Static Analysis-based Validation of Floating-Point Computations

Sylvie Putot,
joint work with Eric Goubault and Matthieu Martel

CEA Saclay
Software Safety Laboratory
Sylvie.Putot@cea.fr

Dagstuhl Seminar “Numerical Software with Result Verification” 19-24.01.03

Introduction

- Validation of critical embedded systems
  - Programs of large size, up to 100000 lines or more
  - Numerically simple
- Necessity of having well-suited tools to verify the programs
  - No instrumentation of the source code
  - Guaranteed bounds on errors for classes of executions
  - Identify the operations responsible for the main losses of precision
- The aim is not to compute an estimation of the real result for given inputs.

Wish to go towards verification of more numerical programs

Static Analysis

Program analysis : proof that a software computes what it is intended to. Based on

- a semantics of the program
- a definition of the properties to compute (e.g. value of variables on the nodes of a control flow graph of the program)

Program $\equiv$ functional f. Static analysis : computation of a fixpoint by monotone iterations for all possible inputs

$$\mathrm{lfp} f = \bigcup_{i \in \mathbb{N}} f^i(\bot)$$

Functional formulation $P = f(P) \longrightarrow$ use of classical methods for such equations : iteration strategies, convergence acceleration, etc.

Example (functional formulation)

Consider

```
1: while (x >= 1000)
   {
2:     x = x + a;
3: } 4:
```

Translated into equations (relations between predicates) :

$$\begin{aligned} P_1 &= (x = \overline{x}) \\ P_2 &= (1000 \leq x) \wedge (P_1 \vee P_3) \\ P_3 &= P_2(x - a) \\ P_4 &= (x < 1000) \wedge (P_1 \vee P_3) \end{aligned}$$

What is a correct program, when there are floating-point computations ?

Could be for example the proof that a program corresponds to a given function or algorithm, or a stability proof

Our approach :

- Specification = real number semantics (what is expected)
- Implementation = IEEE 754 floating-point number semantics

We want to :

- Prove that the result of the computation in finite precision does not grow too far away from the result got with real numbers.
- Give an idea of how control points contribute to the imprecision

Link between idealized and f.p. computation

Chosen approach : abstract interpretation, relies on a difference semantics :

$$r = f + \sum_{l \in \mathcal{L}} \omega^l \vec{\varepsilon}_l + \omega^{ho} \vec{\varepsilon}_{ho}$$

- $f \in \mathbb{F}$ is the f.p. number used by the computer instead of $r \in \mathbb{R}$
- ω^l is the contribution of point (or set of points ; C lines in our implementation) l to the first order error on f
- errors of order more than 1 are agglomerated in ω^{ho} (just a check : often negligible)
- $\vec{\varepsilon}_l$ is a label corresponding to point l (not like A.A. !)

Abstract semantics : f and ω are approximated using intervals

Arithmetic Operations

$$r_1 +^{\ell_i} r_2 \overset{\text{def}}{=} \uparrow_\circ (f_1 + f_2)\vec{\varepsilon} + \sum_{u \in \overline{\mathcal{L}}} (\omega_1^u + \omega_2^u)\vec{\varepsilon}_u + \downarrow_\circ (f_1 + f_2)\vec{\varepsilon}_{\ell_i}$$

