)$ | |
| add_xars | $c + a$ | $(c + A) + (a - A)$ | |
| add_mibs | $(a + b) - (c + d)$ | $(a - c) + (b - d)$ | $a \neq c \wedge b \neq d$ |
| add_fils | $(a + b) - (a + c)$ | $b - c$ | $b \neq c$ |
| add_firs | $(a + b) - (c + b)$ | $a - c$ | $a \neq c$ |
| sub_xals | $a - b$ | $(a - A) + (A - b)$ | $a \neq b \wedge A \neq b$ |
| sub_xars | $b - a$ | $(b - A) + -(a - A)$ | $b \neq a$ |
| sub_mibs | $(a - b) - (c - d)$ | $(a - c) + -(b - d)$ | $a \neq c \wedge b \neq d$ |
| sub_fils | $(a - b) - (a - c)$ | $-(b - c)$ | $b \neq c$ |
| sub_firs | $(a - b) - (c - b)$ | $a - c$ | $a \neq c$ |
| mul_xals | ab | $(a - A)b + Ab$ | |
| mul_xars | ba | $b(a - A) + bA$ | |
| mul_fils | $ab - ac$ | $a(b - c)$ | $b \neq c$ |
| mul_firs | $ac - bc$ | $(a - b)c$ | $a \neq b$ |
| mul_mars | $ab - cd$ | $a(b - d) + (a - c)d$ | $a \neq c \wedge b \neq d$ |
| mul_mals | $ab - cd$ | $(a - c)b + c(b - d)$ | $a \neq c \wedge b \neq d$ |
| mul_mabs | $ab - cd$ | $a(b - d) + (a - c)b + -((a - c)(b - d))$ | $a \neq c \wedge b \neq d$ |
| mul_mibs | $ab - cd$ | $c(b - d) + (a - c)d + (a - c)(b - d)$ | $a \neq c \wedge b \neq d$ |
| mul_filq | $(ab - ac) / (ac)$ | $(b - c) / c$ | $ac \neq 0 \wedge b \neq c$ |
| mul_firq | $(ab - cb) / (cb)$ | $(a - c) / c$ | $bc \neq 0 \wedge a \neq c$ |
| div_mibq | $(a/b - c/d) / (c/d)$ | $((a - c) / c - (b - d) / d) / (1 + (b - d) / d)$ | $bcd \neq 0 \wedge b \neq d$ |
| div_firq | $(a/b - c/b) / (c/b)$ | $(a - c) / c$ | $bc \neq 0 \wedge a \neq c$ |
| sqr_mibs | $\sqrt{a} - \sqrt{b}$ | $(a - b) / (\sqrt{a} + \sqrt{b})$ | $a \geq 0 \wedge b \geq 0 \wedge a \neq b$ |
| sqr_mibq | $(\sqrt{a} - \sqrt{b}) / \sqrt{b}$ | $\sqrt{1 + (a - b) / b} - 1$ | $a \geq 0 \wedge b > 0 \wedge a \neq b$ |
| sub_xals | $b - A$ | $(b - a) + (a - A)$ | $A \neq b \wedge a \neq b$ |
| err_fabq | $1 + (a - b) / b$ | a / b | $b \neq 0 \wedge a \neq b$ |
| val_xabs | a | $A + (a - A)$ | |
| val_xebs | A | $a + -(a - A)$ | |
| val_xabq | a | $A(1 + (a - A) / A)$ | $a \neq 0$ |
| val_xebq | A | $a / (1 + (a - A) / A)$ | $ab \neq 0$ |
| square_sqr | $\sqrt{a} \times \sqrt{a}$ | a | $a \geq 0$ |
| addf_1 | $a / (a + b)$ | $1 / (1 + b / a)$ | $a(a + b) \neq 0 \wedge a \neq 1$ |
| addf_2 | $a / (a + b)$ | $1 - 1 / (1 + a / b)$ | $b(a + b) \neq 0 \wedge a \neq 1$ |
| addf_3 | $a / (a - b)$ | $1 / (1 - b / a)$ | $a(a - b) \neq 0 \wedge a \neq 1$ |
| addf_4 | $a / (a - b)$ | $1 + 1 / (a / b - 1)$ | $b(a - b) \neq 0 \wedge a \neq 1$ |

the second term, Gappa performs a second rewrite: $(a + b) - (A + B)$ is equal to $(a - A) + (b - B)$. This rewriting rule gives sensible results, as long as a and b are close to A and B respectively.

Table III contains some of the rules Gappa tries to apply automatically. There are two kinds of rewriting rules. Rules of the first kind, for example `add_firs`, are meant to produce simpler expressions. Rules of the second kind, for example `sub_xals`, are used to reproduce common practices of computer arithmetic by introducing intermediate terms in expressions. In order for an expression to match an uppercase letter in such a rule, the expression that matches the same letter in lowercase has to be tagged as an approximation of the former.

