Taming floating-point rounding errors with proofs

Laura Titolo
Code Metal, USA

16th International Conference on Interactive Theorem Proving - ITP 2025
Reykjavík, 28th September 2025

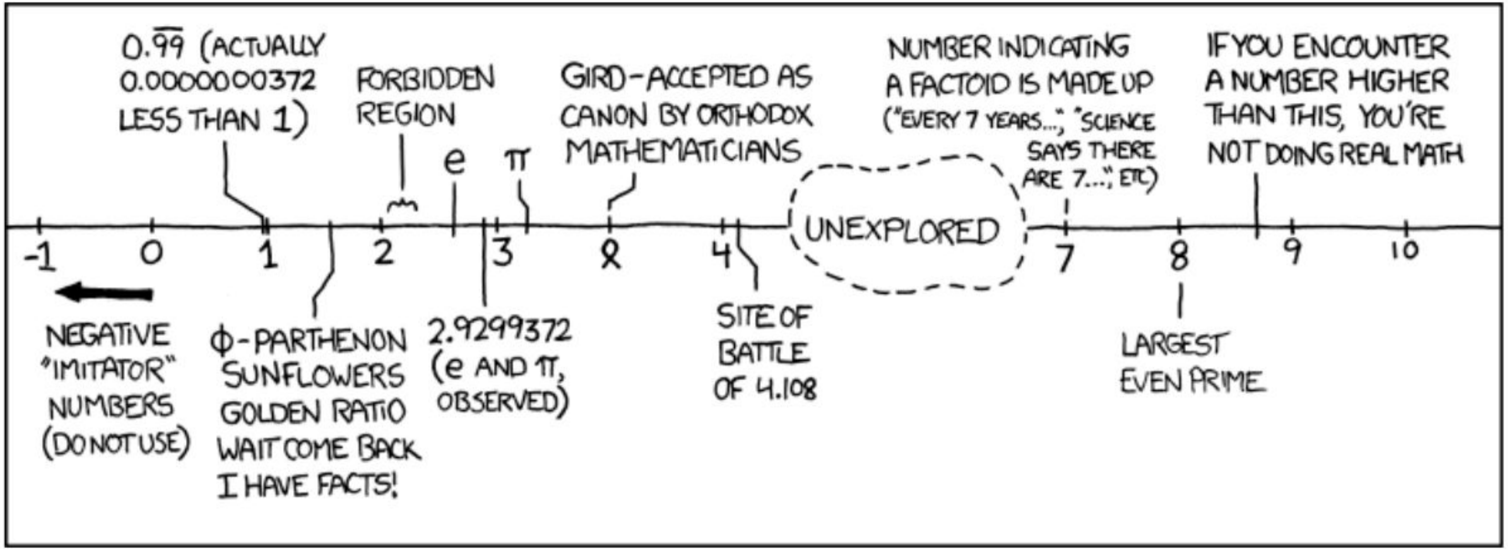
Outline

💻🔍 application of
ITP + other formal methods
for **floating-point** program
analysis

🚀👨‍🚀 ITP at **NASA**

🤖🧠 ITP for **AI** and AI for ITP in
industry

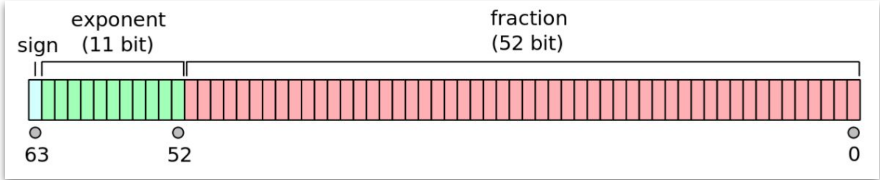
Representing Real Numbers in Digital Computers



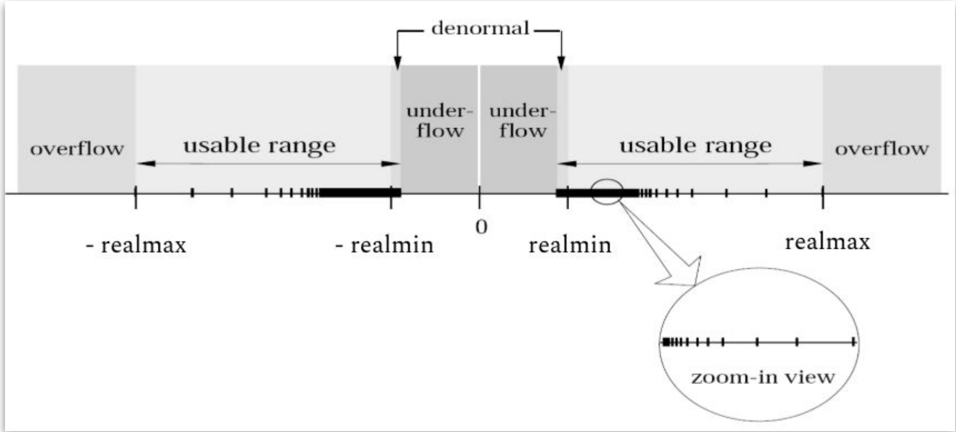
Credits by XKCD

Floating-Point Numbers and Rounding errors

💻 Floating-point (FP) = **finite representation** of real numbers



FP ⊂ Reals
Example: 0.1 is not a FP
(rounded to 3602879701896397 / 36028797018963968)



⚠️ **Round-off error** = | r - round(r) |

Credits by illinois.edu

Writing Floating-Point Code is hard!

```
Prelude> (4/3 -1) * 3 - 1
```

Writing Floating-Point Code is hard!

```
Prelude> (4/3 -1) * 3 - 1
-2.220446049250313e-16
```

Result is **0** if evaluated in exact real number arithmetic

Writing Floating-Point Code is hard!

```
Prelude> (4/3 -1) * 3 - 1
-2.220446049250313e-16
Prelude> floor((4/3 -1) * 3 - 1)
-1
```

Result is **0** if evaluated in exact real number arithmetic

Writing Floating-Point Code is hard!

```
Prelude> (4/3 -1) * 3 - 1
-2.220446049250313e-16
Prelude> floor((4/3 -1) * 3 - 1)
-1
Prelude> if (floor((4/3 -1) * 3 - 1)) < 0 then 100 else 1
100
Prelude>
```

Result is **0** if evaluated in exact real number arithmetic

The Boolean guard is evaluated to **false in real arithmetic** and to **true in floating-point arithmetic**

Writing Floating-Point Code is hard!

```
Prelude> (4/3 -1) * 3 - 1
-2.220446049250313e-16
Prelude> floor((4/3 -1) * 3 - 1)
-1
Prelude> if (floor((4/3 -1) * 3 - 1)) < 0 then 100 else 1
100
Prelude> 
```

Accumulated round-off error = 99 😥

Floating-point behavior is difficult to predict!

- 🌍 Floating-point numbers are ubiquitous
- 🧑‍💻“Many **developers** do not understand core floating point behavior particularly well, yet believe they do.”
- ❗ Reals arithmetic ≠ FP arithmetic (associativity, commutativity, …)
- ⚠️ **Rounding errors, overflows, underflows…**

2018 IEEE International Parallel and Distributed Processing Symposium

Do Developers Understand IEEE Floating Point?

Peter Dinda Conor Hetland
Northwestern University

***Abstract*—Floating point arithmetic, as specified in the IEEE standard, is used extensively in programs for science and engineering. This use is expanding rapidly into other domains, for example with the growing application of machine learning everywhere. While floating point arithmetic often appears to be arithmetic using real numbers, or at least numbers in scientific notation, it actually has a wide range of gotchas. Compiler and hardware implementations of floating point inject additional surprises. This complexity is only increasing as different levels of precision are becoming more common and there are even proposals to automatically reduce program precision (reducing power/energy and increasing performance) when results are deemed “good enough.” Are software developers who depend on floating point aware of these issues? Do they understand how floating point can bite them? To find out, we conducted an anonymous study of different groups from academia, national labs, and industry. The participants in our sample did only slightly better than chance in correctly identifying key unusual behaviors of the floating point standard, and poorly understood which compiler and architectural optimizations were non-standard. These surprising results and others strongly suggest caution in the face of the expanding complexity and use of floating point arithmetic.**

***Keywords*-floating point arithmetic, software development, user studies, correctness, IEEE 754**

bounds [13], and approximate computing [9] where performance/energy and output quality can be traded off. More immediately, any programmer using a modern compiler is faced with dozens of flags that control floating point optimizations which could affect results. Optimizing a program across the space of flags has itself become a subject of research [5].

The floating point arithmetic experienced by a software developer via a particular hardware implementation, language, and compiler, is swelling in complexity at the very same time that the demand for such developers is also growing. This may be setting the stage for increasing problems with numeric correctness in an increasing range of programs. Numeric issues can produce major effects. Recall that Lorenz’s insight, a cornerstone of chaos theory, was triggered by a seemingly innocuous rounding error [10]. Arguably, modern applications, certainly those that model systems with chaotic dynamics, could see small errors in developer understanding of floating point become amplified into bad overall results.

Do software developers understand the core properties of floating point arithmetic? Do they grasp which optimizations might result in non-compliance with the IEEE standard?

Floating-point errors may be dangerous and costly

Patriot missile failure (1991)

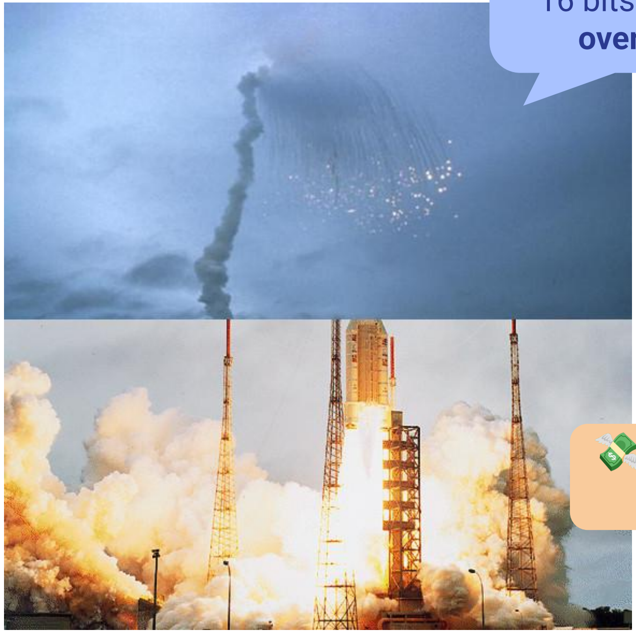
Accumulated **round-off error** in the time window calculation (1/10 constant used)

🪖28 soldiers were killed and 100 injured

Picture: Bernd vdB, Public domain, via Wikimedia Commons

Ariane V (1996)

64 bits floating-point to 16 bits integer **overflow**



💸 US$370 million loss

Photographs credits: 1996 © ESA/CNES; 1996 © ESA; 1996 © ESA.

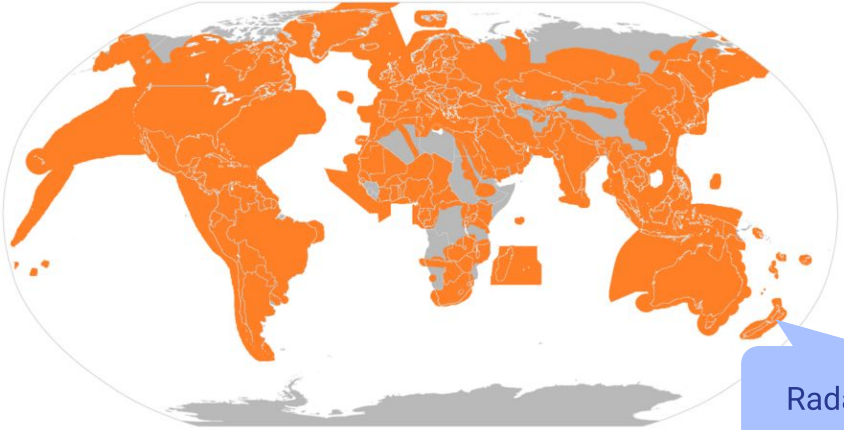
Automatic Dependent Surveillance - Broadcast (ADS-B)

- **NextGen** (Next generation of air traffic management systems) to enhance radar technologies
- Aircraft periodically **broadcasts** surveillance information
- **Automatic** — no pilot intervention
- **Mandatory** from Jan 1, 2020 (in USA and Europe)
- **Thousands of aircraft** currently equipped with ADS-B

ADS-B: Pro and Cons

✅ Pros: Broadcast vs. radar

- 🎯 More precise
- 🌍 More coverage



Radar coverage

❌ Cons: Makes use of **existing hardware**

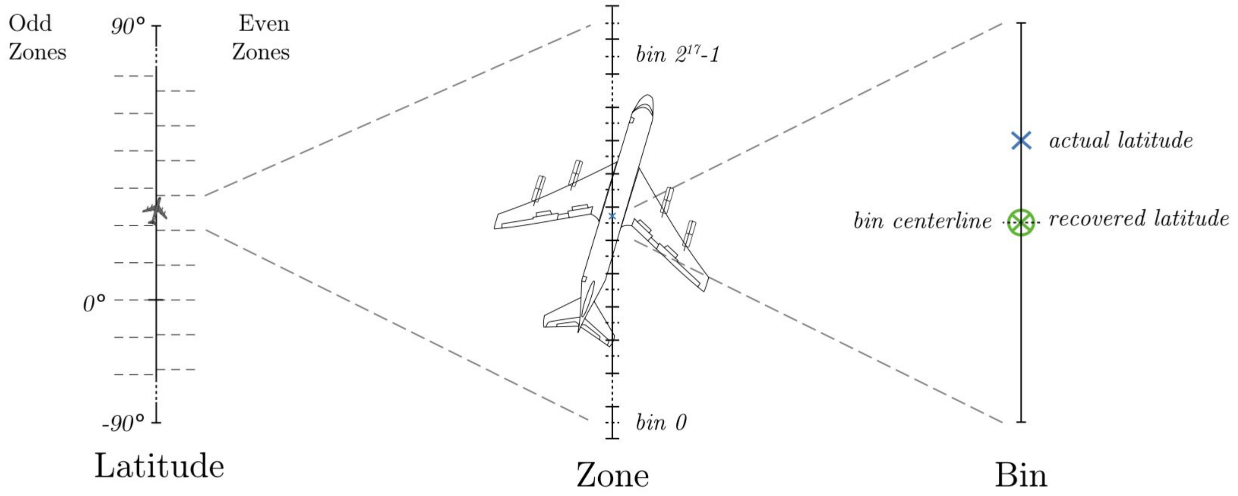
- 📡 TCAS transponders
- 📏 **35 bits** for position in the broadcast message (lat + lon)
- 🌐 Too coarse granularity (~**300 mt**) for **raw positions**

