Second NASA Formal Methods Symposium
Washington D.C. 2010

Hardware-independent proofs of numerical programs

Sylvie Boldo and **Thi Minh Tuyen Nguyen**

INRIA Saclay, France

April 14th, 2010

INSTITUT NATIONAL DE RECHERCHE EN INFORMATIQUE ET EN AUTOMATIQUE

INRIA

centre de recherche SACLAY - ÎLE-DE-FRANCE

A first example

```
void sign(double x){
  if       (x > 0.0)  printf("Positive");
  else if  (x < 0.0)  printf("Negative");
  else                printf("Zero");
}
void main(){
  double a = 0x1p-53 + 0x1p-64;    // a = 2^-53 + 2^-64
  double b = 1.0 + a - 1.0;
  sign(b - a);
}
```

A first example

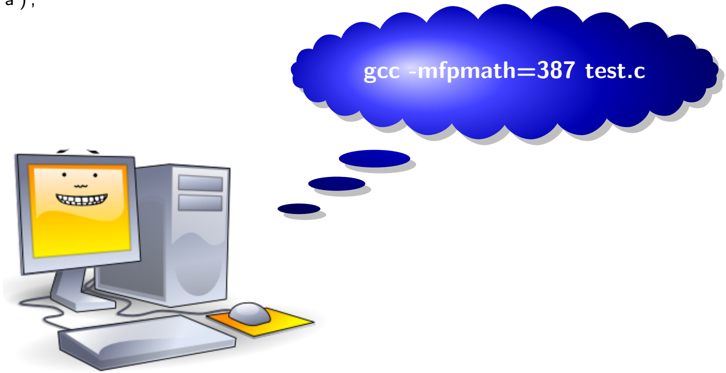
```
void sign(double x){
  if        (x > 0.0)  printf("Positive");
  else if   (x < 0.0)  printf("Negative");
  else                 printf("Zero");
}
void main(){
  double a = 0x1p-53 + 0x1p-64;    // a = 2^-53 + 2^-64
  double b = 1.0 + a - 1.0;
  sign(b - a);
}
```

A first example

```
void sign(double x){
  if      (x > 0.0)  printf("Positive");
  else if (x < 0.0)  printf("Negative");
  else               printf("Zero");
}
void main(){
  double a = 0x1p-53 + 0x1p-64;    // a = 2^-53 + 2^-64
  double b = 1.0 + a - 1.0;
  sign(b - a);
}
```

A first example

```
void sign(double x){
  if      (x > 0.0) printf("Positive");
  else if (x < 0.0) printf("Negative");
  else              printf("Zero");
}
void main(){
  double a = 0x1p-53 + 0x1p-64;    // a = 2^-53 + 2^-64
  double b = 1.0 + a - 1.0;
  sign(b - a);
}
```



A first example

```
void sign(double x){
  if      (x > 0.0)  printf("Positive");
  else if (x < 0.0)  printf("Negative");
  else               printf("Zero");
}
void main(){
  double a = 0x1p-53 + 0x1p-64;    // a = 2^-53 + 2^-64
  double b = 1.0 + a - 1.0;
  sign(b - a);
}
```

Architecture and rounding precision

- All current processors support IEEE-754
  - A floating-point arithmetic standard
- Some architecture-depend issues:
  - x87 floating-point unit uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - FMA(fused multiply-add) instruction supported by the PowerPC and the Intel Itanimum architecture
    - calculates $(x \times y \pm z)$ with a single rounding

Architecture and rounding precision

- All current processors support IEEE-754
  - A floating-point arithmetic standard
- Some architecture-depend issues:
  - x87 floating-point unit uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - FMA(fused multiply-add) instruction supported by the PowerPC and the Intel Itanimum architecture
    - calculates $(x \times y \pm z)$ with a single rounding

Architecture and rounding precision

- All current processors support IEEE-754
  - A floating-point arithmetic standard
- Some architecture-depend issues:
  - x87 floating-point unit uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - FMA(fused multiply-add) instruction supported by the PowerPC and the Intel Itanium architecture
    - calculates $(x \times y \pm z)$ with a single rounding

Architecture and rounding precision

- All current processors support IEEE-754
  - A floating-point arithmetic standard
- Some architecture-depend issues:
  - x87 floating-point unit uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - FMA(fused multiply-add) instruction supported by the PowerPC and the Intel Itanium architecture
    - calculates $(x \times y \pm z)$ with a single rounding

Architecture and rounding precision

- All current processors support IEEE-754
  - A floating-point arithmetic standard
- Some architecture-depend issues:
  - x87 floating-point unit uses 80-bit floating-point registers (supported by IA32 processors)
    - may lead to double rounding (the floating-point results are rounded twice)
  - FMA(fused multiply-add) instruction supported by the PowerPC and the Intel Itanimum architecture
    - calculates $(x \times y \pm z)$ with a single rounding

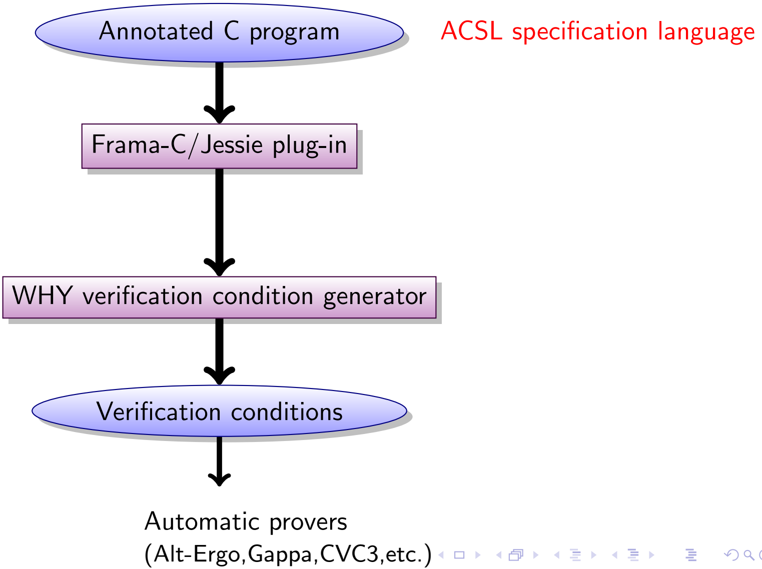
$\Longrightarrow$ introduce subtle inconsistencies between program executions

