Software Verification of Safety-Critical Aerospace Systems[1]

César A. Muñoz
Alwyn Goodloe
{cesar.a.munoz,a.goodloe}@nasa.gov

Frama-C Day 2016
June 20th, 2016

[1] This presentation reports joint work with F. Kirchner, L. Correnson, and G.-A. Jaloyan.

Lifecycle of Our FM Research Activities

1. **Develop** functional requirements for advanced Air Traffic Management concepts (mainly in PVS).
2. Formally **verify** that those functional requirements satisfy operational requirements (mainly in PVS).
3. Formally **specify** algorithms that satisfy those functional requirements and formally **prove** their correctness (mainly in PVS).
4. Either manually **write** or automatically **generate** prototype code that implements those algorithms (mainly for testing).
5. **Repeat.**

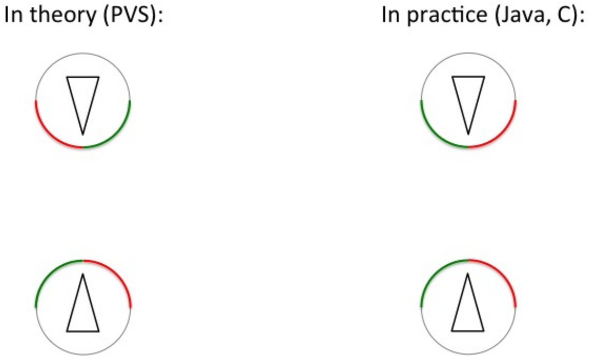
Lifecycle of Our FM Research Activities

1. **Develop** functional requirements for advanced Air Traffic Management concepts (mainly in PVS).
2. Formally **verify** that those functional requirements satisfy operational requirements (mainly in PVS).
3. Formally **specify** algorithms that satisfy those functional requirements and formally **prove** their correctness (mainly in PVS).
4. Either manually **write** or automatically **generate** prototype code that implements those algorithms (mainly for integration).
5. **Release** code under NASA's Open Source Agreement.
6. **Repeat**.

- Research prototypes:
  - Mid- and low-fidelity simulation environments.
  - Flight experiments and demonstrations.
  - Reference implementation of minimum operational standards.
- The algorithms are formally verified, but is the code correct?
- Frama-C:
  - Verification of numerically intensive code.
  - Verification of automatically generated monitors.

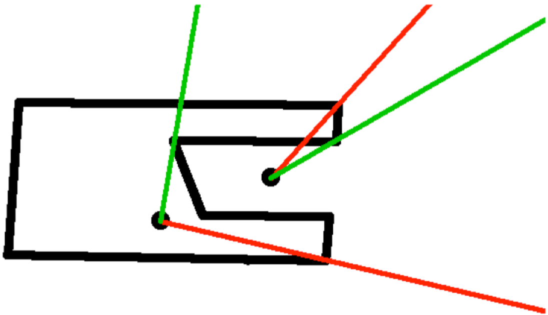
Verification of Numerical Software

Right-of-way in air traffic



Solution has been proposed by T. Nguyen using Frama-C.[2]

[2] *Taking architecture and compiler into account in formal proofs of numerical programs'*, PhD. Thesis, University of Paris-Sud, 2012.

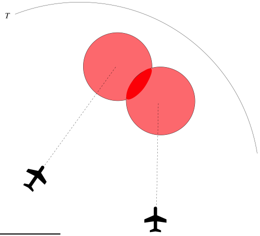


Algorithm has been verified in PVS by A. Narkawicz and G. Hagen.[3]

[3] *Algorithms for Collision Detection Between a Point and a Moving Polygon, with Applications to Aircraft Weather Avoidance*, Proceedings of ATIO 2016.

Objective

- Develop techniques for lifting formally verified algorithms that use real arithmetic into formally verified software.
- Our algorithms:
  - Formally specified and verified in PVS.
  - Simple control logic, e.g., conditionals, bounded loops.
  - No memory management.
  - Numerically intensive: non-linear arithmetic, trig functions, etc.
- Case Study: ACCoRD's CD2D[4].