The first rule, `sub_xals`, has been applied earlier by Gappa, because Gappa

automatically tags $\text{rnd}(x)$ as an approximation of x , for any expression x , since rnd is a rounding operator. Gappa also creates such pairs for expressions that define absolute and relative errors in some hypotheses of a sub-formula of the proposition **PROP**. For example, on the following input, Gappa proposes accurate bounds, as it considers x to be an approximation of y , and $\lfloor x \rfloor$ of x .

```
@floor = int<dn>;
{ x - y in [-0.1,0.1] -> floor(x) - y in ? }
```

Thanks to its built-in rewriting rules and its heuristics to detect approximations, Gappa is able to automatically verify most properties on numerical applications that use common practices. Gappa, however, is not a complete decision procedure¹ and it may fail to prove some propositions. When that happens, users can give some hints to the tool.

For example, in the above input, a user could add the hint $x \sim y$ after the proposition, in order to indicate that x is an approximation of y . As Gappa was able to guess this property automatically, Gappa will warn that the hint is useless: the user can remove it.

Another kind of hint allows the users to directly add rewriting rules. The hint **primary** \rightarrow **secondary** states that Gappa can use an enclosure of **secondary** expression whenever it needs an enclosure of **primary** expression. For example, the following hint describes Newton's relation between the reciprocal $\frac{1}{y}$ and its approximation $x \cdot (2 - x \cdot y)$.

```
x * (2 - x * y) - 1/y -> (x - 1/y) * (x - 1/y) * -y
```

Such rules usually explicit some techniques applied by designers that are not necessarily visible in the source code. We cannot expect an automatic tool to re-discover innovative techniques. Yet, we will incorporate in Gappa any technique that we find to be commonly used. Any additional rewriting rule produces an hypothesis in the generated Coq file that must be proved independently, for example with the **ring** tactic of Coq.

In order for the **primary** \rightarrow **secondary** rule to be valid, any value of **primary** must be contained in the computed enclosure of **secondary**. This property generally holds true if both expressions are equal. As a consequence, Gappa tries to check if they are equal and warns if they are not, in order to detect mistypings early. Note that Gappa does not check if divisors are always different from zero before applying user-defined rewriting rules. Yet, Gappa detects divisors that are trivially equal to zero in expressions that appear in rewriting rules. For example, $y \rightarrow y * (x - x) / (x - x)$ is most certainly an error.

Due to built-in and user-defined rewriting rules, Gappa may hold more than one expression for a quantity, and hence several bounds as evaluations of equivalent expressions in interval arithmetic may yield different results. The intersection of the intervals yielded by the different expressions may be tighter than its previously known bounds. Tightening bounds on one quantity may then lead to tighter bounds

¹While seemingly simple, the formalism of Gappa is rich enough so that any first-order formula for Peano arithmetic can be expressed. As a consequence, it is impossible to design an algorithm that is able to automatically decide whether any proposition is provable or not.

on quantities based on it. Gappa explores the directed acyclic graph of quantities breadth-first until its goal is achieved or all bounds of the graph stopped evolving.

2.4 Sub-paving the range of some quantities by bisection (second use of HINT)

The last kind of hint that can be used when Gappa is unable to prove automatically a formula is to pave the range of some quantities and to prove independent results on each tile. Rewriting expressions is usually very efficient but it fails if different proof structures are needed on various parts of the range, as in the following example. The generic proof structure only works for $x \in [0, \frac{1}{2}]$. A specific proof structure is needed in order to extend the result to $x \in [\frac{1}{2}, 3]$. This proof relies on the fact that $\text{rnd}(y) - y$ is always zero there. But Gappa will not notice this property unless the last line is provided.

```
@rnd = float< ieee_32 , ne >;
x = rnd(x_);
y = x - 1;
z = x * (rnd(y) - y);
{ x in [0,3] -> |z| <= 1b-26 }
|z| $ x;
```

There are three constructions for bisection each involving a \$ sign in the hints section:

- Evenly split the range into as many sub-intervals as asked. E.g. \$ x in 6 splits the range of x in six sub-intervals. If the number of intervals is omitted (e.g. \$ x) and no expression is present on the left of \$, the default is 4.
- Split an interval on user-provided points. E.g. \$ x in (0.5,2) splits the range $[0, 3]$ of x above in three sub-intervals, the middle one being $[0.5, 2]$.
- The third kind of bisection tries to find by dichotomy a good sub-paving such that one goal of the proposition holds. The range of this goal has to be specified in the proposition, and the concerned expression has to be indicated on the left of the \$ symbol.