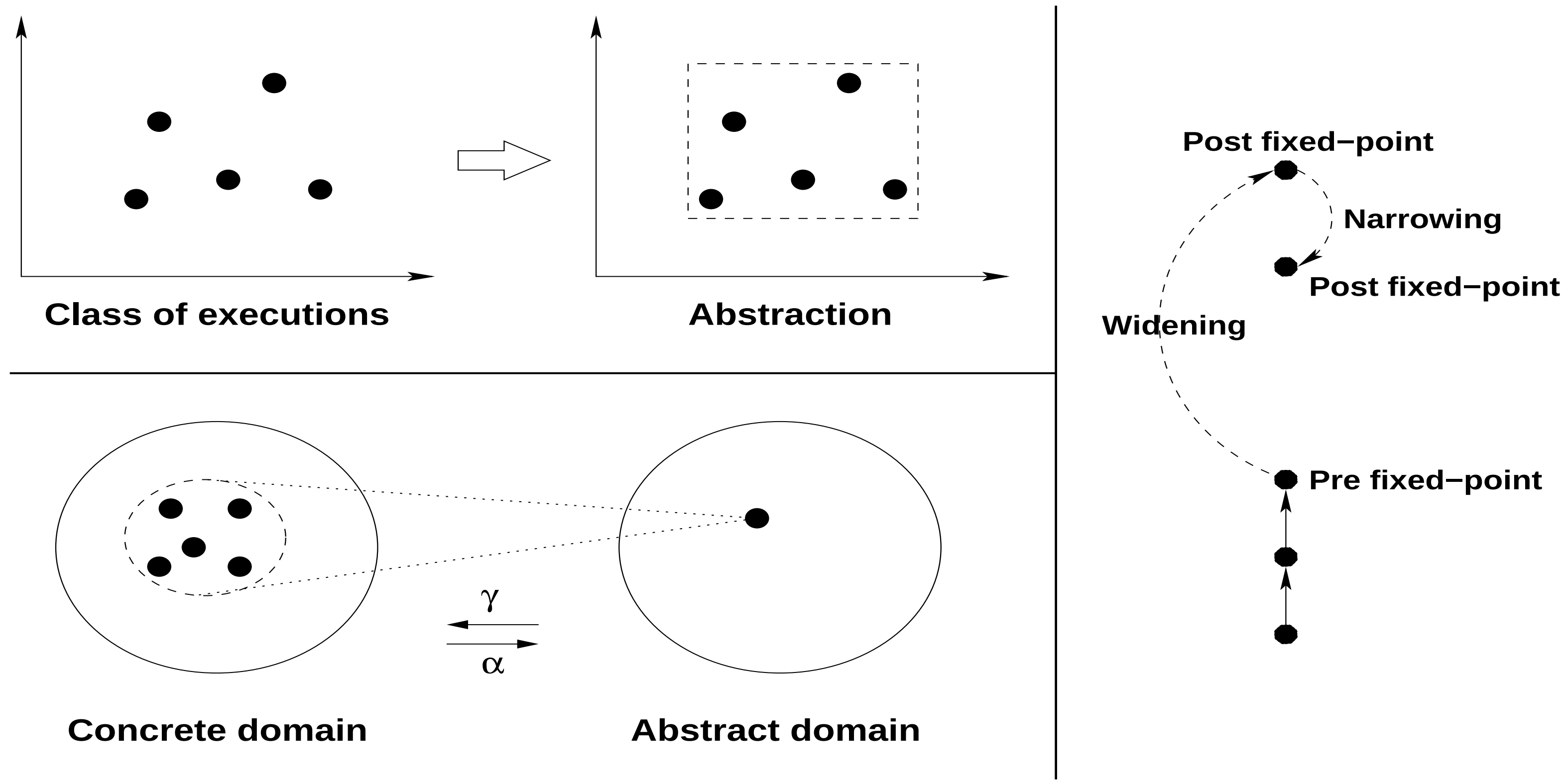
$$r_1 \times^{\ell_i} r_2 \overset{\text{def}}{=} \uparrow_\circ (f_1 f_2)\vec{\varepsilon} + \sum_{u \in \overline{\mathcal{L}}} (f_1 \omega_2^u + f_2 \omega_1^u)\vec{\varepsilon}_u + \sum_{u \in \overline{\mathcal{L}}, v \in \overline{\mathcal{L}}} \omega_1^u \omega_2^v \vec{\varepsilon}_{ho} + \downarrow_\circ (f_1 f_2)\vec{\varepsilon}_{\ell_i}$$

with $\overline{\mathcal{L}} = \mathcal{L} \cup \{ho\}$.