Analyzing numerical program

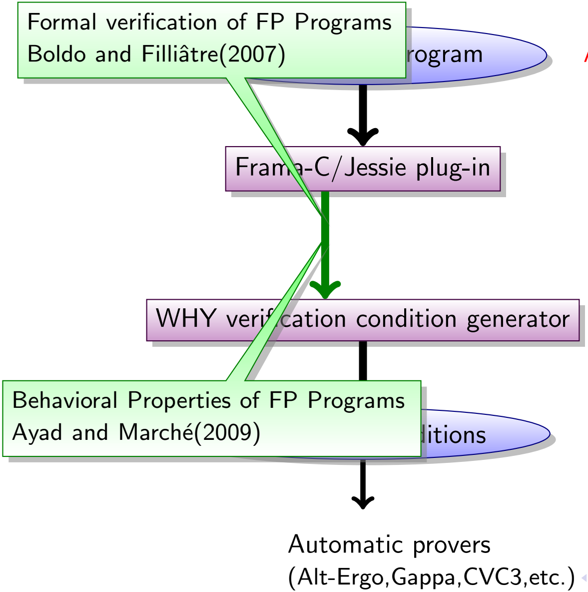
Tools for analyzing numerical programs

- Abstract interpretation:
  - Fluctuat, Astrée, etc.
- Frama-C:
  - A framework for static analysis of C code
  - Flexible: Easy to add a new plug-in
    - Value analysis: plug-in based on abstract interpretation
    - Jessie: deductive verification
- In Jessie: Easy to change the interpretation of floating-point operation

Frama-C and floating-point arithmetic



Frama-C and floating-point arithmetic



ACSL specification language

Frama-C and floating-point arithmetic

Formal verification of FP Programs
Boldo and Filliâtre(2007)

rogram

ACSL specification language

only follow the strict IEEE-754

a-C/Jessie plug-in

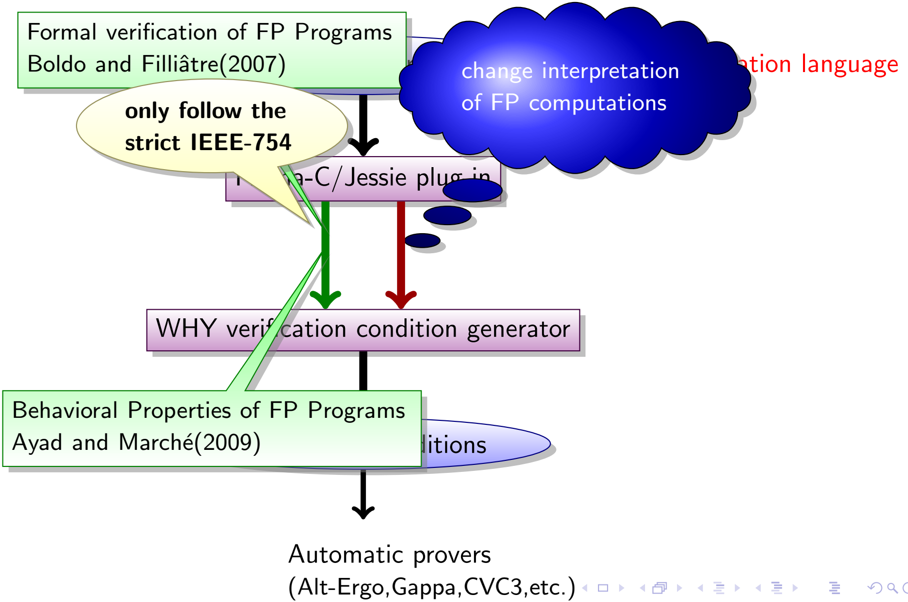
WHY verification condition generator

Behavioral Properties of FP Programs
Ayad and Marché(2009)

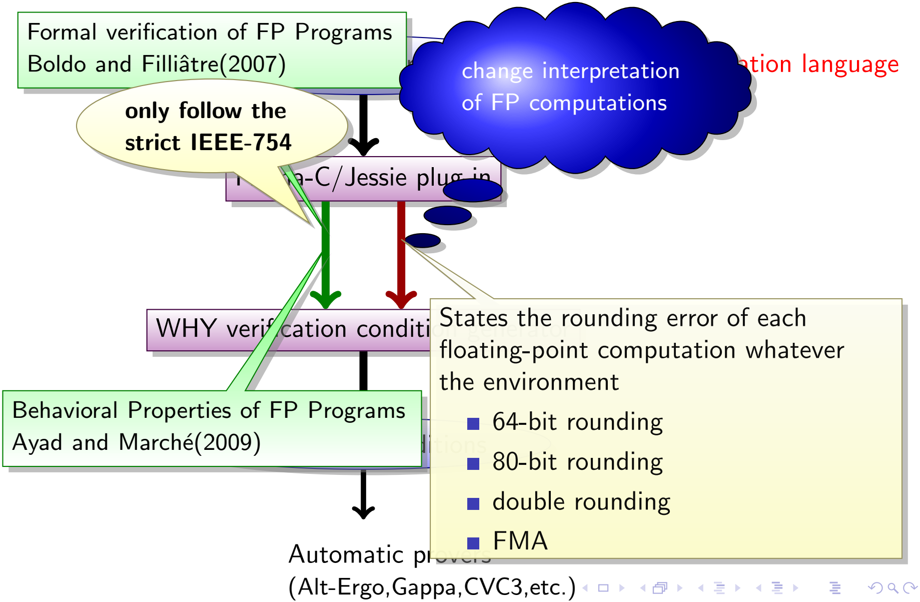
litions

Automatic provers
(Alt-Ergo,Gappa,CVC3,etc.)