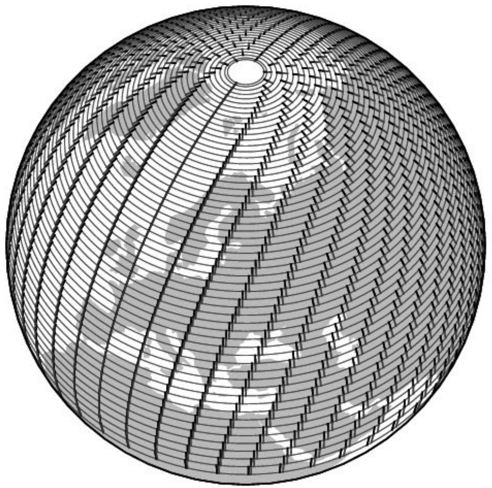
ADS-B CPR: Compact Position Reporting Algorithm

🌍 Divide the globe into 59 (odd) or 60 (even) equally sized **zones**

➗ Divide each zone in 2^17 **bins**

📍 Zone number + Bin number = Position ± ≈ **5.1mt**





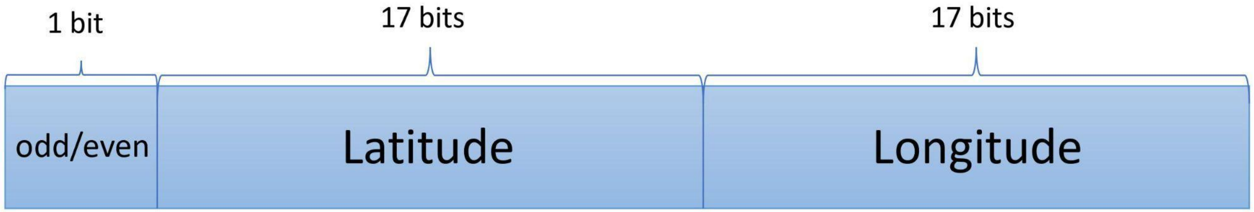
CPR Encoding

$$lat\_enc(i, lat) = \mathrm{mod}\left(\left\lfloor 2^{17}\frac{\mathrm{mod}\,(lat,\, dlat_i)}{dlat_i} + \frac{1}{2}\right\rfloor, 2^{17}\right)$$

🔢 The encoding computes the **bin number**

📤 Only the bin numbers and encoding type are transmitted

🚫 Not the zone!

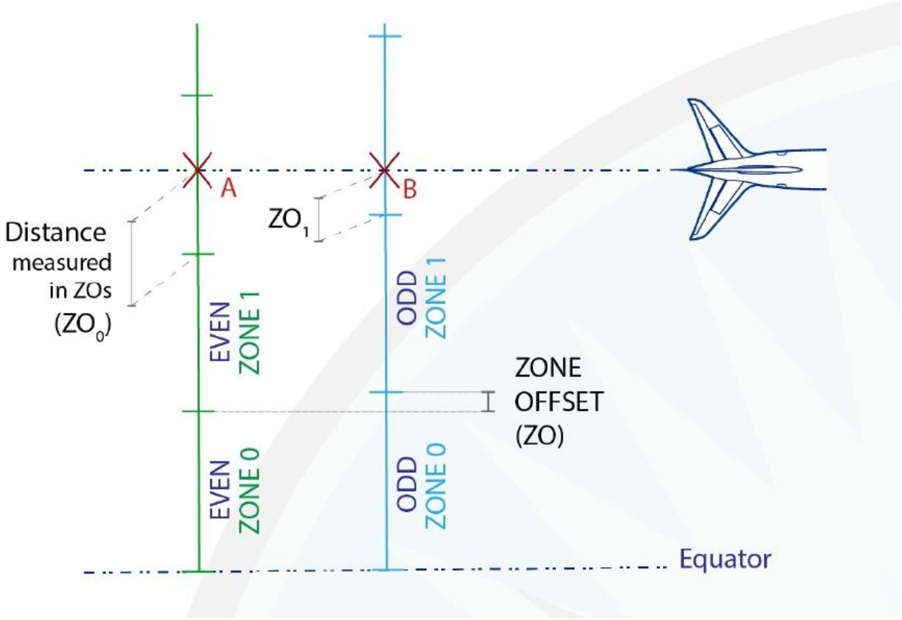


CPR Decoding

- Both **zone** and **bin number** are necessary to recover the position

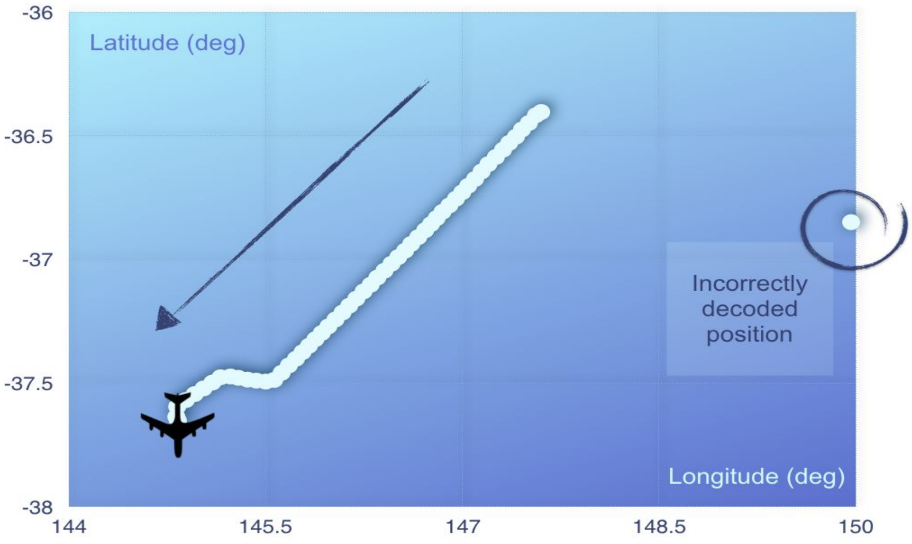
- CPR needs a **reference position or an odd and an even message** sent **sufficiently close** to each other to compute the decoded zone

- **Zone offset number** determines the zone number



CPR issues

ADS-B Compact Position Reporting Algorithm - CPR (2016)



📡 Situation as reported by Airservices Australia 🛬

📍🌐 | actual position - decoded position |

= 241 nautical miles

= 446 km

= 277 miles

What was the issue with CPR?

Dutle A., Moscato M., Titolo L., Muñoz C. *A Formal Analysis of the Compact Position Reporting Algorithm*. VSTTE 2017.

Requirements were **not enough** to guarantee the intended precision even assuming exact real-number arithmetic.

We proposed a slightly **tightened set of requirements** and **formally proved** the algorithm correct in PVS **assuming exact arithmetic**

| original_position 📍 - recovered_position 💻 | ≤ 5.1mt = algorithm accuracy ✅

What about **finite-precision** implementations?

- ⚠️ Heavy use of **modulus** and **floor** ⇒ huge accumulated round-off error
- ❌ Using single-precision floating-point, the recovered position of was off by approx **1500 nautical miles**!

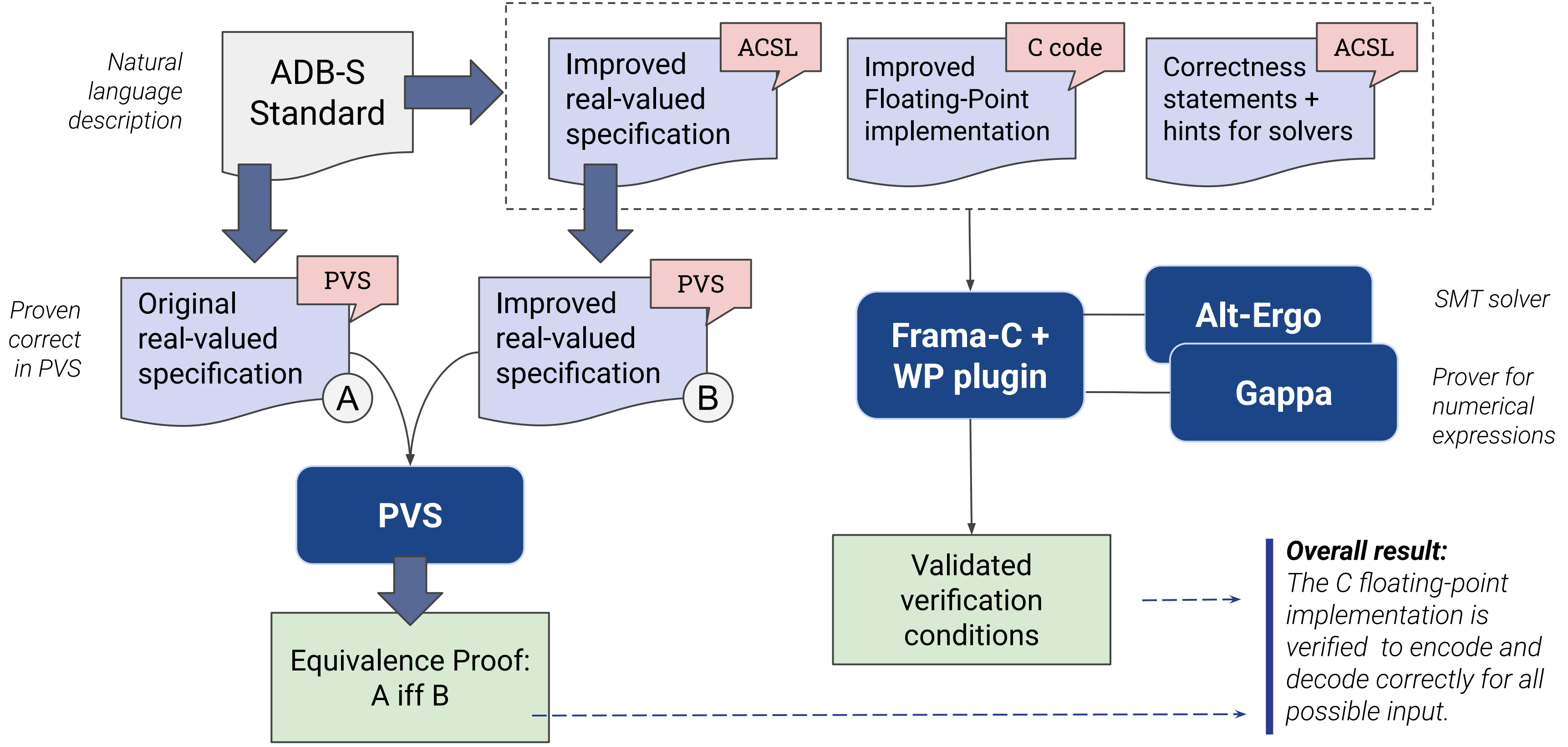
Formally verified finite-precision implementation of CPR

- Propose simpler formulation reducing numerical complexity (**CPR***)
- Use a suite of existing **formal methods** tools to provide a verified and correct implementation of CPR* with finite precision arithmetic:

frama C Software Analyzers + PVS + gappa

Titolo L., Moscato M., Muñoz C., Dutle A., Bobot F. *A Formally Verified Floating-Point Implementation of the Compact Position Reporting Algorithm*. FM 2018.

CPR floating-point implementation verification



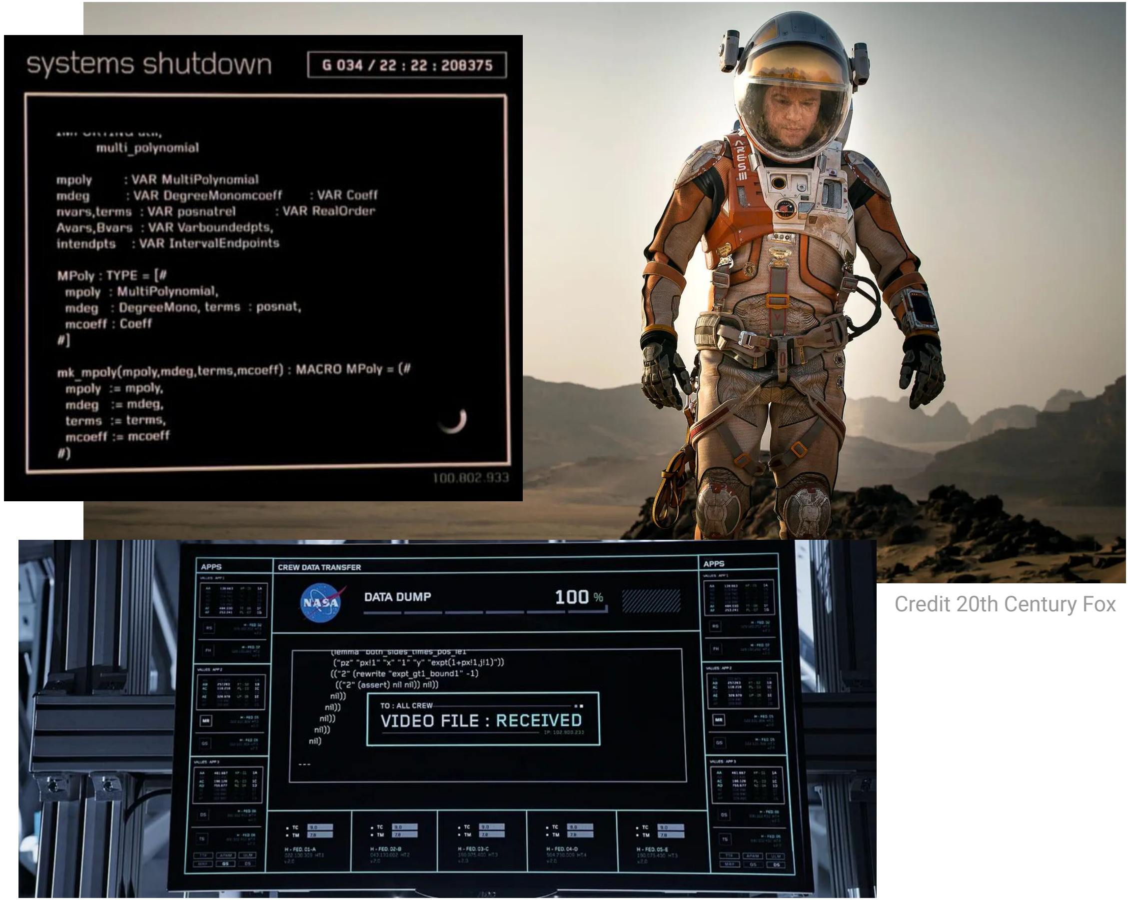
Overall result:
The C floating-point implementation is verified to encode and decode correctly for all possible input.