Arithmetic Operations : an Example

$$
\begin{array}{rrcrl}
 & 621.3\vec{\varepsilon} & + & 0.05\vec{\varepsilon}_{\ell_1} & r_1^{\ell_1} \\
\times^{\ell_3} & 1.287\vec{\varepsilon} & + & 0.0005\vec{\varepsilon}_{\ell_2} & r_2^{\ell_2} \\
\cline{1-4}
= & 799.6131\vec{\varepsilon} & & & \text{Result} \\
 & & + & 0.06435\vec{\varepsilon}_{\ell_1} & \text{Error due to } r_1^{\ell_1} \\
 & & + & 0.31065\vec{\varepsilon}_{\ell_2} & \text{Error due to } r_2^{\ell_2} \\
 & & + & 0.000025\vec{\varepsilon}_{\ell_1}\vec{\varepsilon}_{\ell_2} & \text{Second order error term} \\
\cline{1-4}
= & 799.6\vec{\varepsilon} & & & \text{Machine result } =\uparrow_\circ (r_1 \times r_2) \\
 & & + & 0.06435\vec{\varepsilon}_{\ell_1} & \text{Error due to } r_1^{\ell_1} \\
 & & + & 0.31065\vec{\varepsilon}_{\ell_2} & \text{Error due to } r_2^{\ell_2} \\
 & & + & 0.000025\vec{\varepsilon}_{ho} & \text{Second order error term} \\
 & & + & 0.0131\vec{\varepsilon}_{\ell_3} & \text{Error introduced by } \times^{\ell_3} =\downarrow_\circ (r_1 \times r_2)
\end{array}
$$

Abstract Interpretation : principle



The abstract computation gives an upper approximation of the concrete least fixpoint.

Abstract interpretation : example

Sets of integers $\overset{\gamma}{\underset{\alpha}{\leftrightarrows}}$ Intervals of integers

```
x=0;
while (x<100)
  x=x+1;
```

- Iteration 1 : $x_1 = [1, 1]$
- Iteration 2 : $\left| \begin{array}{l} x = [2, 2] \\ x_2 = x_1 \cup x = [1, 2] \end{array} \right.$
- Iteration 3 : $x_3 = [1, 3]$
- Widening : $x = [1, +\infty]$
- Narrowing : $x = [1, +\infty] \cap [-\infty, 100] = [1, 100]$

Abstract domain for floating-point numbers

$$r = f + \sum_{l \in \mathcal{L}} \omega^l \vec{\varepsilon}_l + \omega^{ho} \vec{\varepsilon}_{ho}$$

- For the float f : interval of float/double
  - computed as on target machine (with rounding to the nearest)
  - if input values are reduced to values, f is the floating-point result got by execution on target machine
- For the errors ω : intervals of higher-precision floating-point numbers (using the multi-precision library MPFR)
  - using the fact that the norm specifies that $+$, $-$, $*$, $/$, $\sqrt{}$ are computed with max imprecision of $ulp/2$ of the exact result
  - if result is a value (not interval), computation of tight bounds

Formulation close to affine interval arithmetic

- This formulation does not take advantage of the correlations between variables to reduce errors.
- Relational analysis using linear error dependencies :

$$r \in \mathbb{R} \quad \longrightarrow \quad x = f^x + \sum_{l \in \mathcal{L}} t_l^x * \gamma_l \, \vec{\varepsilon}_l + \omega_{ho}^x \, \vec{\varepsilon}_{ho}$$

  - γ_l is an abstract value that represents the error committed at point $\vec{\varepsilon}_l$ for one execution. We know a range, $\gamma_l \in [\alpha_l, \beta_l]$
  - t_l^x expresses the dependency of the variable x to the error γ_l
  - Difficulties with loops, and to group errors on sets of points
- One could also use affine interval arithmetic to have closer approximations of values f^x

Relational analysis : example

```
x =__BUILTIN_DAED_FBETWEEN(0,1.0);
y = x + 1;
z = 10 y;
t = z - 5y;
```

Non relational :

$y = [1,2] + [-2,2]ulp(1)\epsilon_2$

$z = [10,20] + [-20,20]ulp(1)\epsilon_2 + [-20,20]ulp(1)\epsilon_3$

$t = [0,15] + [-30,30]ulp(1)\epsilon_2 + [-20,20]ulp(1)\epsilon_3 + [-15,15]ulp(1)\epsilon_4$