[4] A. Goodloe, C. Munoz, F. Kirchner, L. Correnson, *Verification of Numerical Programs: From Real Numbers to Floating Point Numbers*, Proceedings of NFM2013.

CD2D In A Nutshell

$$cd2d(\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i) \equiv \texttt{let}\ \ \mathbf{s} = \mathbf{s}_o - \mathbf{s}_i, \mathbf{v} = \mathbf{v}_o - \mathbf{v}_i\ \ \texttt{in}\ los?(\mathbf{s}) \text{ or } \omega(\mathbf{s}, \mathbf{v}) < 0,$$

$$los?(\mathbf{s}) \equiv \sqrt{s_x^2 + s_y^2} < D,$$

$$\omega(\mathbf{s}, \mathbf{v}) \equiv \begin{cases} \mathbf{s} \cdot \mathbf{v} & \text{if } \mathbf{s}^2 = D^2, \\ \mathbf{v}^2\mathbf{s}^2 + 2\tau(\mathbf{s} \cdot \mathbf{v}) + \tau^2(\mathbf{s}, \mathbf{v}) - D^2\mathbf{v}^2 & \text{otherwise,} \end{cases}$$

$$\tau(\mathbf{s}, \mathbf{v}) \equiv \min(\max(0, -(\mathbf{s} \cdot \mathbf{v})), T\mathbf{v}^2).$$

Proposition 1. *Given a distance $D > 0$ and a lookahead time $T > 0$, for all vectors $\mathbf{s} = \mathbf{s}_o - \mathbf{s}_i$ and $\mathbf{v} = \mathbf{v}_o - \mathbf{v}_i$,*

(soundness) *If $conflict?(\mathbf{s}, \mathbf{v})$ holds then $cd2d(\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i)$ returns true.*
(completeness) *If $cd2d(\mathbf{s}_o, \mathbf{v}_o, \mathbf{s}_i, \mathbf{v}_i)$ returns true then $conflict?(\mathbf{s}, \mathbf{v})$ holds.*

$$conflict?(\mathbf{s}, \mathbf{v}) \equiv \exists 0 \le t \le T : los?(\mathbf{s} + t\mathbf{v}).$$

1. Transform PVS algorithms and specifications into C code and ACSL annotations.
2. Instrument the code and its specifications with arbitrary initial bounds to computation errors.
3. Use Frama-C to generate verification conditions.
4. Use Gappa to verify conditions.
5. If goals are discharged decrease bounds and go to 3.
6. Otherwise, increase bounds and go to 3.

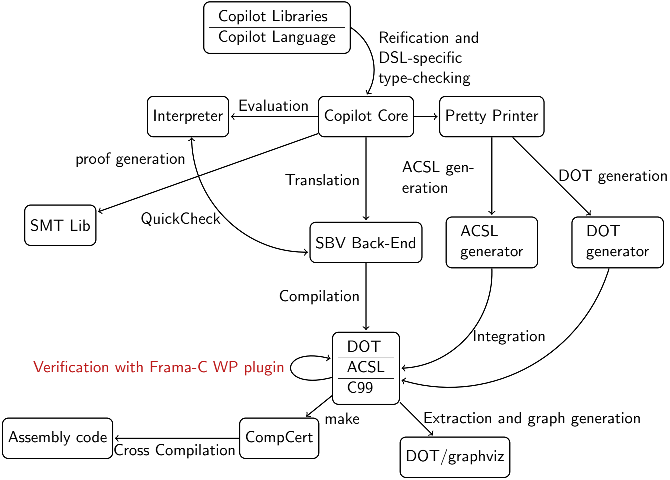
Verification of Runtime Monitors

- Given the current state-of-the art, not all code can be formally verified.
- Runtime monitors detect and respond to property violation at execution time:
  - Logical specification ϕ.
  - Execution trace τ of state information of the system under observation (SUO).
  - Decide if τ satisfies ϕ.