Why PVS?

⚙️ NASA PVS Library has been developed for about 4 decades and contains 79 libraries and 40K theorems

🎬 PVS is the only theorem prover featured in a Hollywood movie

🚀 PVS actually used to verify Mars Rovers' plans (PLEXIL-V)



Credit 20th Century Fox

CPR*

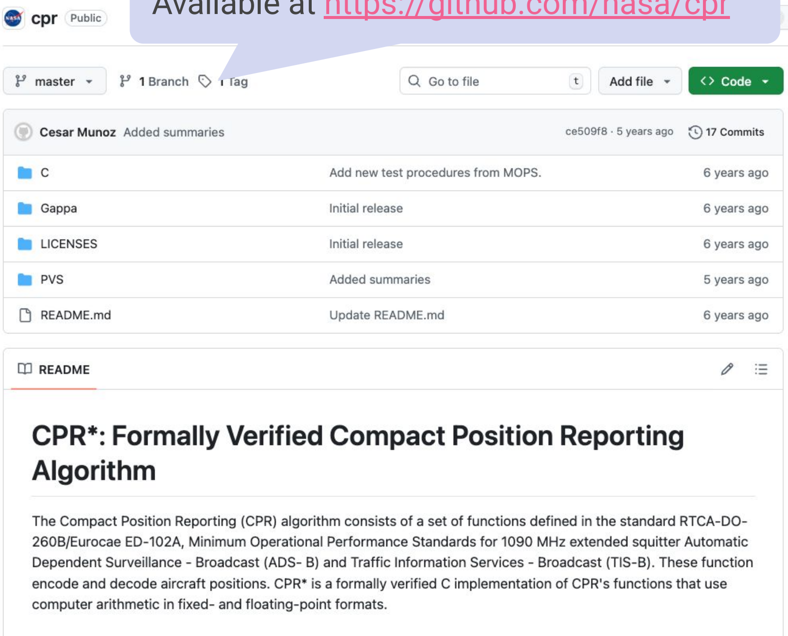
✅ CPR* = **Formally verified** C implementation of CPR

- double-precision floating-point
- 32 bits unsigned integers

📄 **Reference implementation** in the revised ADS-B **standard** document (RTCA DO-260B/Eurocae ED-102A)

🏆 NASA group achievement award 2020

Available at https://github.com/nasa/cpr



CPR* Verification approach

🚀 Successful combination of different formal methods tools

🛑 The CPR verification pipeline required a lot of expertise and manual effort!

- 📝 **Manual annotation** of the C code
- 💡 **Gappa** needs **hints** to verify a tight rounding error
- ⏳ A lot of **time and effort** spend in writing proofs in PVS
- 🎓 Expertise in floating-point arithmetic and ITP

🎯 Goal: Make this process **fully automatic** 🤖

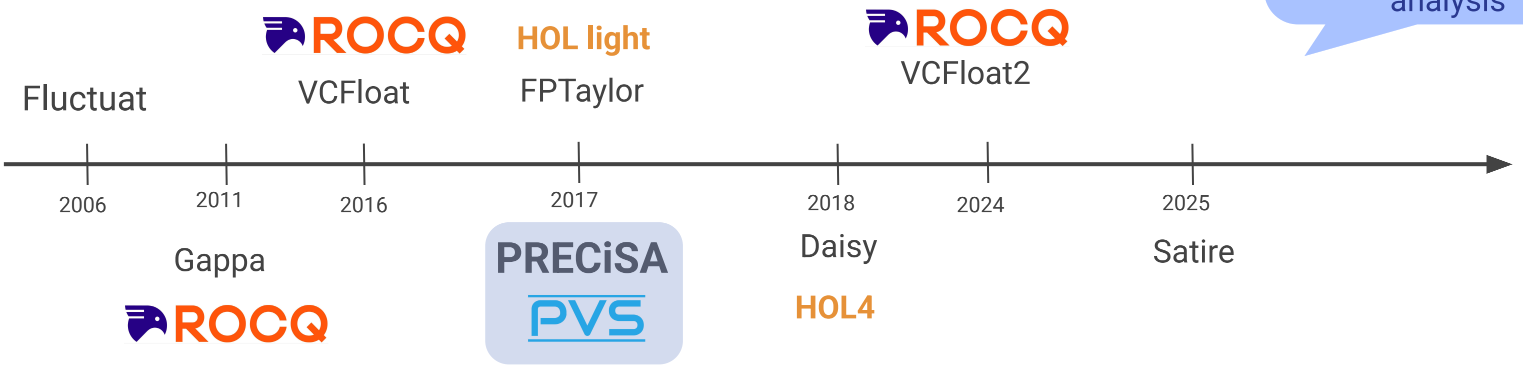
Automatic floating-point rounding error analysis

✅ Formally verified and sound

🔀 Sound treatment of conditionals

🔍 Accurate

Now many tools are available for floating-point round-off error analysis



PRECiSA static analyzer

floating-point implementation

$$|\tilde{f} - f| \le e$$

error bound

real number specification

Floating-point program

PRECiSA

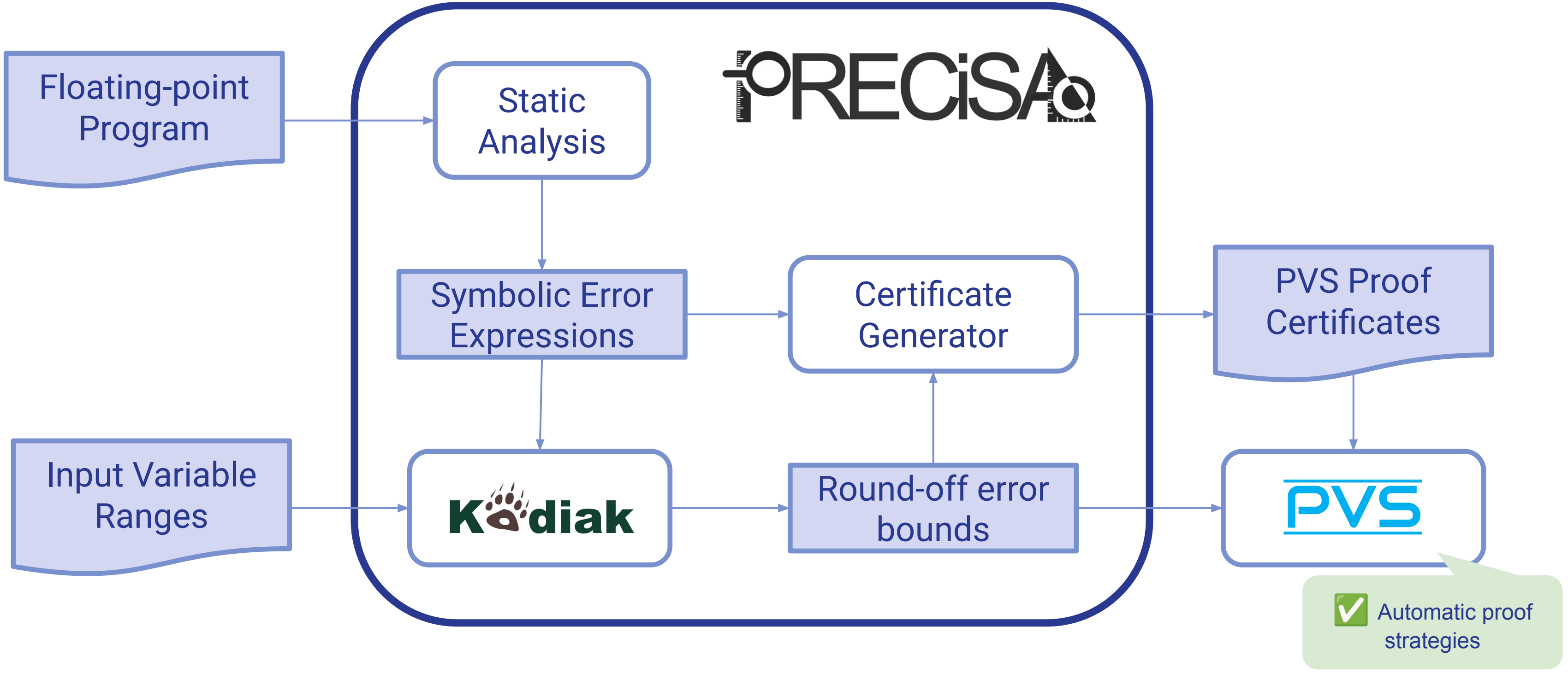
round-off error bounds
+ PVS proof certificates

Moscato M., Titolo L., Dutle A., Muñoz C. Automatic Estimation of Verified Floating-Point Round-Off Errors via Static Analysis. SAFECOMP 2017.

Titolo L., Feliú M., Moscato M., Muñoz C. *An Abstract Interpretation Framework for the Round-Off Error Analysis of Floating-Point Programs*. VMCAI 2018.

Titolo L., Moscato M., Feliú M., Masci P., Muñoz C. *Rigorous Floating-Point Round-Off Error Analysis in PRECiSA 4.0*. FM 2024.

PRECiSA Workflow



PRECiSA - Step I: Static Analysis

For each function declaration PRECiSA computes a set of **conditional error bounds**

condition $\mathbb{R}$

$$\langle \phi, \tilde{\phi} \rangle_t \twoheadrightarrow (r, \mathrm{e})^\pi$$

condition $\mathbb{F}$

stable/unstable

symbolic real value

symbolic round-off error

Everything is **symbolic**! → Compositional analysis

PRECiSA - Step I: Static Analysis

For each function declaration PRECiSA computes a set of **conditional error bounds**

condition $\mathbb{R}$

$$\langle \phi, \tilde{\phi} \rangle_t \twoheadrightarrow (r, \mathrm{e})^{\pi}$$

condition $\mathbb{F}$

stable/unstable

symbolic real value

symbolic round-off error

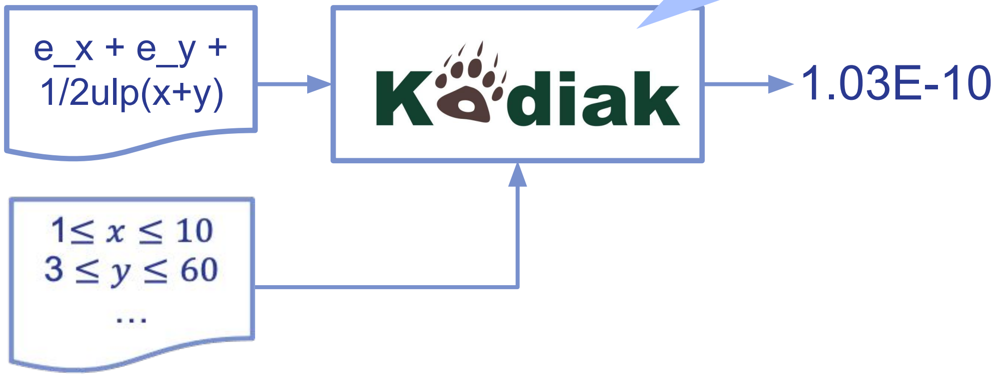
The analysis has to **soundly** take into account the conditional instability

!! Abstractions are used to mitigate the state explosion problem

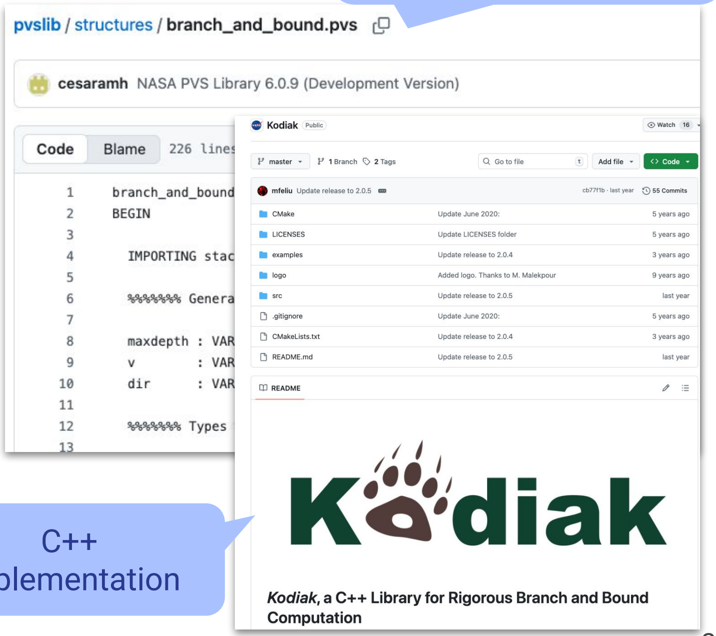
Everything is **symbolic**! ➡ Compositional analysis