Relational (partly) :

$$\begin{array}{ll} y = [1,2] + [1,1]\gamma_2\epsilon_2, & \gamma_2 \in [-2,2]ulp(1) \\ z = [10,20] + [10,10]\gamma_2\epsilon_2 + [1,1]\gamma_3\epsilon_3, & \gamma_3 \in [-20,20]ulp(1) \\ t = [0,15] + [5,5]\gamma_2\epsilon_2 + [1,1]\gamma_3\epsilon_3 + [1,1]\gamma_4\epsilon_4, & \gamma_4 \in [-15,15]ulp(1) \end{array}$$

Iteration strategies to compute an approx of the fixpoint

```
int i; float t=1;
for (i=0; i<20; i++)
  t=t*.618
```

Unfolding twice the loop : analysis of

```
for (i=0; i<10; i++)
  { t=t*.618;
    if (i>=10) break;
    t=t*.618;  }
```

$\Longrightarrow$ if we unfold the loop N times, we compute an upper approx to $\lim_n(X_1^n, X_2^n, \dots, X_N^n)$ with X_i^n such that

$$\begin{cases} X_i^0 = \bot, \ \ i = 1 \dots N, \\ X_i^n = X_i^{n-1} \cup f^{(i)}(X_N^{n-1}), \ \ i = 1 \dots N, \ n \geq 1. \end{cases}$$

Then $X = \bigcup_{i=1}^{N} X_i$.

Need for a virtual unfolding of loops

```
t = 1;
for (i=1 ; i<=20 ; i++)
  t = t*0.618;          (epsilon)
```

- Result without unfolding : $t \in [0, 0.618]$, error $\in]-\infty, +\infty[$

$i = 1 : \quad t_1 = [0.618, 0.618] + 0.618\, ulp(1)\, [-1, 1]\epsilon$

$$i = 2 : \begin{cases} t_2 = [0.618^2, 0.618^2] + 2 * 0.618^2\, ulp(1)[-1, 1]\, \epsilon \\ t_2 := t_1 \cup t_2 = [0.618^2, 0.618] + 2 * 0.618^2\, ulp(1)[-1, 1]\, \epsilon \quad (2 * 0.618^2 > 0.618) \end{cases}$$

$i = 3 : \quad t_3 = [0.618^3, 0.618] + (2 * 0.618^3 + 0.618^2)\, ulp(1)[-1, 1]\, \epsilon$

Because of union on values, the computed error increases while the error really committed decreases

(could be solved with a good widening : limit of error $\frac{1}{1-.618} ulp(1)$)

A possible solution : unfold twice the loop

$$i = 1 : \quad \begin{cases} t_1 = [0.618, 0.618] + 0.618\, ulp(1)\, [-1,1]\epsilon_1 \\ t_2 = [0.618^2, 0.618^2] + 0.618^2\, ulp(1)\, [-1,1]\epsilon_1 + 0.618^2\, ulp(1)\, [-1,1]\epsilon_2 \end{cases}$$

$i = 2$:

$$\begin{cases} t_3 = t_2 * 0.618 = [0.618^3, 0.618^3] + 2 * 0.618^3\, ulp(1)\, [-1,1]\epsilon_1 + 0.618^3\, ulp(1)\, [-1,1]\epsilon_2 \\ t_4 = t_3 * 0.618 = [0.618^4, 0.618^4] + 2 * 0.618^4\, ulp(1) [-1,1]\epsilon_1 + 2 * 0.618^4\, ulp(1)\, [-1,1]\epsilon_2 \end{cases}$$

$$\begin{cases} t_3 := t_3 \cup t_1 = [0.618^3, 0.618] + 0.618\, ulp(1)\, \epsilon_1 + 0.618^3\, ulp(1)\, \epsilon_2 \\ t_4 = t_4 \cup t_2 = [0.618^4, 0.618^2] + 0.618^2\, ulp(1)\, \epsilon_1 + 0.618^2\, ulp(1)\, \epsilon_2 \end{cases}$$