More than one bisection hint can be used and hints of the third kind can try to satisfy more than one goal at once. The two hints below will be used sequentially one after the other. The first one splits the range of u until all the enclosures on a , b , and c are verified.

```
a, b, c $ u;
d, e $ v;
```

Users may build higher dimension sub-paving by using more than one term on the right of the \$ symbol, reaching quickly combinatorial explosions though. The following statement asks Gappa to find a set of sub-ranges of u and w such that the goals on a and b are satisfied when the range of v is split into three sub-intervals.

```
a, b $ u, v in 3, w
```

3. HANDLING AUTOMATIC PROOF CHECKERS

3.1 Work on the logical proposition (PROP)

The proposition is first modified and loosely broken according to the rules of sequent calculus as presented below for the proposition seen in Section 2.1.

```
{ x - 2 in [-2,0] /\ (x + 1 in [0,2] -> y in [3,4])
  -> not x <= 1 \/ x + y in ? }
```

Each of the following new formulas is then verified by Gappa. If both formulas hold true, the original proposition does too.

$$\begin{aligned} x \leq 1 \wedge x - 2 \in [-2, 0] &\implies x + 1 \in [0, 2] \vee x + y \in ? \\ x \leq 1 \wedge x - 2 \in [-2, 0] \wedge y \in [3, 4] &\implies x + y \in ? \end{aligned}$$

Gappa performs this decomposition in order to obtain implication formulas with conjunctions of enclosures on their left hand sides and trees of conjunctions and disjunctions of enclosures on their right hand sides. In particular, all the negation symbols and the inner implications have been removed. For example, a set of implications of the form $e_1 \in I_1 \wedge \dots \wedge e_m \in I_m \Rightarrow f_1 \in J_1 \vee \dots \vee f_m \in J_m$ is suitable for a use by Gappa. Unspecified ranges (interrogation marks) are allowed as long as they appear only on the right hand sides of these decomposed formulas.

Inequalities may appear on both sides of the implications. Any inequality on the left hand side will be used only if Gappa can compute an enclosure of the expression by some other means. Any inequality on the right hand side is copied to the hypotheses as permitted by classical logic, provided that it is reverted first. For example, proposition $x \in [2, 3] \Rightarrow (y \in [4, 5] \wedge z \geq 6)$ is equivalent to proposition $(x \in [2, 3] \wedge z \leq 6) \Rightarrow (y \in [4, 5] \wedge z \geq 6)$, but the second one provides a bigger set of usable enclosures on its left hand side.

When the right hand side of the formula is a disjunction, Gappa searches for a sub-term that holds under the hypotheses of the proposition. It fails to prove valid disjunctions if it cannot find one sub-term that always holds under the hypotheses.

3.2 Structure of the generated proof

Enclosure (BND) is the only predicate available to users but Gappa internally relies on more predicates to describe properties on an expression x . Such predicates appear in intermediate lemmas of generated proofs.

$$\begin{aligned} \text{BND}(x, I) &\equiv x \in I \\ \text{ABS}(x, I) &\equiv |x| \in I \wedge I \geq 0 \\ \text{FIX}(x, e) &\equiv \exists m \in \mathbb{Z}, x = m \cdot 2^e \\ \text{FLT}(x, p) &\equiv \exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge |m| < 2^p \end{aligned}$$

The FIX and FLT predicates express that the set of computer numbers is generally a discrete subset of the real numbers, while intervals only consider connected subsets. They are especially useful for automatically detecting rounded operations that actually are exact operations, and hence do not contribute any rounding error.

Table IV lists most of the theorems used by Gappa. The verification process of these theorems relies on some interval operators defined in Table V. In particular, several operators $\text{err}_{\text{rnd}, k}$ related to rounding modes are needed. Some of these

Table IV. Theorems on interval arithmetic available from the Coq companion library to Gappa