Frama-C and floating-point arithmetic



Frama-C and floating-point arithmetic



Outline

Outline

1. Floating-point arithmetic
2. Floating-point computations independent to hardwares and compilers
3. A case study
4. Conclusion and future work

Floating-point number

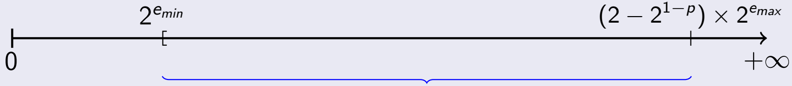
Definition

A floating-point number x in a format (p, e_{min}, e_{max}) is represented by the triple (s, m, e) so that

$$x = (-1)^s \times 2^e \times m$$

- $s \in \{0{,}1\}$
- $e_{min} \leq e \leq e_{max}$
- $0 \leq m < 2$, represented by p bits

Normal number vs. Subnormal number



$|x| \geq 2^{e_{min}}$

x is normal number

Floating-point number

Definition

A floating-point number x in a format (p, e_{min}, e_{max}) is represented by the triple (s, m, e) so that

$$x = (-1)^s \times 2^e \times m$$

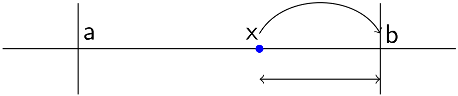
- s $\in \{0,1\}$
- $e_{min} \leq e \leq e_{max}$
- $0 \leq m < 2$, represented by p bits

Normal number vs. Subnormal number

$|x| < 2^{e_{min}}$

x is subnormal number

Rounding error



Absolute error vs. relative error

- Absolute error

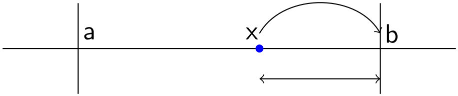
$$\epsilon(x) = |x - \circ(x)|$$

- Relative error

$$\epsilon(x) = \left|\frac{x - \circ(x)}{x}\right|$$

$\circ(x)$ is the rounding value of x

Rounding error



Absolute error vs. relative error

- Absolute error

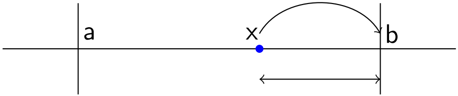
$$\epsilon(x) = |x - \circ(x)|$$

- Relative error

$$\epsilon(x) = \left|\frac{x - \circ(x)}{x}\right|$$

$\circ(x)$ is the rounding value of x

Rounding error



Absolute error vs. relative error

- Absolute error

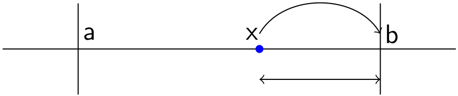
$$\epsilon(x) = |x - \circ(x)|$$

- Relative error

$$\epsilon(x) = \left|\frac{x - \circ(x)}{x}\right|$$

$\circ(x)$ is the rounding value of x

Rounding error



Rounding error in normal range

Use relative error

$$\left|\frac{x - \circ(x)}{x}\right| \leq 2^{-p}$$

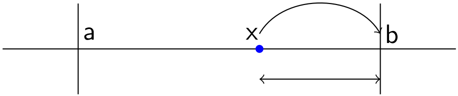
Rounding error in subnormal range

Use absolute error

$$|x - \circ(x)| \leq 2^{e_{min}-p}$$

**Using round-to-nearest mode

Rounding error



IEEE-754 double precision (64-bit rounding)

- precision $p = 53$
- $e_{min} = -1022$ and $e_{max} = 1023$

Rounding error in normal range

Use relative error

$$\left| \frac{x - \circ(x)}{x} \right| \leq 2^{-53}$$

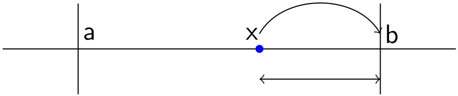
Rounding error in subnormal range

Use absolute error

$$|x - \circ(x)| \leq \alpha \quad (with\ \alpha = 2^{-1075})$$

**Using round-to-nearest mode

Rounding error



x87 (80-bit rounding)

- precision $p = 64$
- $e_{min} = -16382$ and $e_{max} = 16383$

Rounding error in normal range

Use relative error

$$\left| \frac{x - \circ(x)}{x} \right| \leq 2^{-64}$$

Rounding error in subnormal range

Use absolute error

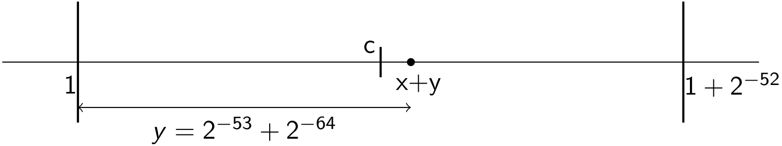
$$|x - \circ(x)| \leq \beta \quad (with\ \beta = 2^{-16446})$$