Copilot Language

- Copilot is an *EDSL* (embedded domain specific language), embedded in *Haskell* and used for writing *runtime monitors* for hard real-time, distributed, reactive systems written in C.
- A Copilot program is a list of streams defined by mutually recursive stream equations.
- Programs can be interpreted and analysed using proof engines, e.g., Z3, CVC4, dReal, Kind, ...
- Programs can be compiled to C using two back-ends: SBV, ATOM.
- Does the C code correspond to the original representation?

The Copilot Toolchain



- ACSL assertions are constructed by induction on the syntax, when pretty-printing Copilot Core.
- WP and CVC4 are used to verify that the C code corresponds to the Copilot Core.

Example of Annotated Monitor Code

```
/*@
 assigns \nothing;
 ensures \result == (((ext_ident_double_8) -
           (((ext_minimal_horizontal_separation) *
             (ext_minimal_horizontal_separation)))));
*/
SDouble ext_sqrt_9_arg0(const SDouble ext_ident_double_8,
   const SDouble ext_ownship_position_x,
   const SDouble ext_intruder_position_x,
   const SDouble ext_ownship_position_y,
   const SDouble ext_intruder_position_y,
   const SDouble ext_minimal_horizontal_separation)
  {
   const SDouble s0 = ext_ident_double_8;
    const SDouble s5 = ext_minimal_horizontal_separation;
    const SDouble s6 = s5 * s5;
    const SDouble s7 = s0 - s6;
    return s7;
   }
```

Concluding Remarks

- We have successfully verified C code that uses floating point computations and C code that is automatically generated from runtime monitors.
- Challenges in the verification of aerospace systems:
  - Even small functional programs with no loops and no memory allocation generate very large verification conditions.
  - These verification conditions are usually beyond the capabilities of automated theorem provers, e.g., Z3, MetiTarski, etc.
  - In the case of interactive theorem proving, these verification conditions usually lead to the statement explosion problem.

Statement Explosion Problem

```
[-1]  eps = 1 OR eps = -1
[-2]  v‘y*eps <= 0
[-3]  rd‘y*eps < 0
[-4]  ((v‘x = 0 AND v‘y = 0) IMPLIES rd‘x >= 0)
[-5]  ((v‘x /= 0 OR v‘y /= 0) IMPLIES rd‘x > v‘x)
[-6]  rd‘x*v‘y*eps-rd‘y*v‘x*eps <= 0
[-7]  mps‘y*eps+rd‘y*eps < 0
[-8]  v‘x >= 0
[-9]  (dv‘x /= 0 OR dv‘y /= 0)
[-10] mps‘x*rd‘y*eps-mps‘y*rd‘x*eps <= 0
[-11] -1*(dv‘x*mps‘y*eps)-dv‘x*rd‘y*eps+ dv‘y*mps‘x*eps+dv‘y*rd‘x*eps < 0
[-12] ((rd‘x*mps‘x+rd‘x*rd‘x+rd‘y*mps‘y+rd‘y*rd‘y < 0 AND
      dv‘x*rd‘y*eps-dv‘y*rd‘x*eps < 0) OR (rd‘x*mps‘x+rd‘x*rd‘x+
      rd‘y*mps‘y+rd‘y*rd‘y >= 0 AND dv‘x*mps‘x+dv‘x*rd‘x+dv‘y*mps‘y+
      dv‘y*rd‘y > rd‘x*mps‘x+rd‘x*rd‘x+rd‘y*mps‘y+rd‘y*rd‘y
      AND dv‘x*rd‘y*eps-dv‘y*rd‘x*eps <= 0))
  |-------
[1] (dv‘x /= 0 OR dv‘y /= 0) AND dv‘y*eps < 0 AND ((v‘x = 0 AND v‘y = 0)
    IMPLIES dv‘x >= 0) AND ((v‘x /= 0 OR v‘y /= 0) IMPLIES dv‘x > v‘x)
    AND dv‘x*v‘y*eps-dv‘y*v‘x*eps <= 0
```