| Target | Hypotheses | Constraint |
|--|--------------------------------------|---|
| $\text{BND}(\text{rnd}(a) - a, I)$ | | $I \supset \text{err}_{\text{rnd},0}$ |
| $\text{BND}(\text{rnd}(a) - a, I)$ | $\text{BND}(a, J)$ | $I \supset \text{err}_{\text{rnd},1}(J)$ |
| $\text{BND}(\text{rnd}(a) - a, I)$ | $\text{BND}(\text{rnd}(a), J)$ | $I \supset \text{err}_{\text{rnd},2}(J)$ |
| $\text{BND}(\text{rnd}(a) - a, I)$ | $\text{ABS}(a, J)$ | $I \supset \text{err}_{\text{rnd},3}(J)$ |
| $\text{BND}(\text{rnd}(a) - a, I)$ | $\text{ABS}(\text{rnd}(a), J)$ | $I \supset \text{err}_{\text{rnd},4}(J)$ |
| $\text{BND}((\text{rnd}(a) - a)/a, I)$ | $\text{BND}(a, J)$ | $I \supset \text{err}_{\text{rnd},5}(J)$ |
| $\text{BND}((\text{rnd}(a) - a)/a, I)$ | $\text{BND}(\text{rnd}(a), J)$ | $I \supset \text{err}_{\text{rnd},6}(J)$ |
| $\text{BND}((\text{rnd}(a) - a)/a, I)$ | $\text{ABS}(a, J)$ | $I \supset \text{err}_{\text{rnd},7}(J)$ |
| $\text{BND}((\text{rnd}(a) - a)/a, I)$ | $\text{ABS}(\text{rnd}(a), J)$ | $I \supset \text{err}_{\text{rnd},8}(J)$ |
| $\text{BND}(\text{rnd}(a), I)$ | $\text{BND}(a, J)$ | $I \supset \text{rnd}(J)$ |
| $\text{BND}(\text{rnd}(a), I)$ | $\text{BND}(\text{rnd}(a), J)$ | $I \supset J \cap \mathbb{F}_{\text{rnd}}$ |
| $\text{BND}(-a, I)$ | $\text{BND}(a, J)$ | $I \supset -J$ |
| $\text{BND}(a , I)$ | $\text{BND}(a, J)$ | $I \supset J $ |
| $\text{BND}(\sqrt{a}, I)$ | $\text{BND}(a, J)$ | $J \geq 0 \wedge I \supset \sqrt{J}$ |
| $\text{BND}(a - a, I)$ | | $0 \in I$ |
| $\text{BND}(a/a, I)$ | $\text{ABS}(a, J)$ | $1 \in I \wedge J > 0$ |
| $\text{BND}(a \times a, I)$ | $\text{BND}(a, J)$ | $I \supset J \times J $ |
| $\text{BND}(a + b, I)$ | $\text{BND}(a, J), \text{BND}(b, K)$ | $I \supset J + K$ |
| $\text{BND}(a - b, I)$ | $\text{BND}(a, J), \text{BND}(b, K)$ | $I \supset J - K$ |
| $\text{BND}(a \times b, I)$ | $\text{BND}(a, J), \text{BND}(b, K)$ | $I \supset JK$ |
| $\text{BND}(a/b, I)$ | $\text{BND}(a, J), \text{BND}(b, K)$ | $0 \notin K \wedge I \supset J/K$ |
| $\text{ABS}(-a, I)$ | $\text{ABS}(a, J)$ | $I \supset J$ |
| $\text{ABS}(a , I)$ | $\text{ABS}(a, J)$ | $I \supset J$ |
| $\text{ABS}(\sqrt{a}, I)$ | $\text{ABS}(a, J)$ | $I \supset \sqrt{J}$ |
| $\text{ABS}(a \pm b, I)$ | $\text{ABS}(a, J), \text{ABS}(b, K)$ | $I \supset J - K \cup (J + K)$ |
| $\text{ABS}(a \times b, I)$ | $\text{ABS}(a, J), \text{ABS}(b, K)$ | $I \supset J \times K$ |
| $\text{ABS}(a/b, I)$ | $\text{ABS}(a, J), \text{ABS}(b, K)$ | $K > 0 \wedge I \supset J/K$ |
| $\text{BND}(a, I)$ | $\text{ABS}(a, J)$ | $I \supset J \cup -J$ |
| $\text{BND}(a, I)$ | $\text{BND}(a, J), \text{ABS}(a, K)$ | $I \supset (J \cap K) \cup (J \cap -K)$ |
| $\text{BND}(a , I)$ | $\text{ABS}(a, J)$ | $I \supset J$ |
| $\text{ABS}(a, I)$ | $\text{BND}(a , J)$ | $I \supset J$ |
| $\text{BND}(a + b + a \times b, I)$ | $\text{BND}(a, J), \text{BND}(b, K)$ | $J \geq -1 \wedge K \geq -1 \wedge I \supset \mathcal{U}(J, K)$ |
| $\text{BND}(\xi, I)$ | | $I \supset \{\xi\}$ |
| $\text{FIX}(a \pm b, e)$ | $\text{FIX}(a, f), \text{FIX}(b, g)$ | $e \leq \min(f, g)$ |
| $\text{FIX}(a \times b, e)$ | $\text{FIX}(a, f), \text{FIX}(b, g)$ | $e \leq f + g$ |
| $\text{FLT}(a \times b, p)$ | $\text{FLT}(a, q), \text{FLT}(b, r)$ | $p \geq q + r$ |
| $\text{FIX}(a, e)$ | $\text{FLT}(a, q), \text{ABS}(a, J)$ | $J > 0 \wedge e \leq 1 + \log_2(\underline{J}) - q$ |
| $\text{FLT}(a, p)$ | $\text{FIX}(a, e), \text{ABS}(a, J)$ | $p \geq 1 + \log_2(\overline{J}) - e$ |
| $\text{FIX}(a, e)$ | $\text{BND}(a, [x, x])$ | $\exists m \in \mathbb{Z}, x = m \cdot 2^e$ |
| $\text{FLT}(a, p)$ | $\text{BND}(a, [x, x])$ | $\exists m, e \in \mathbb{Z}, x = m \cdot 2^e \wedge m < 2^p$ |
| $\text{FIX}(\text{rnd}(a), e)$ | | $e \leq e_{\text{rnd}}$ |
| $\text{FLT}(\text{rnd}(a), p)$ | | $p \geq p_{\text{rnd}}$ |
| $\text{BND}(\text{rnd}(a) - a, I)$ | $\text{FIX}(a, e), \text{FLT}(a, p)$ | $0 \in I \wedge e \geq e_{\text{rnd}} \wedge p \leq p_{\text{rnd}}$ |