**Using round-to-nearest mode

Double rounding

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;

  printf("z=%a\n",z);
}
```



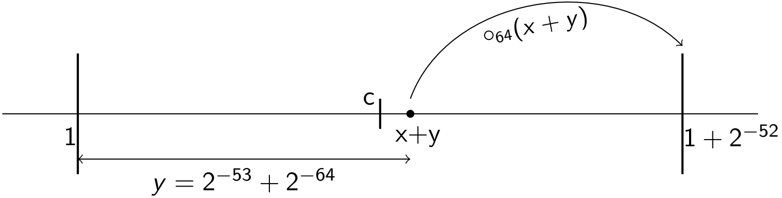
Double rounding

64-bit rounding

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;

  printf("z=%a\n",z);
}
```

$z = 1.0+2^{-52}$

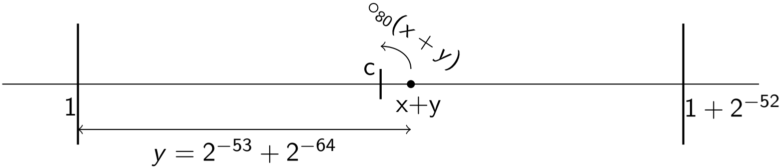


gcc double_rounding.c

Double rounding

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;

  printf("z=%a\n",z);
}
```



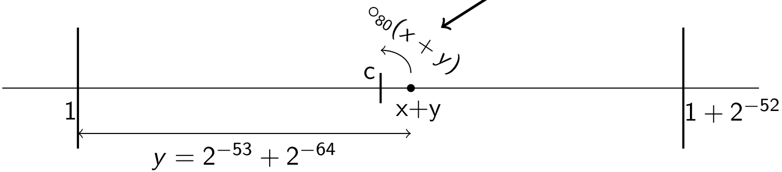
gcc -mfpmath=387 double_rounding.c

Double rounding

```
int main(){
   double x = 1.0;
   double y = 0x1p-53 + 0x1p-64;
   double z = x + y;

   printf("z=%a\n",z);
}
```

stored in 80-bit register



gcc -mfpmath=387 double_rounding.c

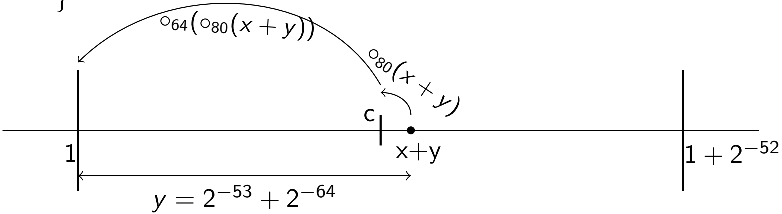
Double rounding

Double rounding

```
int main(){
  double x = 1.0;
  double y = 0x1p-53 + 0x1p-64;
  double z = x + y;