PRECiSA - Step II: Global Optimization

Maximization via branch-and-bound parametric to **max depth** and **precision**



PVS formalization of the **branch-and-bound** algorithm



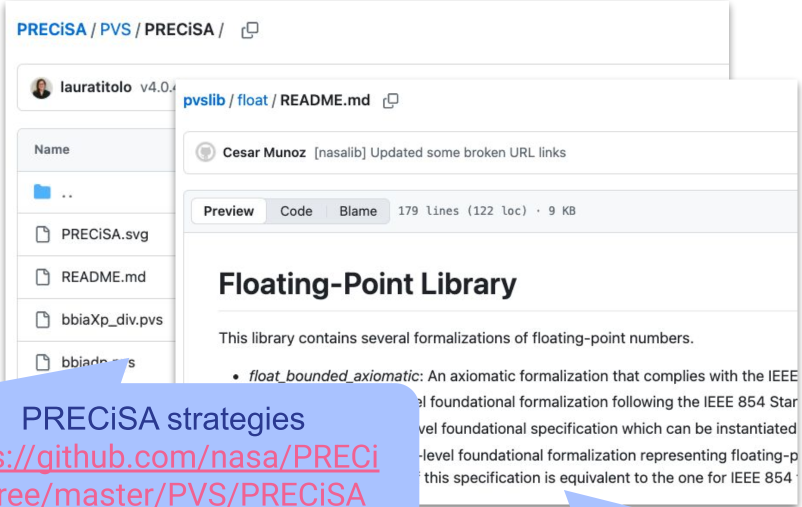
C++ implementation

PRECiSA Step III: Proof certificate generation

```
mul_aerr_bound(r1,e1,r2,e2)
: nonneg_real
= abs(r1)*e2+abs(r2)*e1+e1*e2

% Error of the (correctly-rounded) function w.r.t. the real operation on
% the (real version of the) floating point numbers.
prove | discharge-tccs | status-proofchain | show-prooflite
mul_accum_err: LEMMA
  abs(FtoR(f1)-r1) <= e1 AND
  abs(FtoR(f2)-r2) <= e2
  IMPLIES abs((FtoR(f1)*FtoR(f2))-(r1*r2)) <= mul_aerr_bound(r1,e1,r2,e2)

mul_ulp_bound(r1,e1,r2,e2)
: real
= abs(r1)*abs(r2) + abs(r1)*e2 + e1*abs(r2) + e1*e2

prove | discharge-tccs | status-proofchain | show-prooflite
Fmul_accum_err_bound: LEMMA
  abs(FtoR(f1)-r1) <= e1 AND
  abs(FtoR(f2)-r2) <= e2
  IMPLIES abs(FtoR(f1)*FtoR(f2)) <= mul_ulp_bound(r1,e1,r2,e2)

prove | discharge-tccs | status-proofchain | show-prooflite
accum_err_bound: LEMMA
  abs(FtoR(f1)-r1) <= e1 AND
  abs(FtoR(f2)-r2) <= e2
  IMPLIES abs(FtoR(Fmul(f1,f2)) - (r1*r2))
          <= mul_aerr_bound(r1,e1,r2,e2)
        + ulp(b)(mul_ulp_bound(r1,e1,r2,e2)) / 2
```

Accumulated round-off error

Rounding of the result



PRECiSA strategies
https://github.com/nasa/PRECiSA/tree/master/PVS/PRECiSA

IEEE 745 and 854 + round-off error formalization
https://github.com/nasa/pvslib/blob/master/float

PRECiSA Step III: Proof certificate generation

Users > ltitolo > Desktop > precisa > public > benchmarks > code-generation > daidalus > tcoa > tcoa_num_cert.pvs > tcoa_num_ce

```
% This file is automatically generated by PRECiSA

% maxDepth: 7 , prec: 10^-14

typecheck-file | evaluate-in-pvsio
tcoa_num_cert: THEORY
BEGIN
IMPORTING cert_tcoa, PRECiSA@bbiasp, PRECiSA@bbiadp, PRECiSA@strategies

%|- *_TCC* : PROOF
%|- (precisa-gen-cert-tcc)
%|- QED

% Floating-Point Results: 0, neg_double(div_double(sz, vz))
% Real Results: -((r_sz / r_vz)), 0
% Control Flow:  Stable
prove | status-proofchain | show-prooflite
tcoa_fp_c_0 : LEMMA
FORALL(r_sz, r_vz: real, sz: double, vz: double):
abs(safe_prjct_double(sz) - r_sz)<=ulp_dp(r_sz)/2 AND abs(safe_prjct_double(vz) - r_vz)<=ulp_dp(r_vz)/2
AND ((((r_vz /= 0) AND ((r_sz * r_vz) < 0)) AND ((vz /= integerToDouble(0)) AND (mul_double(sz, vz) <
integerToDouble(0)))) OR (NOT(((r_sz * r_vz) < 0)) AND NOT((mul_double(sz, vz) < integerToDouble(0)))))
AND r_sz ## [|1,1000|] AND  r_vz ## [|1,1000|] AND
finite_double?(tcoa_fp(sz, vz)) AND finite_double?(sz) AND finite_double?(vz) AND finite_double?(mul_double
(sz, vz)) AND finite_double?(integerToDouble(0))
IMPLIES
abs(safe_prjct_double(tcoa_fp(sz, vz)) - tcoa(r_sz, r_vz))<=6876345720111303 / 24758800785707605497982484480

%|- tcoa_fp_c_0 : PROOF
%|- (prove-concrete-lemma tcoa_fp_0 14 7)
%|- QED
```

Initial ranges

Rounding error

Automatic strategy

VSCode-PRECiSA: Round-off Error Analysis GUI

nl_comp.pvs > nl_comp

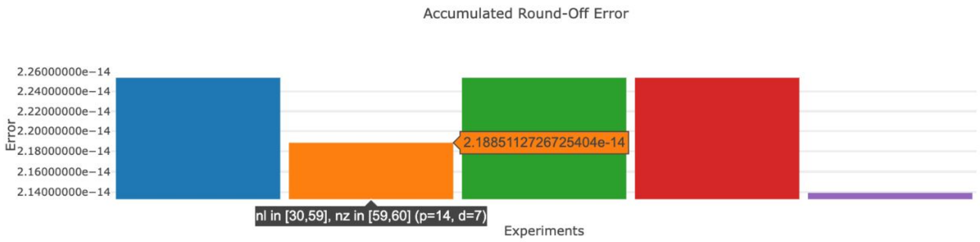
typecheck-file | evaluate-in-pvsio

```
nl_comp: THEORY
BEGIN
IMPORTING float@double64

% @fp-function
% @fp-range nl in [2,59], nz in [59,60]
  estimate-error-bounds | compare-error-bounds
  nl_comp(nl,nz: double): double =
    (180/3.14) * acos(sqrt((1-cos(3.14/(2*nz)))
                  /(1-cos(2*3.14/nl))))

END nl_comp
```

INPUT	MIN	MAX
nl	50	59
nz	60	60

Analyze

Save Results



Experiment	Accumulated Round-Off Error
nl in [2,59], nz in [59,60] (p=14, d=7)	2.2540032743878307e-14
nl in [30,59], nz in [59,60] (p=14, d=7)	2.1885112726725404e-14
nl in [2,30], nz in [59,60] (p=14, d=7)	2.2540013434075158e-14
nl in [2,5], nz in [60,60] (p=14, d=7)	2.2540007151689644e-14
nl in [50,59], nz in [60,60] (p=14, d=7)	2.139337646609609e-14

- Interval Analysis
- Sensitivity Analysis
- Comparative Analysis

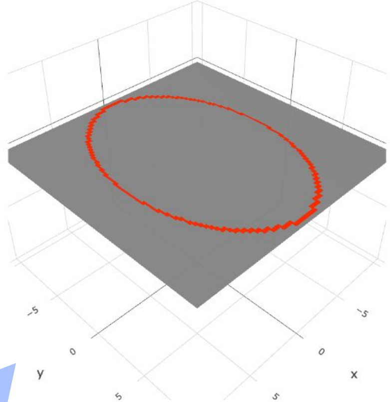
VSCode-PRECiSA: Instability Analysis GUI

ellipse.pvs > ...

```
typecheck-file | evaluate-in-pvsio
ellipse: THEORY
BEGIN
IMPORTING float@aerr754dp

% @fp-function
% @fp-range x in [1,300], y in [1,300]
estimate-error-bounds | compare-error-bounds
pointInEllipse(x,y: double): double =
    IF x*x/4 + y*y/9 <= 10 THEN 1 ELSE -1 ENDIF

END ellipse
```

INPUT	MIN	MAX
x	-10	10
y	-10	10

Analyze



INPUT	MIN	MAX
x	0	5
y	0	5

Analyze

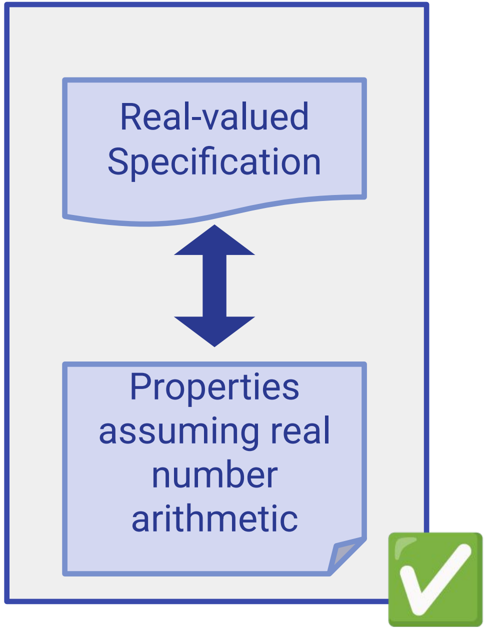
Paving results indicate all Ok!

— This means that the function does not contain unstable guards

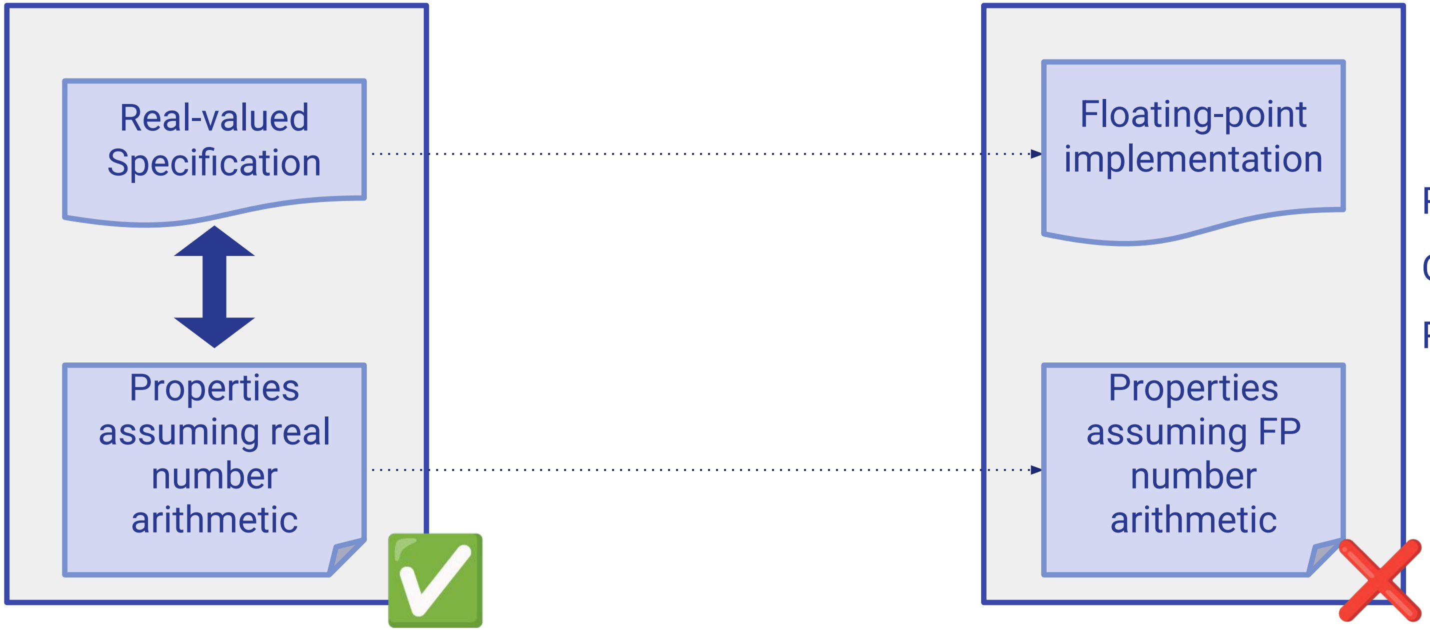
Visualize the values that may cause divergences in the control flow using Kodiak's paving functionality

Check that no conditional instability occurs

Real number ≠ Floating-point arithmetic



Real number ≠ Floating-point arithmetic

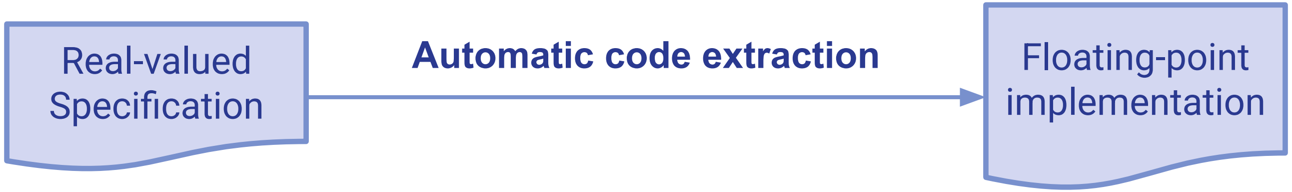


Rounding errors

Overflows

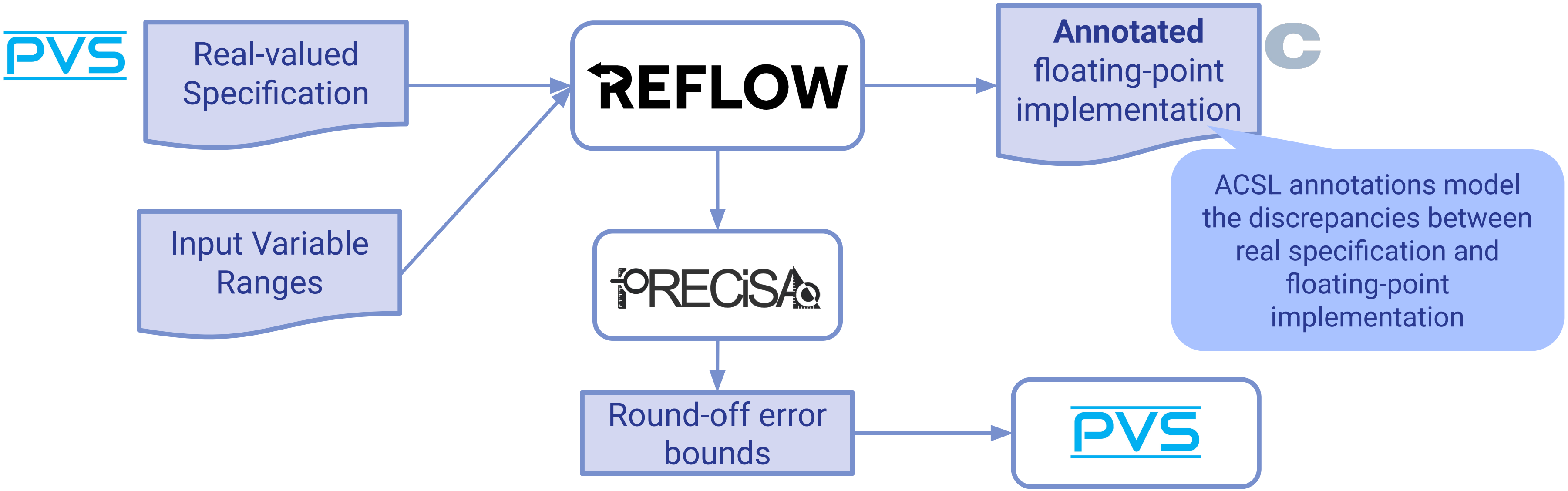
Runtime exceptions

Idea: automatically extracting FP code with formal guarantees on the rounding error