Table V. Interval operators used in Table IV

| Operation | Constraint | Definition |
|--------------------------------|--------------|--|
| $-I$ | $0 \notin I$ | $[-\overline{I}, -\underline{I}]$ |
| I^{-1} | | $[1/\overline{I}, 1/\underline{I}]$ |
| $I + J$ | | $[\underline{I} + \underline{J}, \overline{I} + \overline{J}]$ |
| $I - J$ | | $I + (-J)$ |
| $I \times J$ | | $[\min(\underline{I}\underline{J}, \underline{I}\overline{J}, \overline{I}\underline{J}, \overline{I}\overline{J}), \max(\underline{I}\underline{J}, \underline{I}\overline{J}, \overline{I}\underline{J}, \overline{I}\overline{J})]$ |
| I/J | $0 \notin J$ | $I \times J^{-1}$ |
| \sqrt{I} | $I \geq 0$ | $[\sqrt{\underline{I}}, \sqrt{\overline{I}}]$ |
| $ I $ | | I if $I \geq 0$, $-I$ if $I \leq 0$, $[0, \max(-\underline{I}, \overline{I})]$ otherwise |
| $\mathcal{U}(I, J)$ | | $[\underline{I} + \underline{J} + \underline{I}\underline{J}, \overline{I} + \overline{J} + \overline{I}\overline{J}]$ |
| $\text{rnd}(I)$ | | One operator is associated to each rounding mode of Table I |
| $\text{err}_{\text{rnd},k}(I)$ | | Several operators are associated to each rounding mode of Table I |

operators may be left undefined; in that case, Gappa will generate longer proofs in order to use other operators instead. Some theorems also need to know the structure of the numbers that can be represented with respect to a given rounding mode: $\mathbb{F}_{\text{rnd}} = \{x \in \mathbb{R} \mid x = \text{rnd}(x)\} = \{m \cdot 2^e \mid e \geq e_{\text{rnd}} \wedge |m| < 2^{p_{\text{rnd}}}\}$.

The proof script generated for Coq contains the following kind of lemma whenever the certificate relies on interval addition to prove a proposition, e.g. “if $x \in [1, 2]$ (property p1) and $y \in [3, 4]$ (property p2), then $x + y \in [0, 6]$ (property p3)”.

```

1 Lemma l1 : p1 -> p2 -> p3.
2   intros h0 h1.
3   apply add with (1 := h0) (2 := h1) ; finalize.
4   Qed.

```

The first line defines the lemma: if the hypotheses p1 and p2 are verified, then the property p3 is true too. The second line starts the proof in a suitable state by using the `intros` tactic of Coq. The third line applies the `add` theorem of Gappa support library with the `apply` tactic.

The `add` theorem is as follows. `lower` and `upper` are functions that return the lower and the upper bound of an interval. Intervals are pairs of dyadic fractions (FF or IF). `Fplus2` is the addition of dyadic fractions. `Fle2` compares two dyadic fractions (less or equal) and returns a boolean. The `BND` predicate holds, when its first argument, an expression on real numbers, is an element of its second argument, an interval defined by dyadic fraction bounds.

```

Definition add_helper (xi yi zi : FF) :=
  Fle2 (lower zi) (Fplus2 (lower xi) (lower yi)) &&
  Fle2 (Fplus2 (upper xi) (upper yi)) (upper zi).

```

```

Theorem add :
  forall x y : R, forall xi yi zi : FF,
    BND x xi -> BND y yi ->
    add_helper xi yi zi = true ->
    BND (x + y) zi.