  printf("z=%a\n",z);
}
```

$z = 1.0$



gcc -mfpmath=387 double_rounding.c

Rounding error in double rounding

Based on 64-bit and 80-bit rounding, with $\alpha = 2^{-1022}$

- $|x| \geq \alpha \Rightarrow \left| \frac{x - \circ_{64}(\circ_{80}(x))}{x} \right| \leq \beta$
- $|x| \leq \alpha \Rightarrow |x - \circ_{64}(\circ_{80}(x))| \leq \gamma$

with $\beta = 2050 \times 2^{-64}$
$\gamma = 2049 \times 2^{-1086}$

Outline

C expression

$$a = x * y + z$$

C expression

$$a = \boxed{x * y} + z$$

is interpreted as

$$\boxed{\square(x \times y)}$$

where $\square$ is one of the following rounding:

- 64-bit rounding
- 80-bit rounding
- double rounding

C expression

$$a = \boxed{\boxed{x * y} + z}$$

is interpreted as

$$a = \boxed{\Box(\boxed{\Box(x \times y)} + z)}$$

where $\Box$ is one of the following rounding:

- 64-bit rounding
- 80-bit rounding
- double rounding

Theorem 1

Theorem

For a real number x*, let* $\Box(x)$ *be either* $\circ_{64}(x)$*, or* $\circ_{80}(x)$*, or the double rounding* $\circ_{64}(\circ_{80}(x))$*.*
With $\alpha = 2^{-1022}$*, we have either*

$$|x| \geq \alpha \textit{ and } \left|\frac{x - \Box(x)}{x}\right| \leq \beta \textit{ and } |\Box(x)| \geq \alpha$$

or

$$|x| \leq \alpha \textit{ and } |x - \Box(x)| \leq \gamma \textit{ and } |\Box(x)| \leq \alpha.$$

with $\beta = 2050 \times 2^{-64}$
$\gamma = 2049 \times 2^{-1086}$

Theorem 1

Theorem

For a real number x, let $\Box(x)$ be either $\circ_{64}(x)$, or $\circ_{80}(x)$, or the double rounding $\circ_{64}(\circ_{80}(x))$.
With $\alpha = 2^{-1022}$, we have either

$$|x| \geq \alpha \text{ and } \left|\frac{x - \Box(x)}{x}\right| \leq \beta \text{ and } |\Box(x)| \geq \alpha$$

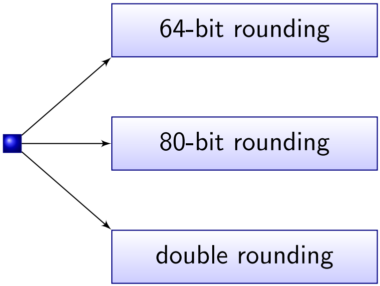
or

$$|x| \leq \alpha \text{ and } |x - \Box(x)| \leq \gamma \text{ and } |\Box(x)| \leq \alpha.$$

with $\beta = 2050 \times 2^{-64}$
$\gamma = 2049 \times 2^{-1086}$

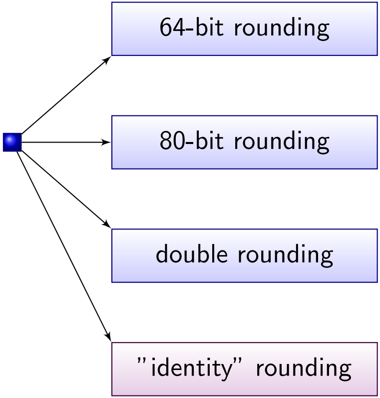
proved in Coq
(Available at `http://www.lri.fr/~nguyen/research/rnd_64_80_post.html`)

Rounding error in presence of FMA



FMA (Fused Multiply-Add)

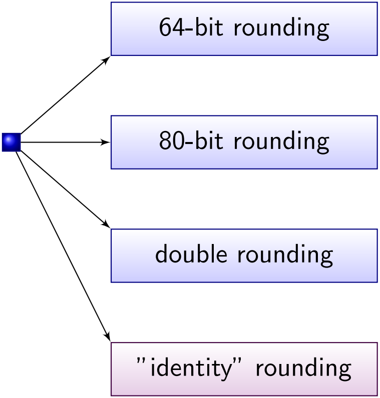
Rounding error in presence of FMA



$$\Box(x) = x$$

FMA (Fused Multiply-Add)

Rounding error in presence of FMA

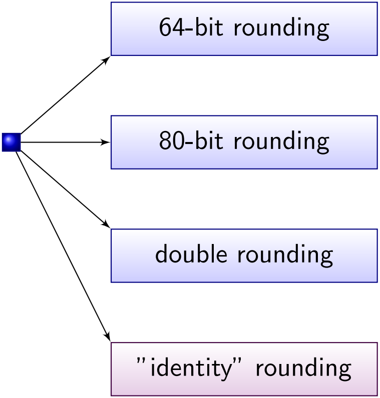


$$\square(x) = x$$

FMA (Fused Multiply-Add)

$$\circ(x \times y \pm z)$$

Rounding error in presence of FMA



$$\square(x) = x$$

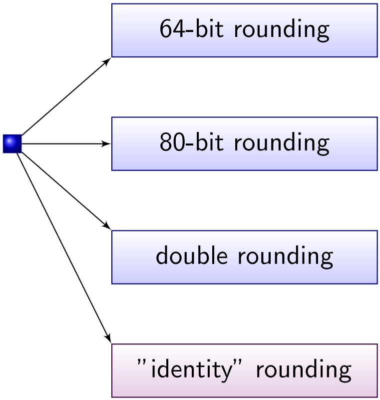
FMA (Fused Multiply-Add)

$$\circ(x \times y \pm z)$$

$$\downarrow$$

$$\square(\square(x \times y) \pm z)$$

Rounding error in presence of FMA



$$\square(x) = x$$

FMA (Fused Multiply-Add)

$$\square(\boxed{\square(x \times y)} \pm z)$$

"identity" rounding

Theorem 2

Theorem

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval $\mathcal{I}$ for the final result, then the really computed final result is in $\mathcal{I}$ whatever the architecture and the compiler that preserves the order of operations.

Theorem 2

Theorem

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval $\mathcal{I}$ for the final result, then the really computed final result is in $\mathcal{I}$ whatever the architecture and the compiler that preserves the order of operations.

$$\boxed{\begin{array}{l} x_1 = a \times b \\ x_2 = x_1 + c \\ \vdots \\ x_n = x_{n-1} * d \end{array}}$$

Theorem 2

Theorem

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval $\mathcal{I}$ for the final result, then the really computed final result is in $\mathcal{I}$ whatever the architecture and the compiler that preserves the order of operations.

$$\begin{aligned} x_1 &= a \times b \\ x_2 &= x_1 + c \\ &\vdots \\ x_n &= x_{n-1} * d \end{aligned}$$

$$\begin{aligned} &\xrightarrow{\text{Theorem 1}} x_1 \in \mathcal{I}_1(a, b) \\ &\xrightarrow{\text{Theorem 1}} x_2 \in \mathcal{I}_2(x_1, c) \\ &\vdots \\ &\xrightarrow{\text{Theorem 1}} x_n \in \mathcal{I}_n(x_{n-1}, d) \end{aligned}$$

Frama-C

Theorem 2

Theorem

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval $\mathcal{I}$ for the final result, then the really computed final result is in $\mathcal{I}$ whatever the architecture and the compiler that preserves the order of operations.

$$\begin{aligned} x_1 &= a \times b \\ x_2 &= x_1 + c \\ &\vdots \\ x_n &= x_{n-1} * d \end{aligned}$$

$$\begin{aligned} &\xrightarrow{\textit{Theorem 1}} \quad x_1 \in \mathcal{I}_1(a, b) \\ &\xrightarrow{\textit{Theorem 1}} \quad x_2 \in \mathcal{I}_2(x_1, c) \\ &\vdots \\ &\xrightarrow{\textit{Theorem 1}} \quad x_n \in \mathcal{I}_n(x_{n-1}, d) \end{aligned}$$

$$\longrightarrow x_n \in \mathcal{I}$$

Frama-C **Gappa**

Theorem 2

Theorem

If we define each result of an operation by the formulas of Theorem 1, and if we are able to deduce from these intervals an interval $\mathcal{I}$ for the final result, then the really computed final result is in $\mathcal{I}$ whatever the architecture and the compiler that preserves the order of operations.

$$
\begin{aligned}
x_1 &= a \times b\\
x_2 &= x_1 + c\\
&\vdots\\
x_n &= x_{n-1} * d
\end{aligned}
\qquad
\begin{aligned}
&\xrightarrow{\textit{Theorem 1}} x_1 \in \mathcal{I}_1(a, b)\\
&\xrightarrow{\textit{Theorem 1}} x_2 \in \mathcal{I}_2(x_1, c)\\
&\vdots\\
&\xrightarrow{\textit{Theorem 1}} x_n \in \mathcal{I}_n(x_{n-1}, d)
\end{aligned}
\qquad
\longrightarrow x_n \in \mathcal{I}
$$

Correct $\forall$ compiler and architecture

Frama-C **Gappa**

Outline

A case study

KB3D(NASA Langley Research Center)

- An aircraft conflict detection and resolution program
- Formally proved correct using PVS (C. Muñoz, G. Dowek. . .)
- Provided the calculations are exact

Our case study

- Use a small part of KB3D
- Make a decision corresponding to value -1 and 1 to decide if the plane will go to its left or its right

A case study

KB3D(NASA Langley Research Center)

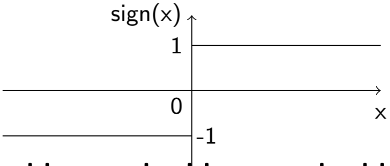
- An aircraft conflict detection and resolution program
- Formally proved correct using PVS (C. Muñoz, G. Dowek...)
- Provided the calculations are exact

Our case study

- Use a small part of KB3D
- Make a decision corresponding to value -1 and 1 to decide if the plane will go to its left or its right

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;    // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;   // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;     // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;    // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```