ReFlow automatically generates a C floating-point implementation from a PVS real number specification

ReFlow workflow



Example: eps_line code extraction

```
eps_line(sx,vx,sy,vy: real): real = ((sx*vx) + (sy*vy)) * ((sx*vx) - (sy*vy))
```

```
/*@
logic real eps_line (real sx, real vx, real sy, real vy) =
(((sx * vx) + (sy * vy)) * ((sx * vx) - (sy * vy)));

ensures \forall real sx, real vx, real sy, real vy;
-60000 <= sx && sx <= 60000 && -60000 <= vx && vx <= 60000 &&
-60000 <= sy && sy <= 60000 && -60000 <= vy && vy <= 60000 &&
\abs(sx_d - sx) <= ulp_dp(sx)/2 && \abs(vx_d - vx) <= ulp_dp(vx)/2 &&
\abs(sy_d - sy) <= ulp_dp(sy)/2 && \abs(vy_d - vy) <= ulp_dp(vy)/2
==> \abs((\result - eps_line(sx, vx, sy, vy)) <= 0x1.db070f4580002p14 ;
*/
double eps_line_fp (double sx_d, double vx_d, double sy_d, double vy_d) {
  return ((sx_d * vx_d) + (sy_d * vy_d)) * ((sx_d * vx_d) - (sy_d * vy_d));
}
```

RTCA/FAA Minimum Operational Performance Standards (MOPS) DO365 for detect and avoid of unmanned aircraft systems (UAS)

DAIDALUS

Example: eps_line code extraction

```
eps_line(sx,vx,sy,vy: real): real = ((sx*vx) + (sy*vy)) * ((sx*vx) - (sy*vy))
```

```
/*@
logic real eps_line (real sx, real vx, real sy, real vy) =
(((sx * vx) + (sy * vy)) * ((sx * vx) - (sy * vy)));

ensures \forall real sx, real vx, real sy, real vy;
-60000 <= sx && sx <= 60000 && -60000 <= vx && vx <= 60000 &&
-60000 <= sy && sy <= 60000 && -60000 <= vy && vy <= 60000 &&
\abs(sx_d - sx) <= ulp_dp(sx)/2 && \abs(vx_d - vx) <= ulp_dp(vx)/2 &&
\abs(sy_d - sy) <= ulp_dp(sy)/2 && \abs(vy_d - vy) <= ulp_dp(vy)/2
==> \abs((\result - eps_line(sx, vx, sy, vy)) <= 0x1.db070f4580002p14 ;
*/
double eps_line_fp (double sx_d, double vx_d, double sy_d, double vy_d) {
  return ((sx_d * vx_d) + (sy_d * vy_d)) * ((sx_d * vx_d) - (sy_d * vy_d));
}
```

Axiomatic real-valued program

Example: eps_line code extraction

```
eps_line(sx,vx,sy,vy: real): real = ((sx*vx) + (sy*vy)) * ((sx*vx) - (sy*vy))
```

```
/*@
logic real eps_line (real sx, real vx, real sy, real vy) =
(((sx * vx) + (sy * vy)) * ((sx * vx) - (sy * vy)));

ensures \forall real sx, real vx, real sy, real vy;
-60000 <= sx && sx <= 60000 && -60000 <= vx && vx <= 60000 &&
-60000 <= sy && sy <= 60000 && -60000 <= vy && vy <= 60000 &&
\abs(sx_d - sx) <= ulp_dp(sx)/2 && \abs(vx_d - vx) <= ulp_dp(vx)/2 &&
\abs(sy_d - sy) <= ulp_dp(sy)/2 && \abs(vy_d - vy) <= ulp_dp(vy)/2
==> \abs((\result - eps_line(sx, vx, sy, vy)) <= 0x1.db070f4580002p14 ;
*/
double eps_line_fp (double sx_d, double vx_d, double sy_d, double vy_d) {
  return ((sx_d * vx_d) + (sy_d * vy_d)) * ((sx_d * vx_d) - (sy_d * vy_d));
}
```

Axiomatic real-valued program

Floating-point implementation

Example: eps_line code extraction

```
eps_line(sx,vx,sy,vy: real): real = ((sx*vx) + (sy*vy)) * ((sx*vx) - (sy*vy))
```

```
/*@
logic real eps_line (real sx, real vx, real sy, real vy) =
(((sx * vx) + (sy * vy)) * ((sx * vx) - (sy * vy)));

ensures \forall real sx, real vx, real sy, real vy;
-60000 <= sx && sx <= 60000 && -60000 <= vx && vx <= 60000 &&
-60000 <= sy && sy <= 60000 && -60000 <= vy && vy <= 60000 &&
\abs(sx_d - sx) <= ulp_dp(sx)/2 && \abs(vx_d - vx) <= ulp_dp(vx)/2 &&
\abs(sy_d - sy) <= ulp_dp(sy)/2 && \abs(vy_d - vy) <= ulp_dp(vy)/2
==> \abs((\result - eps_line(sx, vx, sy, vy)) <= 0x1.db070f4580002p14 ;
*/
double eps_line_fp (double sx_d, double vx_d, double sy_d, double vy_d) {
  return ((sx_d * vx_d) + (sy_d * vy_d)) * ((sx_d * vx_d) - (sy_d * vy_d));
}
```

Axiomatic real-valued program

Initial ranges

Floating-point implementation

Example: eps_line code extraction

```
eps_line(sx,vx,sy,vy: real): real = ((sx*vx) + (sy*vy)) * ((sx*vx) - (sy*vy))
```

```
/*@
logic real eps_line (real sx, real vx, real sy, real vy) =
(((sx * vx) + (sy * vy)) * ((sx * vx) - (sy * vy)));

ensures \forall real sx, real vx, real sy, real vy;
-60000 <= sx && sx <= 60000 && -60000 <= vx && vx <= 60000 &&
-60000 <= sy && sy <= 60000 && -60000 <= vy && vy <= 60000 &&
\abs(sx_d - sx) <= ulp_dp(sx)/2 && \abs(vx_d - vx) <= ulp_dp(vx)/2 &&
\abs(sy_d - sy) <= ulp_dp(sy)/2 && \abs(vy_d - vy) <= ulp_dp(vy)/2
==> \abs((\result - eps_line(sx, vx, sy, vy)) <= 0x1.db070f4580002p14 ;
*/
double eps_line_fp (double sx_d, double vx_d, double sy_d, double vy_d) {
  return ((sx_d * vx_d) + (sy_d * vy_d)) * ((sx_d * vx_d) - (sy_d * vy_d));
}
```

Axiomatic real-valued program

Initial ranges

Vars are round-to the nearest

Floating-point implementation

Example: eps_line code extraction

```
eps_line(sx,vx,sy,vy: real): real = ((sx*vx) + (sy*vy)) * ((sx*vx) - (sy*vy))
```

```
/*@
logic real eps_line (real sx, real vx, real sy, real vy) =
(((sx * vx) + (sy * vy)) * ((sx * vx) - (sy * vy)));

ensures \forall real sx, real vx, real sy, real vy;
-60000 <= sx && sx <= 60000 && -60000 <= vx && vx <= 60000 &&
-60000 <= sy && sy <= 60000 && -60000 <= vy && vy <= 60000 &&
\abs(sx_d - sx) <= ulp_dp(sx)/2 && \abs(vx_d - vx) <= ulp_dp(vx)/2 &&
\abs(sy_d - sy) <= ulp_dp(sy)/2 && \abs(vy_d - vy) <= ulp_dp(vy)/2
==> \abs((\result - eps_line(sx, vx, sy, vy)) <= 0x1.db070f4580002p14 ;
*/
double eps_line_fp (double sx_d, double vx_d, double sy_d, double vy_d) {
  return ((sx_d * vx_d) + (sy_d * vy_d)) * ((sx_d * vx_d) - (sy_d * vy_d));
}
```

Axiomatic real-valued program

Initial ranges

Vars are round-to the nearest

Maximum round-off error

Floating-point implementation

Example: Unstable conditional instrumentation

```
tcoa(sz,vz:real): real = IF (sz*vz < 0) THEN -(sz/vz) ELSE -1 ENDIF
```

```
/*@
requires (0 <= E) ;
ensures \forall real sz, real vz; ((\abs(Dmul(sz_dp, vz_dp) - (sz * vz)) <= E
  ==> \result.isValid ==> (sz * vz < 0 && Dmul(sz_dp, vz_dp) < 0)
                          || (sz * vz >= 0 && Dmul(sz_dp, vz_dp) >= 0) ;
*/
struct maybeDouble tcoa_fp (double sz_dp, double vz_dp, double E) {
  if (sz_dp * vz_dp < - E)
  { return someDouble(-sz_dp / vz_dp);
  } else { if (sz_dp * vz_dp >= E)
           { return someDouble(-1);
           } else { return instability_warning();}}
}
/*@
ensures \forall real sz, real vz; (0 <= sz) && (sz <= 1000) && (400 <= vz) && (vz <= 600) &&
\abs(sz_dp - sz) <= ulp_dp(sz)/2 && \abs(vz_dp - vz) <= ulp_dp(vz)/2) &&
\result.isValid ==> \abs(\result.value - tcoa(sz, vz)) <= 0x1.c7ae147ae147dp-51;
*/
struct maybeDouble tcoa_num (double sz_dp, double vz_dp) {
  return tcoa_fp (sz_dp, vz_dp, 0x1.4800000000001p-33);
}
```

DAIDALUS

Example: Unstable conditional instrumentation

```
tcoa(sz,vz:real): real = IF (sz*vz < 0) THEN -(sz/vz) ELSE -1 ENDIF
```

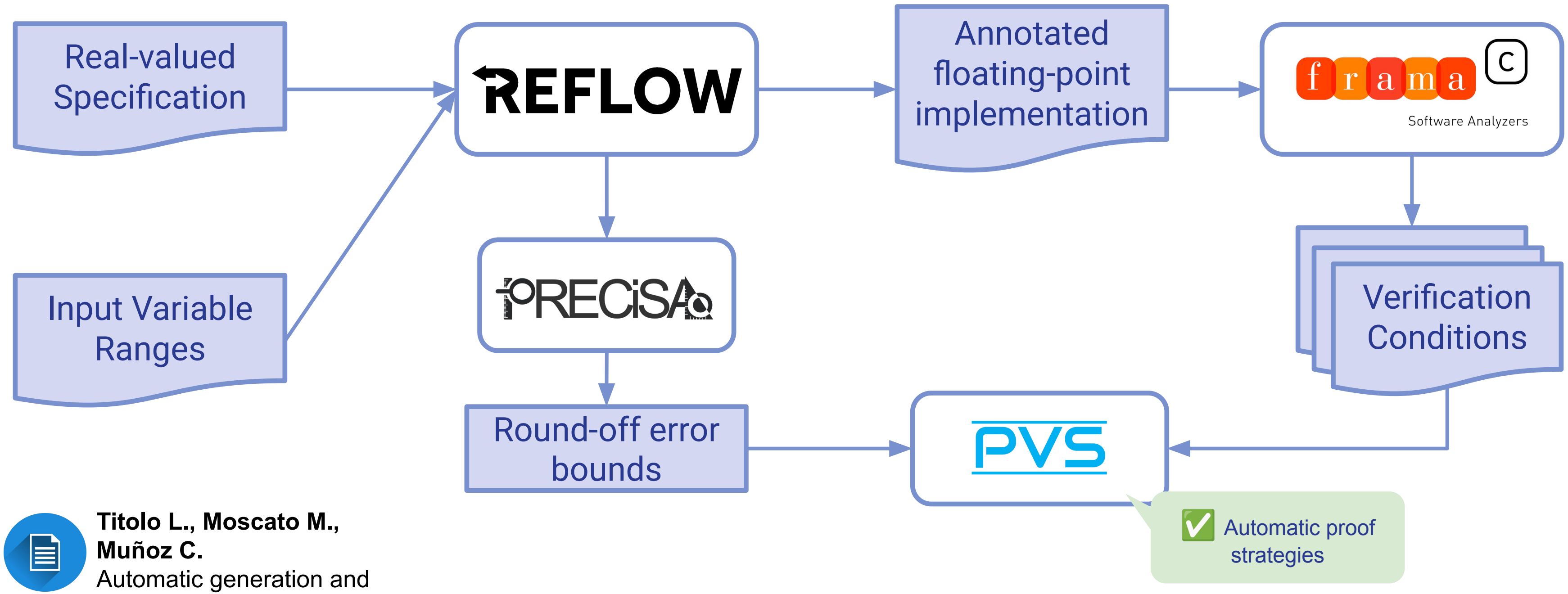
```
/*@
requires (0 <= E) ;
ensures \forall real sz, real vz; ((\abs(Dmul(sz_dp, vz_dp) - (sz * vz)) <= E
  ==> \result.isValid ==> (sz * vz < 0 && Dmul(sz_dp, vz_dp) < 0)
                          || (sz * vz >= 0 && Dmul(sz_dp, vz_dp) >= 0) ;
*/
struct maybeDouble tcoa_fp (double sz_dp, double vz_dp, double E) {
  if (sz_dp * vz_dp < - E)
  { return someDouble(-sz_dp / vz_dp);
  } else { if (sz_dp * vz_dp >= E)
           { return someDouble(-1);
           } else { return instability_warning();}}
}
/*@
ensures \forall real sz, real vz; (0 <= sz) && (sz <= 1000) && (400 <= vz) && (vz <= 600) &&
\abs(sz_dp - sz) <= ulp_dp(sz)/2 && \abs(vz_dp - vz) <= ulp_dp(vz)/2) &&
\result.isValid ==> \abs(\result.value - tcoa(sz, vz)) <= 0x1.c7ae147ae147dp-51;
*/
struct maybeDouble tcoa_num (double sz_dp, double vz_dp) {
  return tcoa_fp (sz_dp, vz_dp, 0x1.4800000000001p-33);
}
```

Code instrumentation to detect **unstable conditionals** (real ≠ fp control flow)

Instrumented program

Titolo L., Muñoz C., Feliú M., Moscato M., Eliminating unstable tests in floating-point programs. LOPSTR 2018.