```

The mathematical expression of the theorem is as follows:

$$\begin{aligned} \text{add} : \forall x, y \in \mathbb{R}, \quad \forall I_x, I_y, I_z \in \mathbb{IF}, \\ x \in I_x \Rightarrow y \in I_y \Rightarrow \\ f_{\text{add}}(I_x, I_y, I_z) = \text{true} \Rightarrow \\ x + y \in I_z. \end{aligned}$$

If we simply needed a theorem describing the addition in interval arithmetic, the $f_{\text{add}}(I_x, I_y, I_z) = \text{true}$ hypothesis would be replaced by $I_x + I_y \subseteq I_z$. But we also need for the theorem hypotheses to be automatically checkable. It is the case for the $x \in I_x$ and $y \in I_y$ hypotheses of the **add** theorem, since they can be directly matched to the hypotheses **h0** ($x \in [1, 2]$) and **h1** ($y \in [3, 4]$) of lemma 11.

Hypothesis $I_x + I_y \subseteq I_z$, however, cannot be matched so easily. Consequently, it is replaced by an equivalent boolean expression that can be computed by a proof checker. In lemma 11, the computation is triggered by the **finalize** tactic that checks that the current goal can be reduced to $\text{true} = \text{true}$. This concludes the proof.

All the theorems of Gappa companion library are built the same way: instead of having standard hypotheses that Coq would be unable to automatically decide, they use a computable boolean expression. The companion library formally proves that, when this expression evaluates to *true*, the standard hypotheses hold true, and hence the goal of the theorem applies. This approach is a simpler form of reflection techniques [Boutin 1997]. Although the use of booleans seems to restrict the use of Gappa to Coq proof checker, the interval arithmetic library [Daumas et al. 2005] developed for PVS shows that proofs through interval computations are also attainable to other proof assistants.

3.3 Widening intervals to speedup proof certification

All the interval bounds are dyadic fractions ($m \cdot 2^n$ with m and n relative integers) in order to ensure that the boolean expressions are computable. Dyadic fractions are easily and efficiently added, multiplied, and compared. Rational numbers could also have been used: they would have been almost as efficient and would have provided a division operator. But common floating-point properties involved in certifying numerical codes are better described and verified by using dyadic fractions.

The proof checker does not need to compute any of these dyadic numbers, it just has to check that the interval bounds generated by Gappa make the boolean expressions evaluate to *true*, and hence are valid. In particular, there is absolutely no need for Gappa to compute the sharpest enclosing interval of an expression: any wider interval can be used. As long as the boolean expressions evaluate to *true*, the proof remains correct.

For example, manipulating the expression $x/\sqrt{3}$ will sooner or later require $\sqrt{3} \neq 0$ to be proved. This is done by computing an enclosing interval of $\sqrt{3}$ and verifying that its lower bound is positive. Hence there is no need to compute an enclosing interval with thousands of bits of precision, the interval $[1, 2]$ is accurate enough. Checking that $\sqrt{3} \in [1, 2]$ holds true is fast, as it just requires checking $1^2 \leq 3 \leq 2^2$. In order to get simplified dyadic numbers in intermediate lemmas, Gappa first finds a correct proof path and then it greedily operates backwards from the last proved results to the first proved results, widening the intervals along the way.

Such simplifications are important, since a proof checker like Coq is considerably slower than a specialized mathematical library. As a consequence, these simplified numbers can considerably speed up the verification process of propositions, especially when they involve error bounds. These considerations are also true for case studies: searching for a better sub-paving and certifying it, will always be faster than directly certifying the first sub-paving that has been found by Gappa. The time spent by Gappa in doing all the computations over and over in order to find a better sub-paving is negligible in comparison to the time necessary to certify the property on one single tile with a proof checker.

4. PERSPECTIVES AND CONCLUDING REMARKS

In our approach to program certification, generation of proof obligations, proof generation, and proof verification, are distinct steps. The intermediate step is performed by Gappa with its own computational methods, and the last one is done by a proof checker with the help of our support library.

The developments presented so far already allowed us to guarantee the correct behavior of many useful functions. As we continue using Gappa, we may discover practices that cannot be handled appropriately. We will extend Gappa, should this become necessary. Our software, a user's guide and details of some projects using Gappa are available on the Internet at the address below.

<http://lipforge.ens-lyon.fr/www/gappa/>

Gappa is used to certify CRLibm, a library of elementary functions with correct rounding in the four IEEE-754 rounding modes and performances comparable to standard mathematical libraries [de Dinechin et al. 2006; de Dinechin et al. 2004]. Figure 2 presents the input file needed to reproduce some parts of an earlier validation in HOL Light [Harrison 1997]. These expressions define an almost correctly rounded elementary function in single precision [Tang 1989]. Gappa is also used to develop robust semi-static filters for the CGAL project [Melquiond and Pion 2007] and in the validation of delayed linear algebra over finite fields [Daumas and Giorgi 2007].