```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;      // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;     // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;    // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;   // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;    // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;   // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

```
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy,double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;    // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy = 0x1.fffffffffffffp-1;    // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

gcc eps_line.c

Result = 1

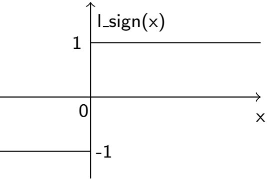
```c
int sign(double x) {
  if (x >= 0) return 1;
  else return -1;
}
int eps_line(double sx, double sy, double vx, double vy) {
  int s1,s2;
  s1=sign(sx*vx+sy*vy);
  s2=sign(sx*vy-sy*vx);
  return s1*s2;
}
int main(){
  double sx = -0x1.0000000000001p0;    // sx = -1 - 2^-52
  double vx = -1.0;
  double sy = 1.0;
  double vy =  0x1.fffffffffffffp-1;   // vy = 1 - 2^-53
  int result = eps_line(sx,sy,vx,vy);
  printf("Result = %d\n",result);
}
```

gcc -mfpmath=387 eps_line.c

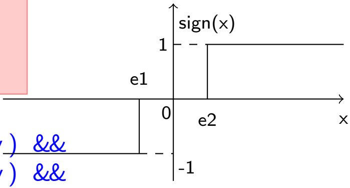
Result = -1

```c
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures \abs(\result) <= 1 &&
  @         (\result != 0 ==> \result == l_sign(\exact(x)));*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
/*@ requires
  @  sx == \exact(sx)  && sy == \exact(sy) &&
  @  vx == \exact(vx)  && vy == \exact(vy) &&
  @  \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @  \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @  \result != 0 ==>
  @  \result==l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @          * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  int s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```

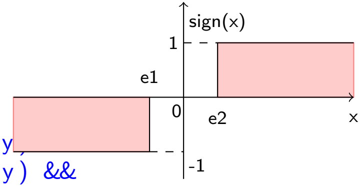
```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures \abs(\result) <= 1 &&
  @         (\result != 0 ==> \result == l_sign(\exact(x)));*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
/*@ requires
  @  sx == \exact(sx)  && sy == \exact(sy) &&
  @  vx == \exact(vx)  && vy == \exact(vy) &&
  @  \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @  \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @  \result != 0 ==>
  @  \result==l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @          * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  int s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```



```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures \abs(\result) <= 1 &&
  @         (\result != 0 ==> \result == l_sign(\exact(x)));*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
/*@ requires
  @  sx == \exact(sx)  && sy == \exact(sy) &&
  @  vx == \exact(vx)  && vy == \exact(vy) &&
  @  \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @  \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @  \result != 0 ==>
  @  \result==l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @         * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  int s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```

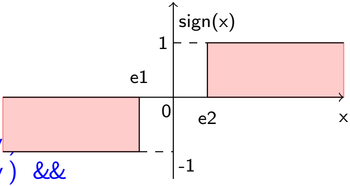


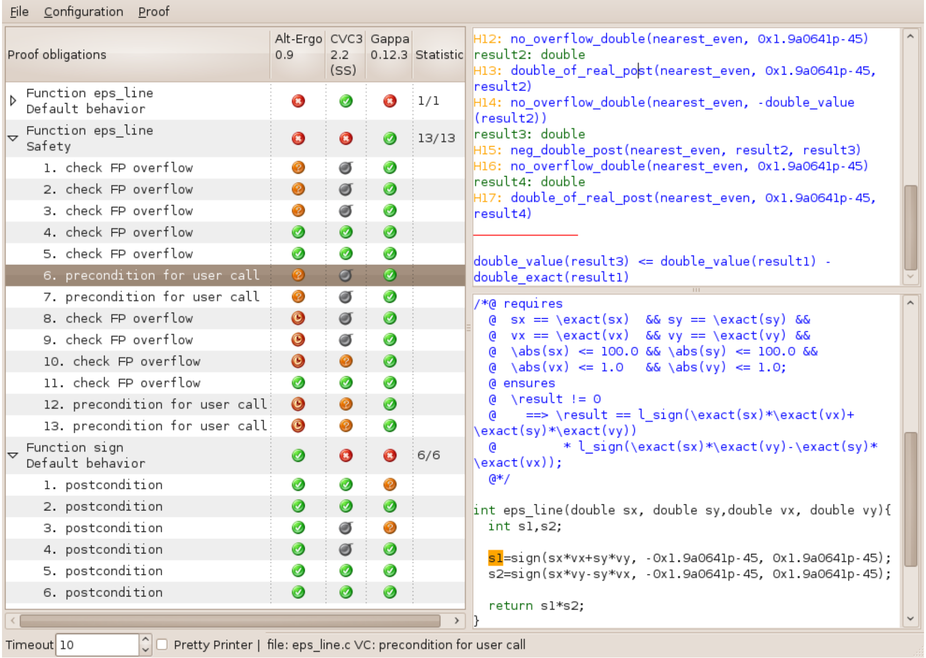
```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures \abs(\result) <= 1 &&
  @           (\result != 0 ==> \result == l_sign(\exact(x)));*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
```