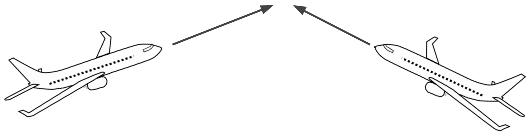
C code automatically verified in Frama-C and PVS



Titolo L., Moscato M., Muñoz C.
Automatic generation and verification of test-stable floating-point code. iFM 2020.

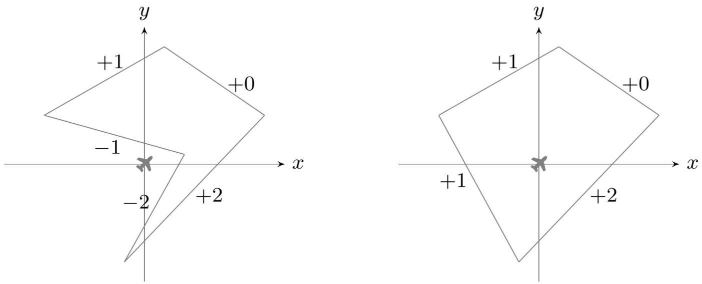
Applications of REFLOW

DAIDALUS - Detect-and-avoid



Bernardes N., Moscato M., Titolo L., Ayala M.
A provably correct floating-point implementation of Well Clear Avionics Concepts. FMCAD 2023

POLYCARP - Geofencing

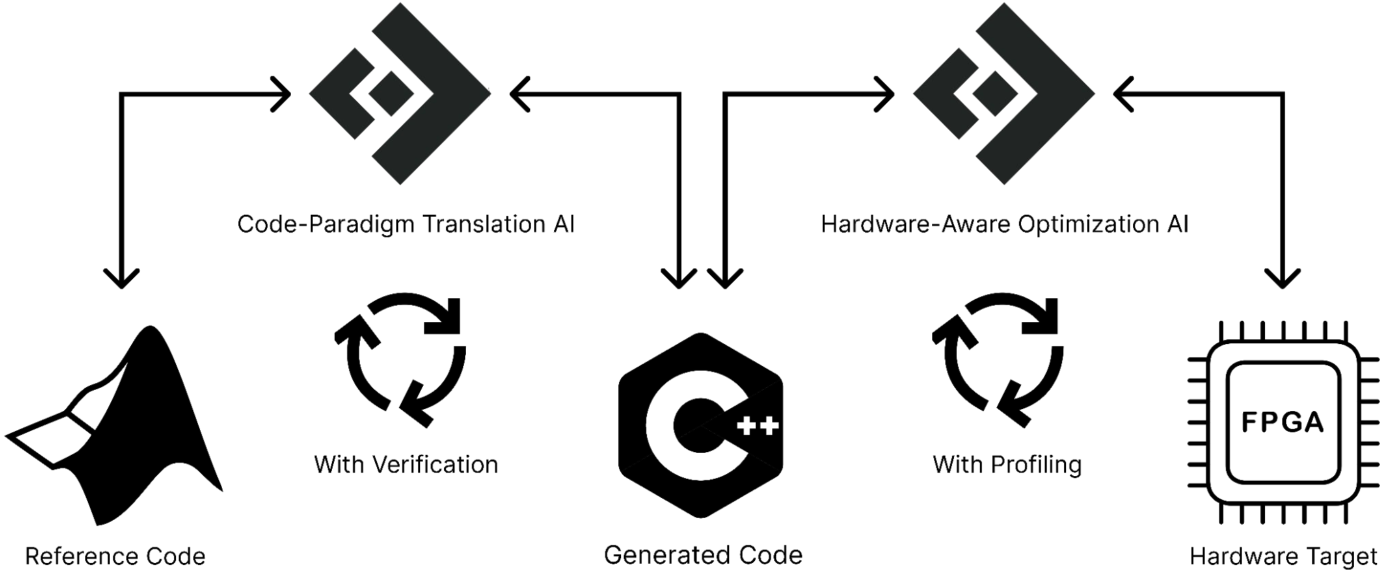


Moscato M., Titolo L., Feliú M., Muñoz C.
Provably Correct Floating-Point Implementation of a Point-in-Polygon Algorithm. FM 2019

Interactive Theorem Proving at NASA

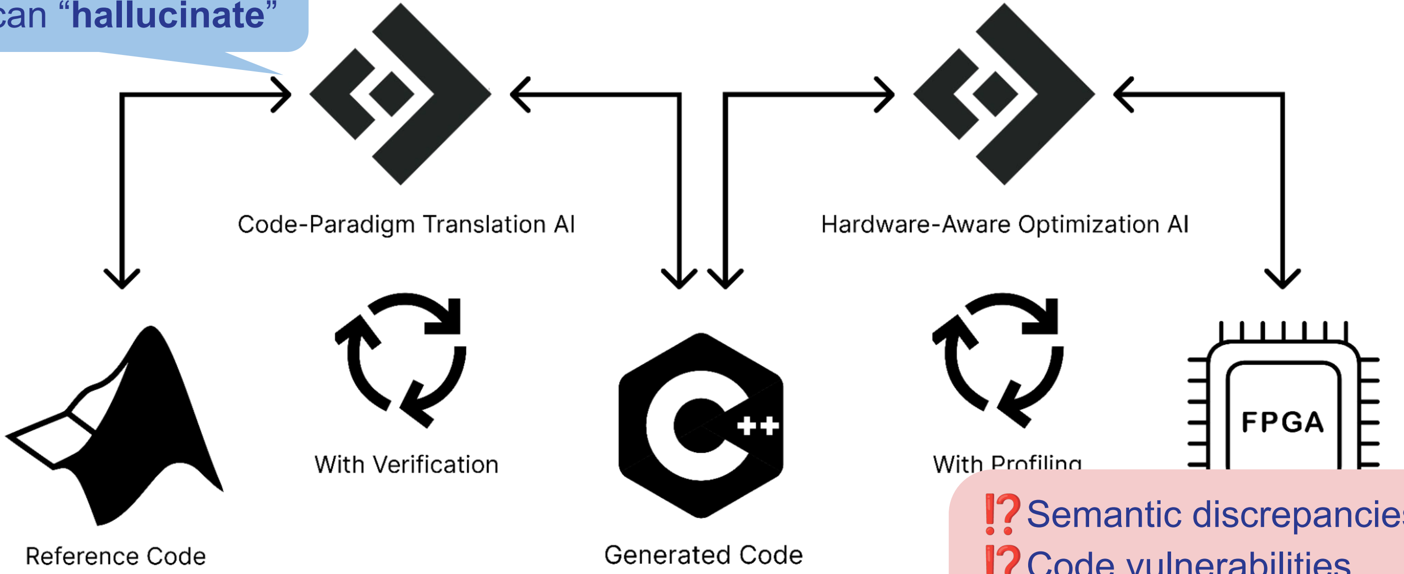
- Static Analysis
- Global optimization
- Hybrid Systems Verification (dDL)
- Requirements elicitation (FRET)
- Some applications:
    - Unmanned Aerial Vehicles
    - Detect-and-avoid
    - Mars Rovers (PLEXIL V)

Proofs for LLM-guided transpilation in industry



Proofs for LLM-guided transpilation in industry

❗ LLMs are **Black Boxes**!
🌀 LLMs can “**hallucinate**”

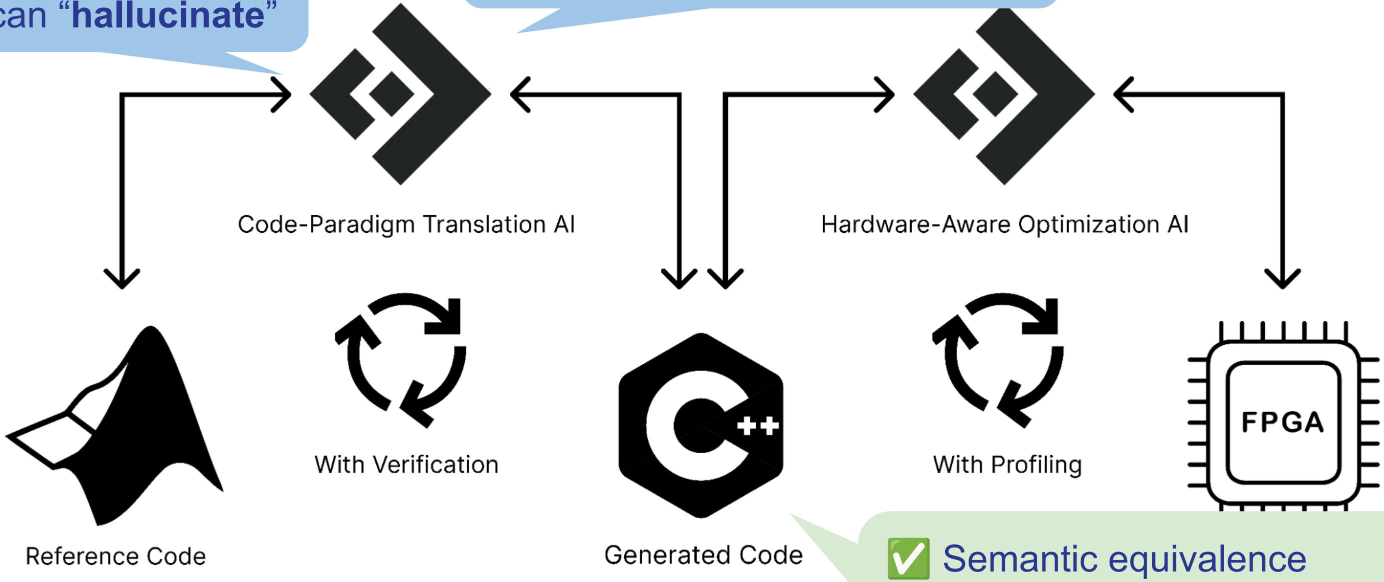


⁉️ Semantic discrepancies
⁉️ Code vulnerabilities
⁉️ Specification violation

Proofs for LLM-guided transpilation in industry

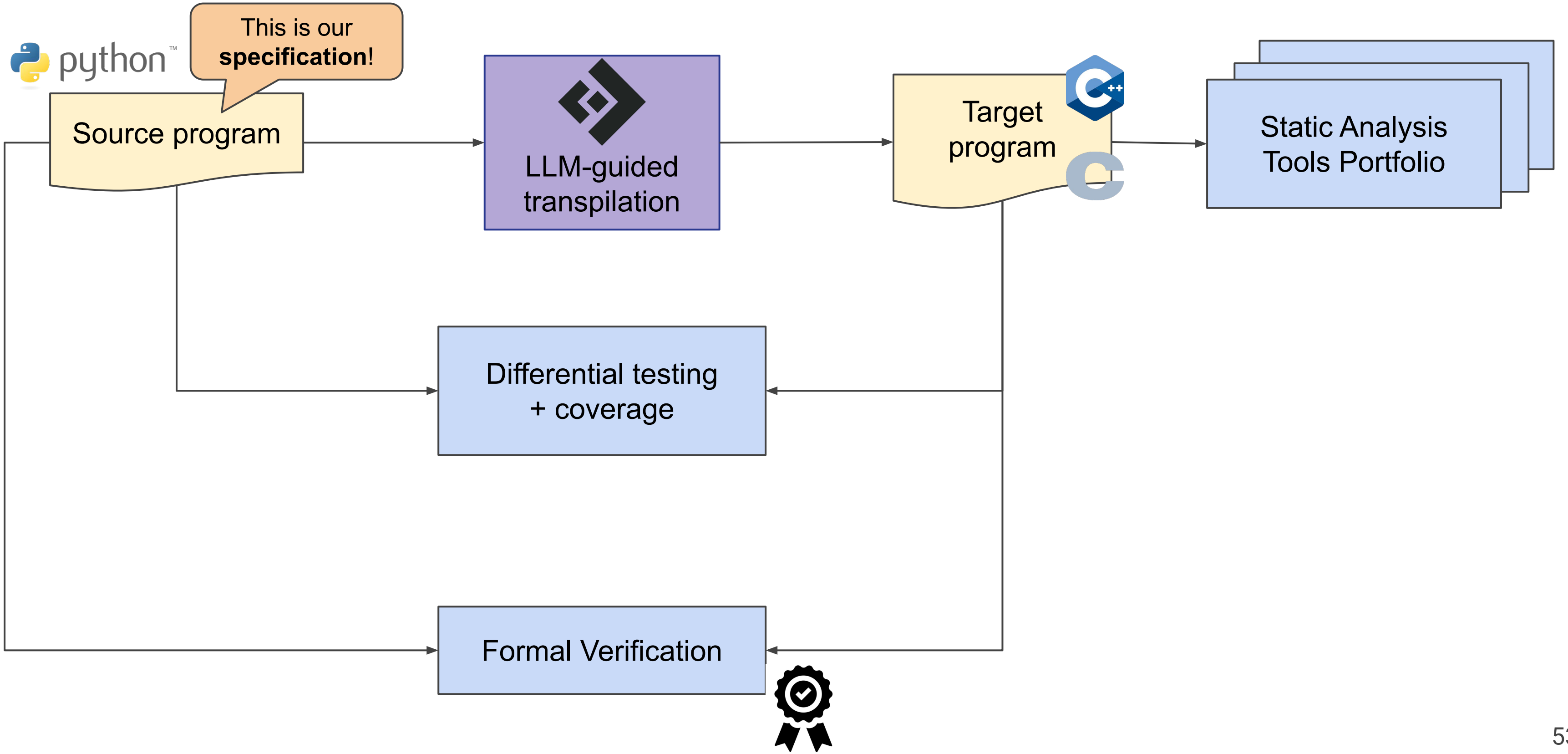
❗ LLMs are **Black Boxes**!
🌀 LLMs can "**hallucinate**"