The whole work of generating the proof is pushed toward the external program. All the intervals are precomputed and none of the complex tactics of Coq are used. The proof checker only has to be able to add, multiply, and compare integers; it does not have to be able to manipulate rational or real numbers. Consequently, one of our goal is to generate proofs not only for Coq, but for other proof checkers too.

Branches and loops handling are outside the scope of this work. Both problems are not new to program verification and nice results have been published in both areas. We do not want to propose our solution for these problems. Our decision is to interact with the two following tools.

—Why [Filliâtre 2003] is a tool to certify programs written in a generic language (C and Java can be converted to this language). It certifies appropriate memory allocation and usage. It is able to handle hierarchically structured code with functions and assertions. Why also takes care of conditional branches. It duplicates the appropriate proof obligations and guarantees that both pieces of code meet their shared post-conditions. A floating-point formalism designed with Gappa in

```

# 1. PROG: Definitions of aliases
@rnd = float< ieee_32, ne >;

# a few floating-point constants
a1 = 8388676b-24;
a2 = 11184876b-26;
l2 = 12566158b-48;
s1 = 8572288b-23;
s2 = 13833605b-44;

# the algorithm for computing the exponential
r2 rnd= -n * l2;
r rnd= r1 + r2;
q rnd= r * r * (a1 + r * a2);
p rnd= r1 + (r2 + q);
s rnd= s1 + s2;
e rnd= s1 + (s2 + s * p);

# a few mathematical expressions to simplify later sections
R = r1 + r2;
S = s1 + s2;

E = s1 + (s2 + S * (r1 + (r2 + R * R * (a1 + R * a2))));
Er = S * (1 + R + a1 * R * R + a2 * R * R * R + 0);
E0 = S0 * (1 + R0 + a1 * R0 * R0 + a2 * R0 * R0 * R0 + Z);

# 2. PROP: Logical proposition Gappa has to verify
{ # provide the domains and accuracies of some variables
  Z in [-55b-39,55b-39] /\ S - S0 in [-1b-41,1b-41] /\
  R - R0 in [-1b-34,1b-34] /\ R in [0,0.0217] /\ n in [-10176,10176] ->
  # ask for the range of e and its absolute error
  e in ? /\ e - E0 in ? }

# 3. HINTS: Hints provided by the user
e - E0 -> (e - E) + (Er - E0); # true as E = Er
r1 -> R - r2;                  # true as R = r1 + r2
    
```

Fig. 2. Gappa script for proving e accurately approximates $E_0 = \exp(R_0)$ in single-precision.

mind has recently been added to Why [Boldo and Filliâtre 2007]. Used together, Why and Gappa will be able to handle large pieces of software.

- Fluctuat [Putot et al. 2004] handles loops by effectively computing loop invariants. Once these invariants are provided, Gappa can certify the correct behavior of any numerical code. Results of Fluctuat will be used as oracles and certified by Gappa. Should there be a significant bug in Fluctuat, Gappa will stop without being able to meet its goals as it cannot certify erroneous results.

REFERENCES

- BOLDO, S. AND DAUMAS, M. 2004. A simple test qualifying the accuracy of Horner's rule for polynomials. *Numerical Algorithms* 37, 1-4, 45-60.
- BOLDO, S. AND FILLIÂTRE, J.-C. 2007. Formal verification of floating point programs. In *Proceedings of the 18th Symposium on Computer Arithmetic*. Montpellier, France.