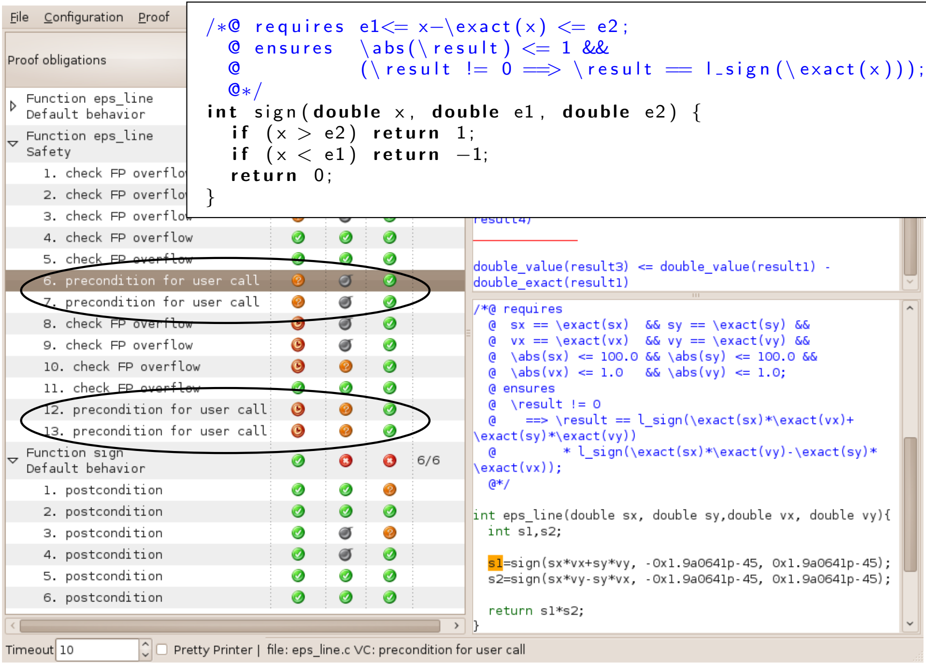

```
/*@ requires
  @  sx == \exact(sx)  && sy == \exact(sy)
  @  vx == \exact(vx)  && vy == \exact(vy) &&
  @  \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @  \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @  \result != 0 ==>
  @  \result==l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @         * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  int s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```

```
//@ logic integer l_sign(real x) = (x >= 0.0) ? 1 : -1;
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures \abs(\result) <= 1 &&
  @         (\result != 0 ==> \result == l_sign(\exact(x)));*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
/*@ requires
  @  sx == \exact(sx)  && sy == \exact(sy
  @  vx == \exact(vx)  && vy == \exact(vy) &&
  @  \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
  @  \abs(vx) <= 1.0   && \abs(vy) <= 1.0;
  @ ensures
  @  \result != 0 ==>
  @  \result==l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
  @        * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
  @*/
int eps_line(double sx, double sy,double vx, double vy){
  int s1=sign(sx*vx+sy*vy, -0x1.90641p-45, 0x1.90641p-45);
  int s2=sign(sx*vy-sy*vx, -0x1.90641p-45, 0x1.90641p-45);
  return s1*s2;
}
```

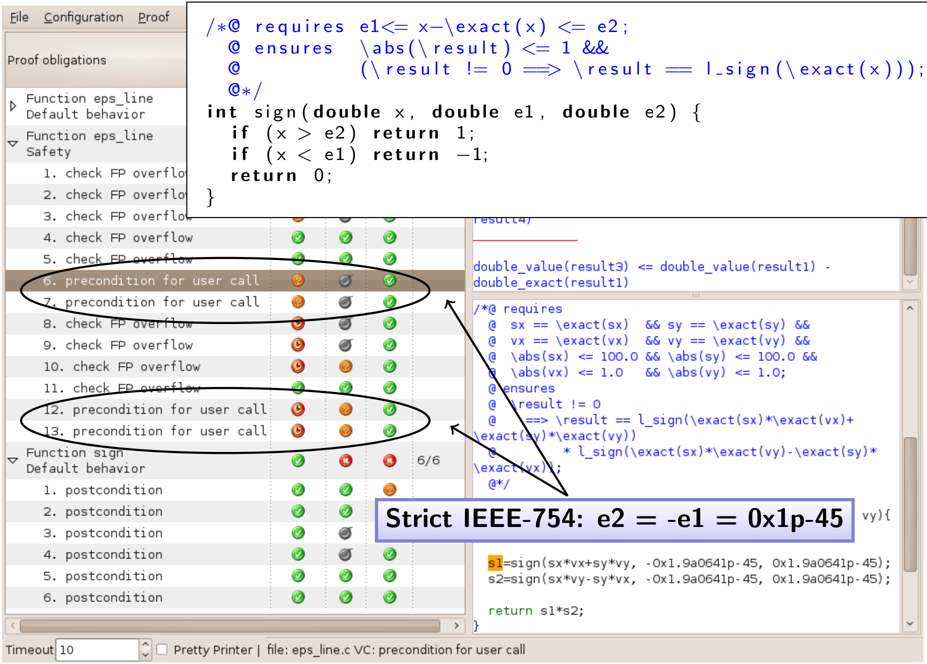



File Configuration Proof
Proof obligations
Alt-Ergo 0.9
CVC3 2.2 (SS)
Gappa 0.12.3
Statistic
Function eps_line Default behavior 1/1
Function eps_line Safety 13/13
1. check FP overflow
2. check FP overflow
3. check FP overflow
4. check FP overflow
5. check FP overflow
6. precondition for user call
7. precondition for user call
8. check FP overflow
9. check FP overflow
10. check FP overflow
11. check FP overflow
12. precondition for user call
13. precondition for user call
Function sign Default behavior 6/6
1. postcondition
2. postcondition
3. postcondition
4. postcondition
5. postcondition
6. postcondition
H12: no_overflow_double(nearest_even, 0x1.9a0641p-45)
result2: double
H13: double_of_real_post(nearest_even, 0x1.9a0641p-45, result2)
H14: no_overflow_double(nearest_even, -double_value (result2))
result3: double
H15: neg_double_post(nearest_even, result2, result3)
H16: no_overflow_double(nearest_even, 0x1.9a0641p-45)
result4: double
H17: double_of_real_post(nearest_even, 0x1.9a0641p-45, result4)
double_value(result3) <= double_value(result1) - double_exact(result1)
/*@ requires
@ sx == \exact(sx) && sy == \exact(sy) &&
@ vx == \exact(vx) && vy == \exact(vy) &&
@ \abs(sx) <= 100.0 && \abs(sy) <= 100.0 &&
@ \abs(vx) <= 1.0 && \abs(vy) <= 1.0;
@ ensures
@ \result != 0
@ ==> \result == l_sign(\exact(sx)*\exact(vx)+\exact(sy)*\exact(vy))
@ * l_sign(\exact(sx)*\exact(vy)-\exact(sy)*\exact(vx));
@*/
int eps_line(double sx, double sy,double vx, double vy){
int s1,s2;
s1=sign(sx*vx+sy*vy, -0x1.9a0641p-45, 0x1.9a0641p-45);
s2=sign(sx*vy-sy*vx, -0x1.9a0641p-45, 0x1.9a0641p-45);
return s1*s2;
}
Timeout 10
Pretty Printer | file: eps_line.c VC: precondition for user call