$i = 3$:

$$\begin{cases} t_5 = t_4 * 0.618 = [0.618^5, 0.618^3] + (0.618^2 + 0.618^3)\, ulp(1)\, [-1,1]\epsilon_1 + 0.618^4\, ulp(1)\, [-1,1]\epsilon_2 \\ t_6 = t_5 * 0.618 = [0.618^6, 0.618^4] + (0.618^3 + 0.618^4)\, ulp(1)\, [-1,1]\epsilon_1 + (0.618^5 + 0.618^4)\, ulp(1)\, [-1 \end{cases}$$

$$\begin{cases} t_5 := t_5 \cup t_3 = [0.618^3, 0.618] + 0.618\, ulp(1)\, [-1,1]\epsilon_1 + 0.618^3\, ulp(1)\, [-1,1]\epsilon_2 \\ t_4 = t_5 \cup t_3 = [0.618^4, 0.618^2] + 0.618^2\, ulp(1)\, [-1,1]\epsilon_1 + 0.618^2\, ulp(1)\, [-1,1]\epsilon_2 \end{cases}$$

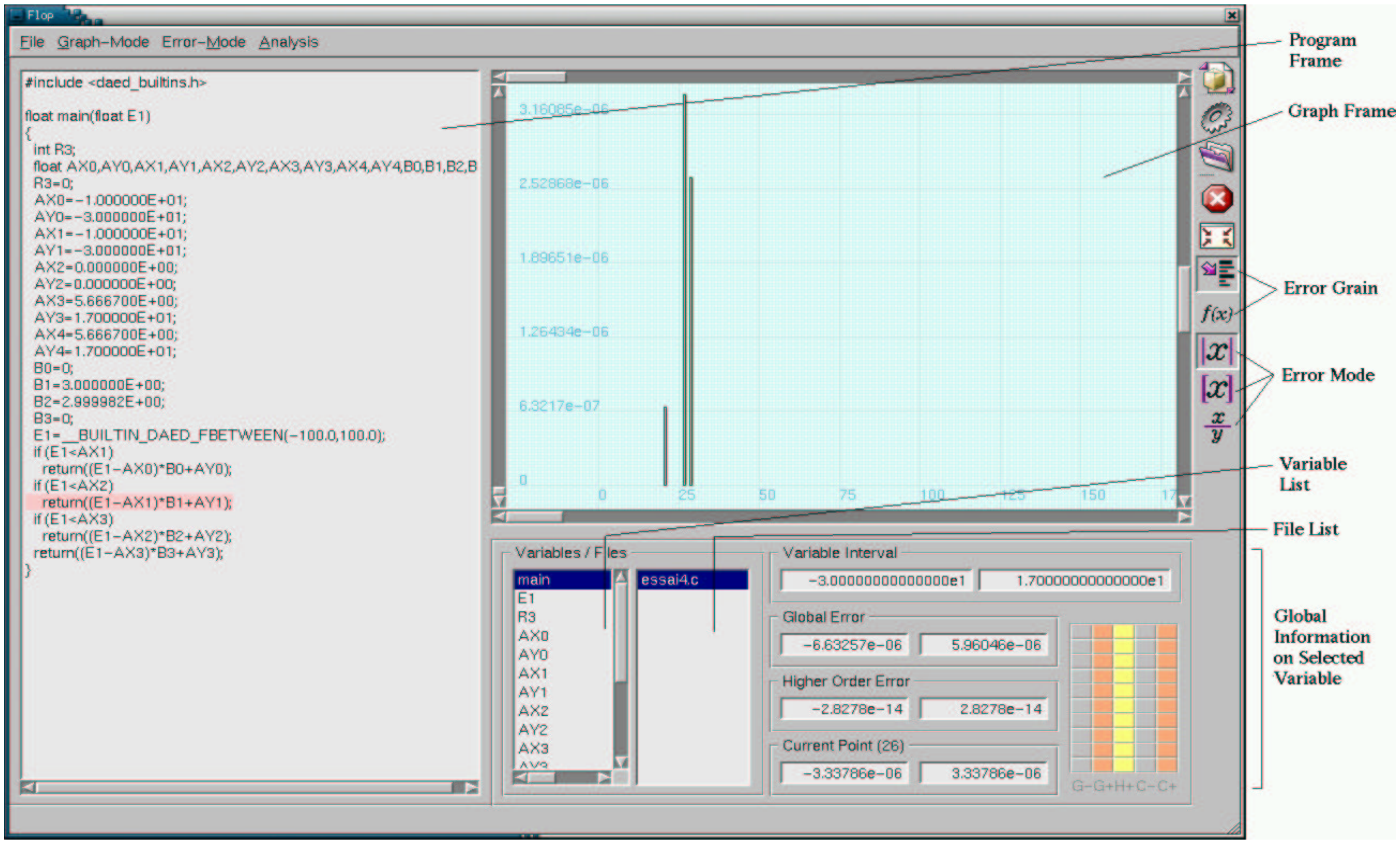
Convergence for the errors

The Fluctuat tool

- We have a first prototype for not too numerically intensive programs (instrumentation and control).
  - Interprocedural analysis of a fragment of ANSI C; few library functions, current work on compound data structure - alias, struct, arrays.
- *Short demo* :