- BOLDO, S. AND MELQUIOND, G. 2005. When double rounding is odd. In *Proceedings of the 15th IMACS World Congress on Computational and Applied Mathematics*. Paris, France.
- BOUTIN, S. 1997. Using reflection to build efficient and certified decision procedures. In *Proceedings of the Third International Symposium on Theoretical Aspects of Computer Software*. London, United Kingdom, 515–529.
- BRÖNNIMANN, H., MELQUIOND, G., AND PION, S. 2003. The Boost interval arithmetic library. In *Real Numbers and Computers*. Lyon, France, 65–80.
- DAUMAS, M. AND GIORGI, P. 2007. Proof checking for delayed finite field arithmetic using floating point operators. Tech. Rep. hal-00135090, Centre pour la Communication Scientifique Directe, Villeurbanne, France.
- DAUMAS, M. AND MELQUIOND, G. 2004. Generating formally certified bounds on values and round-off errors. In *Real Numbers and Computers*. Dagstuhl, Germany, 55–70.
- DAUMAS, M., MELQUIOND, G., AND MUÑOZ, C. 2005. Guaranteed proofs using interval arithmetic. In *Proceedings of the 17th Symposium on Computer Arithmetic*, P. Montuschi and E. Schwarz, Eds. Cape Cod, Massachusetts, 188–195.
- DE DINECHIN, F., DEFOUR, D., AND LAUTER, C. 2004. Fast correct rounding of elementary functions in double precision using double-extended arithmetic. Research report 5137, Institut National de Recherche en Informatique et en Automatique, Le Chesnay, France.
- DE DINECHIN, F., LAUTER, C. Q., AND MELQUIOND, G. 2006. Assisted verification of elementary functions using Gappa. In *Proceedings of the 2006 ACM Symposium on Applied Computing*. Dijon, France, 1318–1322.
- EVEN, G. AND SEIDEL, P.-M. 1999. A comparison of three rounding algorithms for IEEE floating-point multiplication. In *Proceedings of the 14th Symposium on Computer Arithmetic*, I. Koren and P. Kornerup, Eds. Adelaide, Australia, 225–232.
- FILLIÁTRE, J.-C. 2003. Why: a multi-language multi-prover verification tool. Research Report 1366, Université Paris Sud.
- FOUSSE, L., HANROT, G., LEFÈVRE, V., PÉLISSIER, P., AND ZIMMERMANN, P. 2005. MPFR: A multiple-precision binary floating-point library with correct rounding. Tech. Rep. RR-5753, INRIA.
- GAMEIRO, M. AND MANOLIOS, P. 2004. Formally verifying an algorithm based on interval arithmetic for checking transversality. In *Fifth International Workshop on the ACL2 Theorem Prover and Its Applications*. Austin, Texas, 17.
- HARRISON, J. 1997. Floating point verification in HOL light: the exponential function. Technical Report 428, University of Cambridge Computer Laboratory.
- HARRISON, J. 2000. *The HOL Light manual*. Version 1.1.
- HIGHAM, N. J. 2002. *Accuracy and stability of numerical algorithms*. SIAM. Second edition.
- HUET, G., KAHN, G., AND PAULIN-MÖHRING, C. 2004. *The Coq proof assistant: a tutorial: version 8.0*.
- JAULIN, L., KIEFFER, M., DIDRIT, O., AND WALTER, E. 2001. *Applied interval analysis*. Springer.
- KAHAN, W. 1965. Further remarks on reducing truncation errors. *Communications of the ACM* 8, 1, 40.
- KAUFMANN, M., MANOLIOS, P., AND MOORE, J. S. 2000. *Computer-Aided Reasoning: An Approach*. Kluwer Academic Publishers.
- MELQUIOND, G. AND PION, S. 2007. Formally certified floating-point filters for homogeneous geometric predicates. *Theoretical Informatics and Applications*. To appear.
- MICHARD, R., TISSERAND, A., AND VEYRAT-CHARVILLON, N. 2006. Optimisation d'opérateurs arithmétiques matériels à base d'approximations polynomiales. In *Symposium en Architecture de Machines*. Perpignan, France, 1318–1322.
- NEUMAIER, A. 1990. *Interval methods for systems of equations*. Cambridge University Press.
- OWRE, S., RUSHBY, J. M., AND SHANKAR, N. 1992. PVS: a prototype verification system. In *11th International Conference on Automated Deduction*, D. Kapur, Ed. Springer-Verlag, Saratoga, New-York, 748–752.

- PUTOT, S., GOUBAULT, E., AND MARTEL, M. 2004. Static analysis based validation of floating point computations. In *Novel Approaches to Verification*. Lecture Notes in Computer Science, vol. 2991. Dagstuhl, Germany, 306–313.
- REVV, G. 2006. Analyse et implantation d'algorithmes rapides pour l'évaluation polynomiale sur les nombres flottants. Tech. Rep. ensl-00119498, École Normale Supérieure de Lyon.
- ROCKWELL COLLINS. 2005. Rockwell Collins receives MILS certification from NSA on microprocessor. Press Releases.
- RUMP, S. M., OGITA, T., AND OISHI, S. 2005. Accurate floating-point summation. Tech. Rep. 05.12, Hamburg University of Technology, Hamburg, Germany.
- SCHLUMBERGER. 2003. Schlumberger leads the way in smart card security with common criteria EAL7 security methodology. Press Releases.
- STEVENSON, D. ET AL. 1987. An American national standard: IEEE standard for binary floating point arithmetic. *ACM SIGPLAN Notices* 22, 2, 9–25.
- TANG, P. T. P. 1989. Table driven implementation of the exponential function in IEEE floating point arithmetic. *ACM Transactions on Mathematical Software* 15, 2, 144–157.
- Texas Instruments 1997. *TMS320C3x — User's guide*. Texas Instruments.
- TIWARI, A., SHANKAR, N., AND RUSHBY, J. 2003. Invisible formal methods for embedded control systems. *Proceedings of the IEEE* 91, 1, 29–39.