```
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  \abs(\result) <= 1 &&
  @          (\result != 0 ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
```

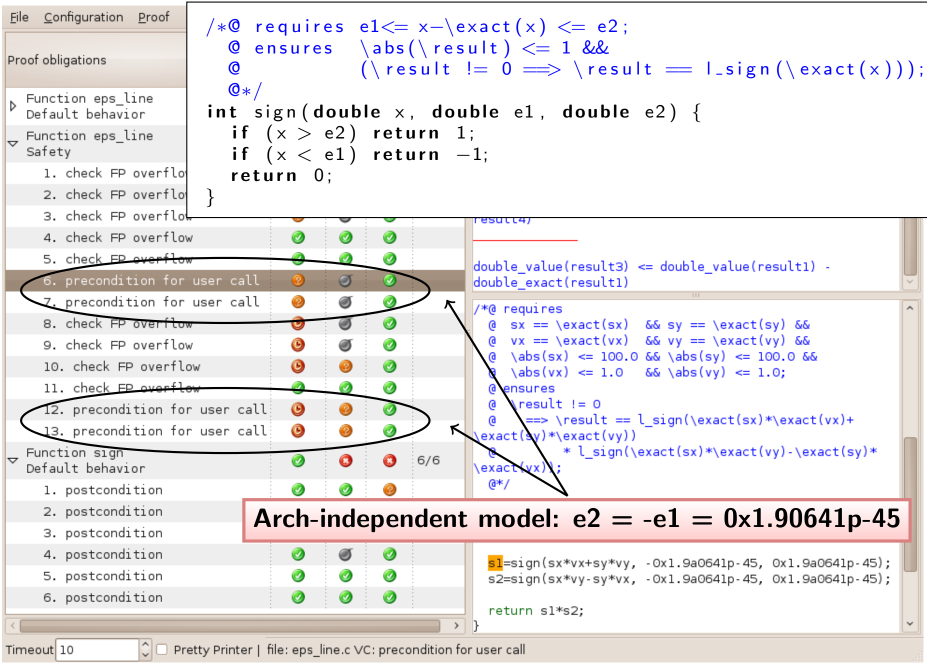


```
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  \abs(\result) <= 1 &&
  @          (\result != 0 ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
```



Strict IEEE-754: e2 = -e1 = 0x1p-45

```
/*@ requires e1<= x-\exact(x) <= e2;
  @ ensures  \abs(\result) <= 1 &&
  @          (\result != 0 ==> \result == l_sign(\exact(x)));
  @*/
int sign(double x, double e1, double e2) {
  if (x > e2) return 1;
  if (x < e1) return -1;
  return 0;
}
```



Arch-independent model: e2 = -e1 = 0x1.90641p-45

Outline

Conclusion

An approach

- gives correct rounding errors whatever the architecture and the choices of the compiler
- is implemented in the Frama-C for all basic operations with the same conditions
- can be applied to single precision computation
- is proved correct in Coq

Drawback

- time to run a program verification (10s for a proof obligation)
- Incomplete: only proves rounding errors

Future work

- reduce time to run
- allow the compiler to do anything, including re-organizing the operations
  Example: if $|e| \ll |x|$
  - $(e + x) - x = 0$
  - $e + (x - x) = e$
- look into the assembly
  - know the order of the operations
  - know the precision used for each operation
  - know if the architecture supports FMA or not

Thank you for your attention!