  [We replace it here, for "paper version", by a photo of the main window of the graphic interface, and computed bounds for values and errors for the examples shown during the presentation.]

Graphic interface



```
#include <daed_builtins.h>

float main(float E1)
{
 int R3;
 float AX0,AY0,AX1,AY1,AX2,AY2,AX3,AY3,AX4,AY4,B0,B1,B2,B
 R3=0;
 AX0=-1.000000E+01;
 AY0=-3.000000E+01;
 AX1=-1.000000E+01;
 AY1=-3.000000E+01;
 AX2=0.000000E+00;
 AY2=0.000000E+00;
 AX3=5.666700E+00;
 AY3=1.700000E+01;
 AX4=5.666700E+00;
 AY4=1.700000E+01;
 B0=0;
 B1=3.000000E+00;
 B2=2.999982E+00;
 B3=0;
 E1=__BUILTIN_DAED_FBETWEEN(-100.0,100.0);
 if (E1<AX1)
  return((E1-AX0)*B0+AY0);
 if (E1<AX2)
  return((E1-AX1)*B1+AY1);
 if (E1<AX3)
  return((E1-AX2)*B2+AY2);
 return((E1-AX3)*B3+AY3);
}
```

Variables / Files

main	essai4.c
E1	
R3	
AX0	
AY0	
AX1	
AY1	
AX2	
AY2	
AX3	

Variable Interval: -3.00000000000000e1 | 1.70000000000000e1

Global Error: -6.63257e-06 | 5.96046e-06

Higher Order Error: -2.8278e-14 | 2.8278e-14

Current Point (26): -3.33786e-06 | 3.33786e-06

Example 1 (simple interpolation)

```
#include <daed_builtins.h>
float main(float E1)
{
  float R1_X3,R2_0,R2_1,R2_2;
  R2_0=0; R2_1=3;
  R2_2=2.999982;
  E1=__BUILTIN_DAED_FBETWEEN(-100.0,100.0);    // assertion meaning that -100 <= E1 <= 100
  if (E1 < -10)
    return((E1+10.0)*R2_0-20.0);
  if (E1 < 1.2)
    return((E1+51.1)*R2_1-40.0);
  return(E1*R2_2);
}
```

- Result = bounds for floating point value + errors at the end of the program, for all variables
- We consider here the return value (called by the name of the function, i.e. 'main') :

```
main = type : simple float
  Floating-point value : [-2.000000e1,2.999982e2]
      Error line  6 : [-1.18606567234280646e-5,0]
      Error line 11 : [-3.19744231092045084e-14,3.19744231092045084e-14]
      Error line 12 : [-1.52587890625000000e-5,1.52587890625000000e-5]
  Real value : [-2.00000271194458179e1,2.99998214721679719e2]
```

Example 2

Exact limit of this famous sequence is 6, but limit with rounding errors is 100.

```
int main(void)
{
  float x0, x1, x2;
  int i;
  x0 = 11/2.0;
  x1 = 61/11.0;
  for (i=1 ; i<=13 ; i++)  {
    x2 = 111 - (1130 - 3000/x0) / x1;
    x0 = x1;
    x1 = x2; }
}
```

Results of the analysis with default precision, and complete unfolding of the loop :

```
x2 = type : simple float
 Floating-point value : [1.000000e2,1.000000e2]    // = result got by execution
        Error line 6 : [1.51444913011716410e-7,1.51444921303193455e-7]
        Error line 8 : [2.85099357418704060e-6,2.85099359135853044e-6]
  Higher order error : [-9.50041512715925676e1,-9.31771829974241715e1]
 Real value  : [4.99585173084591960,6.82282000501434116]
```

The floating-point value tends towards 100, and there is a large global error, between -95.1 and -93.2, that comes mainly from errors of order larger that one. The estimation of the error is not very tight, however it is assured that the magnitude of the error is larger than 92, we guess that the example is unstable.