💻✅ LLM generated code must be validated and verified

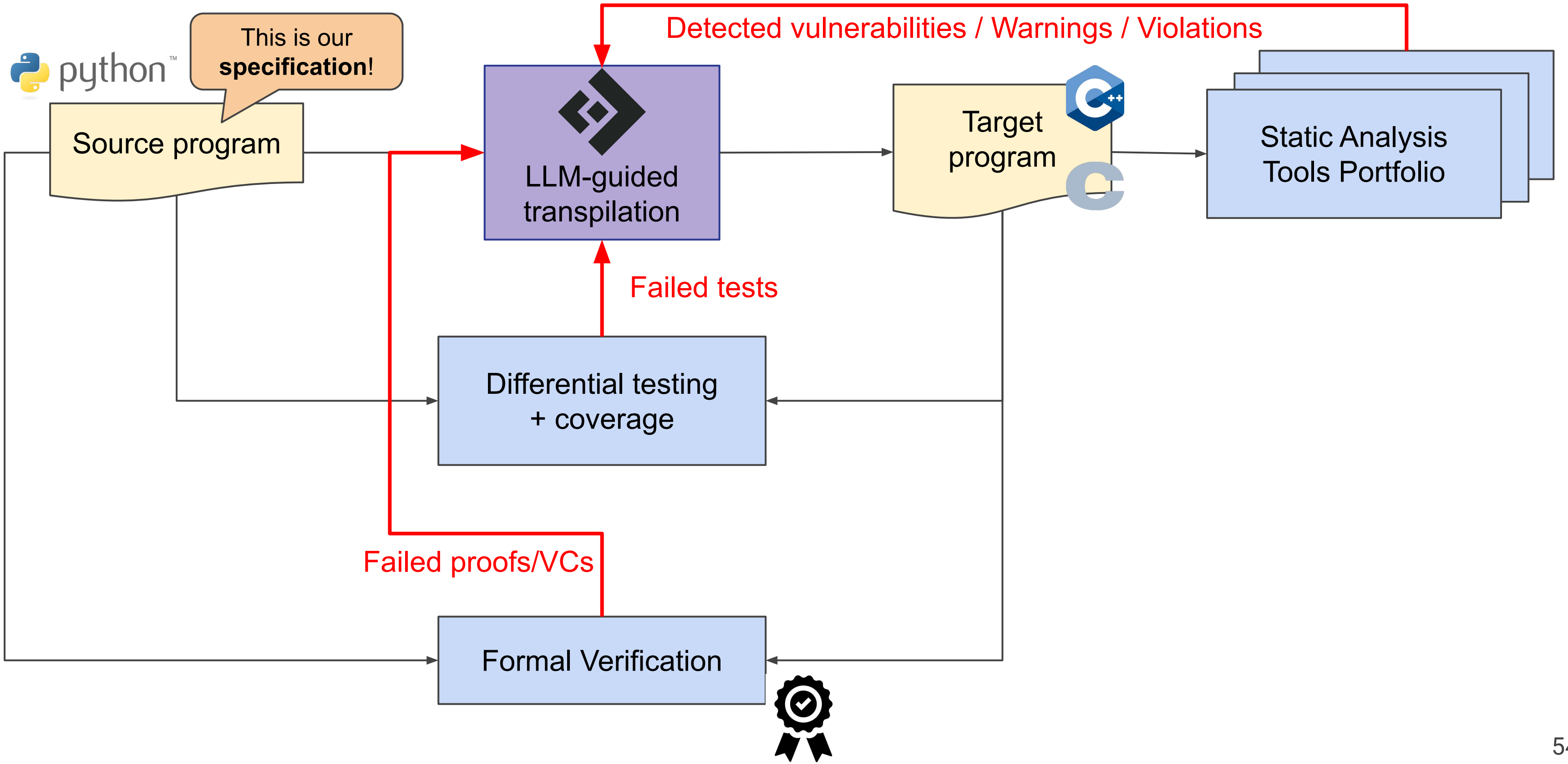


✅ Semantic equivalence
🛡️ No code vulnerabilities
📘 Standard compliance (ex MISRA)

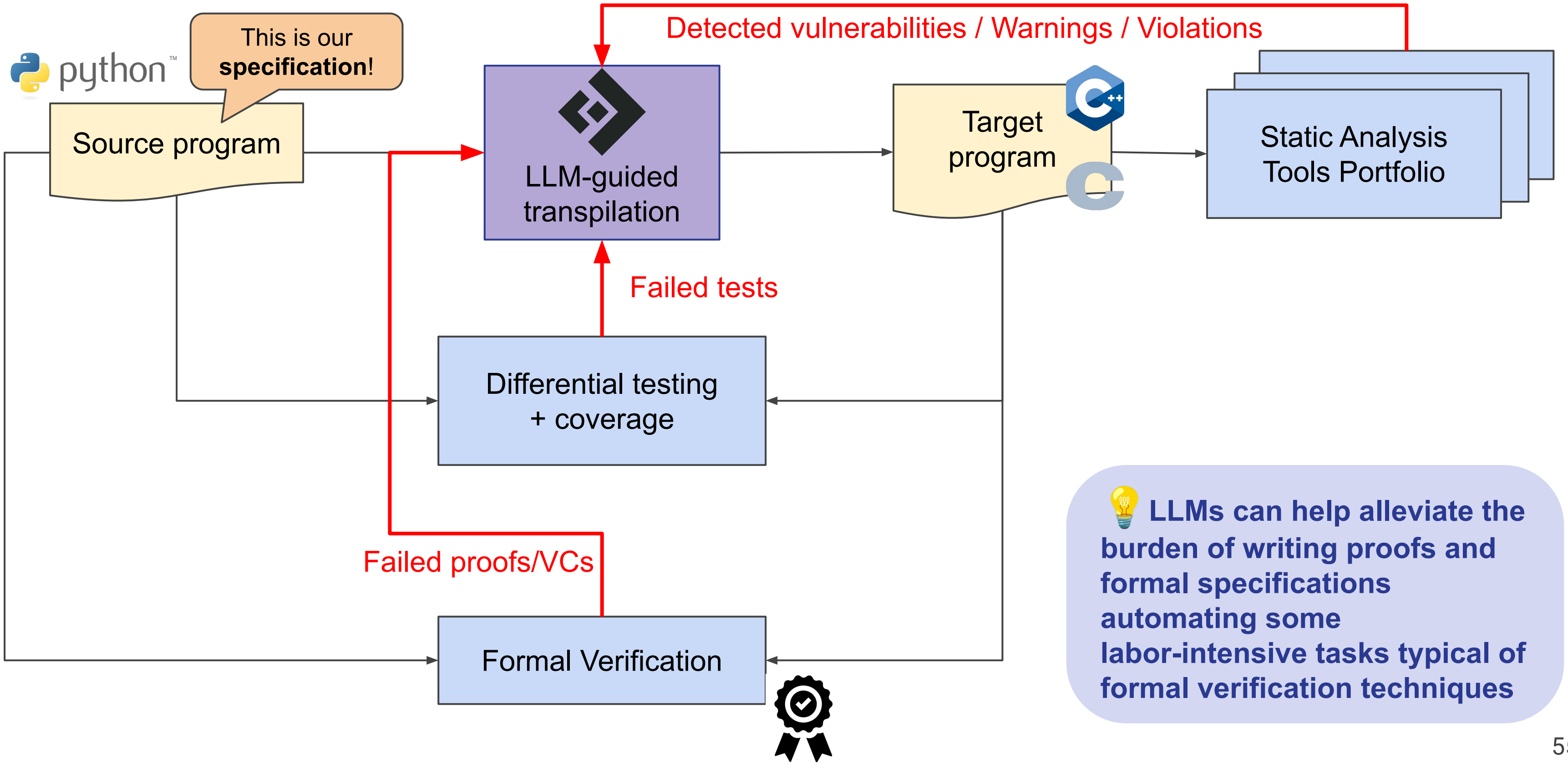
LLM-generated code must be validated and verified!



LLM-generated code must be validated and verified!



LLM-generated code must be validated and verified!



💡 **LLMs can help alleviate the burden of writing proofs and formal specifications automating some labor-intensive tasks typical of formal verification techniques**

Conclusion

🚀 Interactive theorem provers are heavily used at NASA for safety critical applications

💻 ITP for AI - LLM-guided code generation greatly benefits from formal proofs

🤖AI for ITP

- 🔄 Proof translation for importing/exporting libraries across different ITP platforms
- 🛠️ Proof repair and simplification
- 🪄 Proof generation/automation

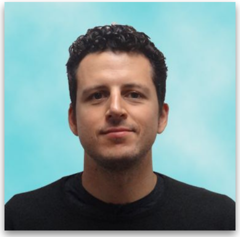
📝🌟 Exciting research and topics to explores at the intersection of ITP and AI

Thanks for your attention!

Aaron Dutle
NASA

Cesar Muñoz
NASA

Mariano Moscato
NASA/AMA



Marco A. Feliu
NASA/AMA

Paolo Masci
x-NASA

CODEMETAL

laura@codemetal.ai
https://lauratitolo.github.io/

Backup slides

Example: Unstable conditional instrumentation

```
tcoa(sz,vz:real): real = IF (sz*vz < 0) THEN -(sz/vz) ELSE -1 ENDIF
```

```
/*@
requires (0 <= E) ;
ensures \forall real sz, real vz; ((\abs(Dmul(sz_dp, vz_dp) - (sz * vz)) <= E
  ==> \result.isValid ==> (sz * vz < 0 && Dmul(sz_dp, vz_dp) < 0)
                          || (sz * vz >= 0 && Dmul(sz_dp, vz_dp) >= 0) ;
*/
struct maybeDouble tcoa_fp (double sz_dp, double vz_dp, double E) {
  if (sz_dp * vz_dp < - E)
  { return someDouble(-sz_dp / vz_dp);
  } else { if (sz_dp * vz_dp >= E)
         { return someDouble(-1);
         } else { return instability_warning();}}
}
/*@
ensures \forall real sz, real vz; (0 <= sz) && (sz <= 1000) && (400 <= vz) && (vz <= 600) &&
\abs(sz_dp - sz) <= ulp_dp(sz)/2 && \abs(vz_dp - vz) <= ulp_dp(vz)/2) &&
\result.isValid ==> \abs(\result.value - tcoa(sz, vz)) <= 0x1.c7ae147ae147dp-51;
*/
struct maybeDouble tcoa_num (double sz_dp, double vz_dp) {
  return tcoa_fp (sz_dp, vz_dp, 0x1.4800000000001p-33);
}
```

ACSL contract

Instrumented program

symbolic

Example: Unstable conditional instrumentation

```
tcoa(sz,vz:real): real = IF (sz*vz < 0) THEN -(sz/vz) ELSE -1 ENDIF
```

```
/*@
requires (0 <= E) ;
ensures \forall real sz, real vz; ((\abs(Dmul(sz_dp, vz_dp) - (sz * vz)) <= E
  ==> \result.isValid ==> (sz * vz < 0 && Dmul(sz_dp, vz_dp) < 0)
                         || (sz * vz >= 0 && Dmul(sz_dp, vz_dp) >= 0) ;
*/
struct maybeDouble tcoa_fp (double sz_dp, double vz_dp, double E) {
  if (sz_dp * vz_dp < - E)
  { return someDouble(-sz_dp / vz_dp);
  } else { if (sz_dp * vz_dp >= E)
           { return someDouble(-1);
           } else { return instability_warning();}}
}
/*@
ensures \forall real sz, real vz; (0 <= sz) && (sz <= 1000) && (400 <= vz) && (vz <= 600) &&
\abs(sz_dp - sz) <= ulp_dp(sz)/2 && \abs(vz_dp - vz) <= ulp_dp(vz)/2) &&
\result.isValid ==> \abs(\result.value - tcoa(sz, vz)) <= 0x1.c7ae147ae147dp-51;
*/
struct maybeDouble tcoa_num (double sz_dp, double vz_dp) {
  return tcoa_fp (sz_dp, vz_dp, 0x1.4800000000001p-33);
}
```

ACSL contract

instrumented program

symbolic

function call to instrumented tcoa_fp

Example: Unstable conditional instrumentation

```
tcoa(sz,vz:real): real = IF (sz*vz < 0) THEN -(sz/vz) ELSE -1 ENDIF
```

```
/*@
requires (0 <= E) ;
ensures \forall real sz, real vz; ((\abs(Dmul(sz_dp, vz_dp) - (sz * vz)) <= E
  ==> \result.isValid ==> (sz * vz < 0 && Dmul(sz_dp, vz_dp) < 0)
                          || (sz * vz >= 0 && Dmul(sz_dp, vz_dp) >= 0) ;
*/
struct maybeDouble tcoa_fp (double sz_dp, double vz_dp, double E) {
  if (sz_dp * vz_dp < - E)
  { return someDouble(-sz_dp / vz_dp);
  } else { if (sz_dp * vz_dp >= E)
           { return someDouble(-1);
           } else { return instability_warning();}}
}
/*@
ensures \forall real sz, real vz; (0 <= sz) && (sz <= 1000) && (400 <= vz) && (vz <= 600) &&
\abs(sz_dp - sz) <= ulp_dp(sz)/2 && \abs(vz_dp - vz) <= ulp_dp(vz)/2) &&
\result.isValid ==> \abs(\result.value - tcoa(sz, vz)) <= 0x1.c7ae147ae147dp-51;
*/
struct maybeDouble tcoa_num (double sz_dp, double vz_dp) {
  return tcoa_fp (sz_dp, vz_dp, 0x1.4800000000001p-33);
}
```

ACSL contract

Instrumented program

symbolic

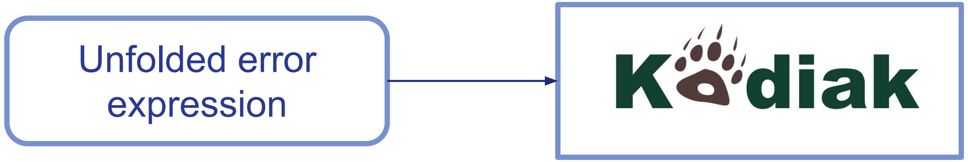
ACSL contract

function call to instrumented tcoa_fp

numeric

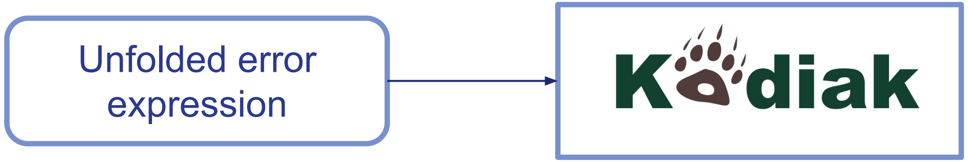
Function calls abstract analysis

Before:

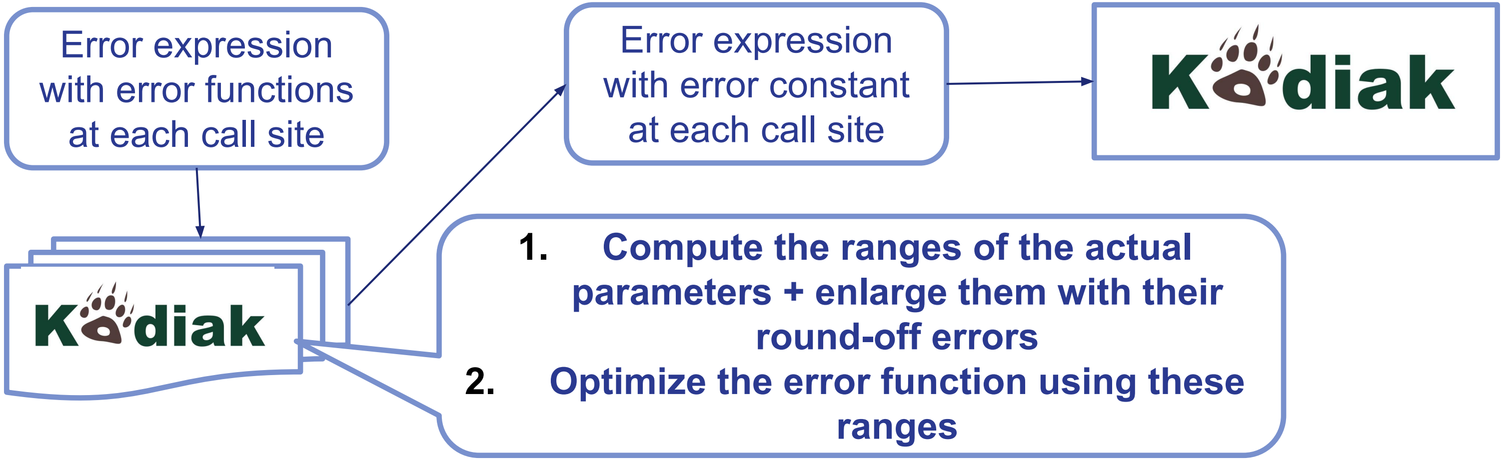


Function calls abstract analysis

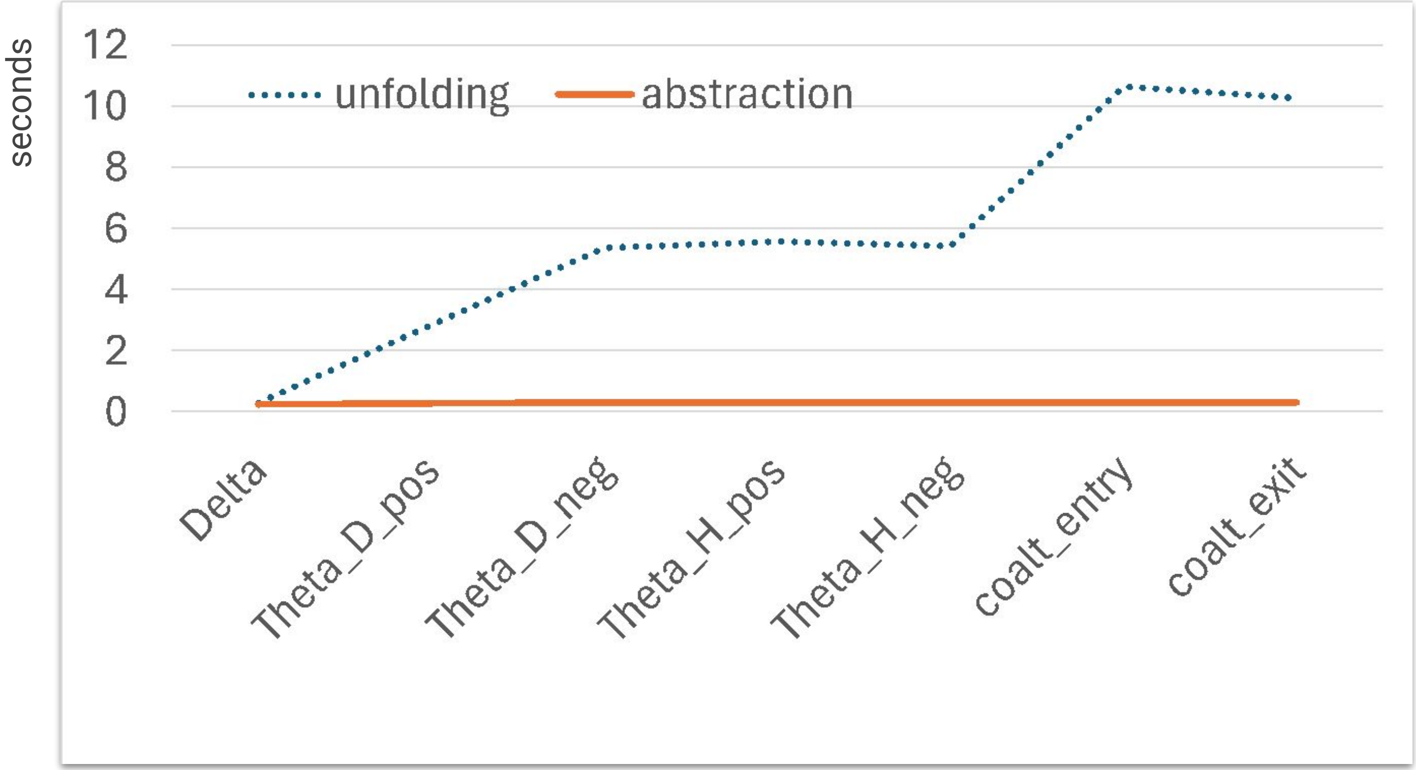
Before:



Now:



Function calls abstract analysis experiments



What's next for PRECiSA and Reflow?

- ReFlow is currently under revision for NASA open-source release
- Currently limited support for a class of for loops
  ⇒ Add support for more complex loops
- Improve the precision in the instability analysis for conditional and loops
- Reduce the complexity of the ACSL annotation
- Integration with precision optimization tools (Herbie)

Example: detect-and-avoid coordination

```
eps_line(vx, vy, sx, sy) =
    if (sx*vx + sy*vy) * (sx*vx - sy*vy) > 0 then
        1 // right turn
    else
        -1 // left turn
```

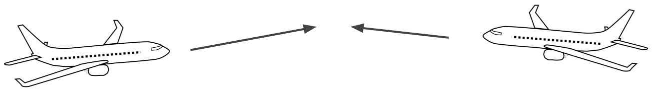
Instrumentation to detect instability

```
eps_line'(vx, vy, sx, sy) =
    if (sx*vx + sy*vy) * (sx*vx - sy*vy) > ε then
        1 // right turn
    elsif (sx*vx + sy*vy) * (sx*vx - sy*vy) ≤ -ε then
        -1 // left turn
    else ω // warning!
```

ε is a sound overestimation of the error of the expression

Strengthen the guards

Cases in which the rounding error may affect the evaluation of the guard

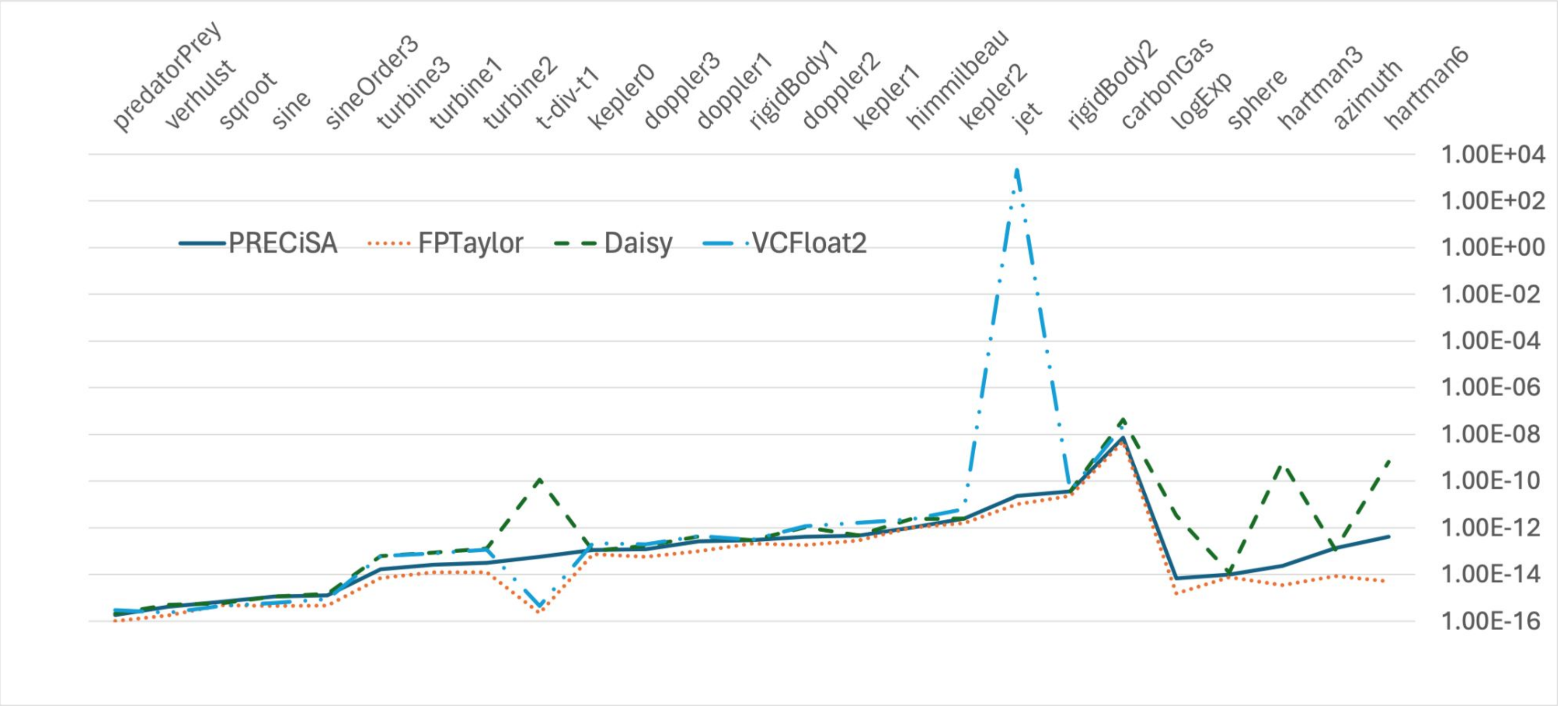


Titolo L., Muñoz C., Feliú M., Moscato M., Eliminating unstable tests in floating-point programs. LOPSTR 2018.

Tool comparison

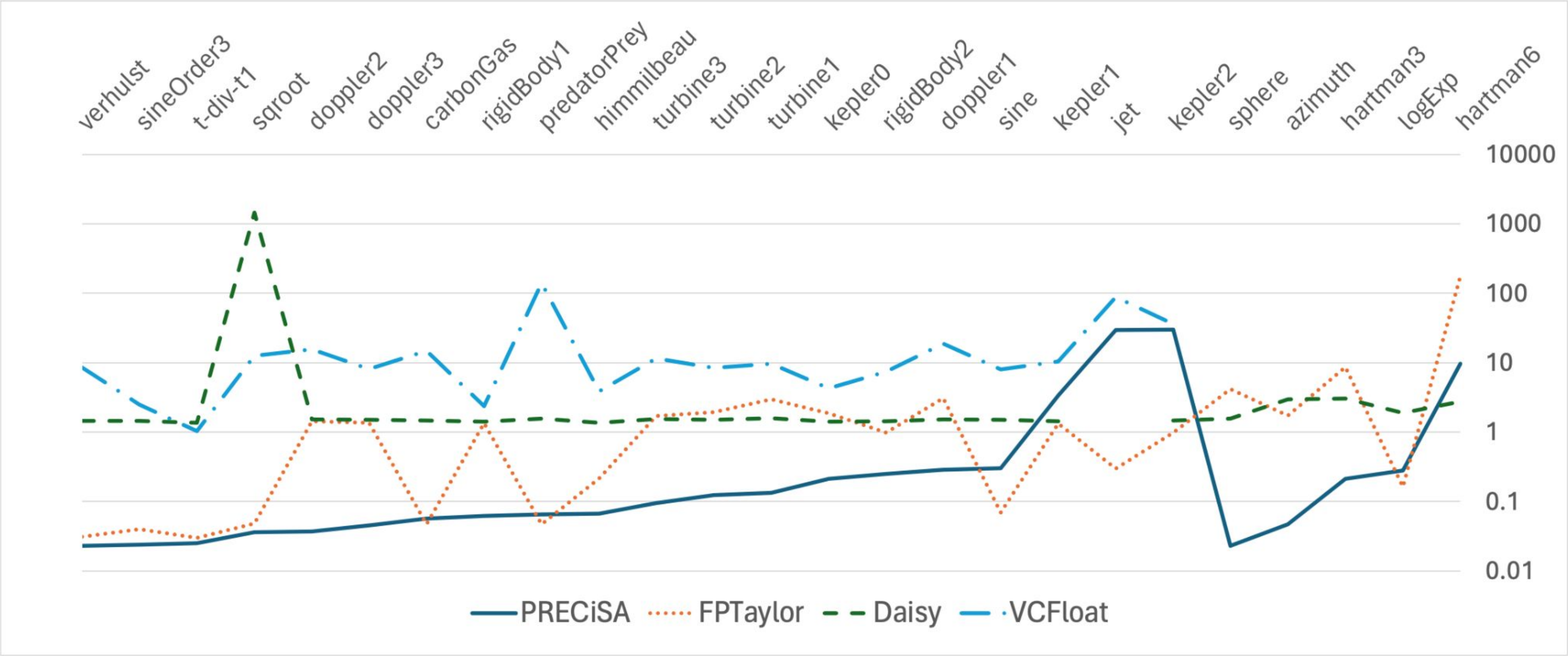
	PRECiSA	FPTaylor	Daisy	VCFloat	Fluctuat	Gappa
proof certificates	✓	✓	✗	✓	✗	✓
conditionals	✓	✗	✗	✗	✓	✗
instability detection	✓	✗	✗	✗	✓	✗
instability analysis	✓	✗	✗	✗	✗	✗
function calls	✓	✗	✗	✗	✓	✗
bounded loops	✓	✗	✗	✗	✓	✗
widening	✓	✗	✗	✗	✓	✗
data collections	✓	✗	✓	✗	✓	✗
rounding modes	✗	✓	✗	✗	✗	✗
fixed-point arith.	✗	✗	✓	✗	✗	✓

Tool comparison



Experimental results for absolute round-off error bounds.

Tool comparison



Times in seconds for the generation of round-off error bounds.