Analysis with 80 bits of precision :

```
x2 = type : simple float
 Floating-point value : [1.000000e2,1.000000e2]    // = result got by execution
        Error line 6 : [1.51444917165164321972338e-7,1.51444917165172229341030e-7]
        Error line 8 : [2.85099358276164098879841e-6,2.85099358276165736480817e-6]
  Higher order error : [-9.40722624813775285718735e1,-9.40722607395473614241816e1]
 Real value  : [5.92774052106097135493179,5.92774226289113850264803]
```

The estimation of the error is now tight. And the more iterations of the sequence we want to compute, the more precision in analysis we will need for a tight estimation of errors and real value.

Example 3 (loop)

```
#include <daed_builtins.h>
main(int n) {
  float x,y,z,t;
  int i;
  x=1.0; t = 0;
  y = __BUILTIN_DAED_FBETWEEN(-2.0,3.0);
  y = y-1.0/3.0;
  for (i=1;i<=n;i++) {                    // Note that n can take any integer value
    z=x; x=y;
    y=(x+z)/6.0;
    t=(2*t+y)/2.0; }
}
```

The value of x, y and z, and the errors committed on these variables, decrease during the iterations. The value of t, and the error committed, increase.

Result of analysis (for all value of n) without unfolding the loop :

```
 x = type : simple float
Floating-point value : [-2.333333,2.666667]
   Error line  7 : [-1.19209289791474132e-7,1.19209289754611258e-7]
   Error line 10 : [-oo,+oo]

 t = type : simple float
Floating-point value : [-oo,+oo]
   Error line  7 : [-oo,+oo]
   Error line 10 : [-oo,+oo]
   Error line 11 : [-oo,+oo]
```

Note that we get a bounded value for x (or y or z), and a bounded error coming from line 7 : even if we did not get bounds for line 10 error, this is a stable scheme for y. Indeed, the error committed outside the loop is not amplificated inside the loop. Whereas it is amplificated for t.

Result of analysis with 3 unfoldings of the loop :

```
x = type : simple float
Floating-point value : [-2.333333,2.666667]
      Error line  7 : [-1.19209289791474132e-7,1.19209289754611258e-7]
      Error line 10 : [-6.75874097750114471e-8,6.75874097750114471e-8]
```

Values and errors for t are still unbounded. But now we get bounds for all errors on x (or y and z) .

Some problems

- Unstable tests :
  What to do when, because of a test, the real values can follow a different path than the floating-point values !?
  - Follow only the floating-point value path, and signal any unstable test (done now)
  - Follow the different paths, and take results of real path as reference to compute extra errors (correct, but often pessimistic)
- Evaluation order in expressions with several arithmetic operations :
  We do not execute the code but we analyze a graph of the program, thus we do not know the optimizations that the compiler will do (a solution is to analyze the assembly code ...)

References

- “Static analyses of the precision of floating-point operations”, Eric Goubault, SAS’01, Lecture Notes in Computer Science # 2126

- “Concrete and Abstract Semantics of Floating-Point Operations”, Eric Goubault, Matthieu Martel, Sylvie Putot, Research Report DRT/LIST/DTSI/SLA/LSL/01-058, 2001

- “Asserting the precision of floating-point computations: a simple abstract interpreter”, Eric Goubault, Matthieu Martel, Sylvie Putot, ESOP’02, Lecture Notes in Computer Science

- “Static Analysis of the Numerical Stability of Loops”, Matthieu Martel, SAS’02

- “Fluctuat : a Static Analyzer to assert the precision of floating-point computations. User Manual”, Eric Goubault, Matthieu Martel, Sylvie Putot, Research Report DTSI/SLA/02-497/EG, 2002

http://www-dta.cea.fr/Pages/List/lse/LSL/Flop/index.html
http://www.di.ens.fr/ cousot/projects/DAEDALUS/synthetic_summary/CEA/Fluctuat/