Development and Formal Verification of a Flight Stack for a High-Altitude Micro Glider

Emanuel Regnath
Toulouse, 10.10.2017
Motivation

https://www.brightwork.com/blog/project-failures-boeings-787-dreamliner
Formal Verification

Rejected because considered to ...

• require a lot of additional specification
• require user interaction; little automation
• require experts; results are difficult to understand
Mission – Novel Weather Balloon

- **Intro**
- **Approach**
- **Devel**
- **Results**

The diagram illustrates the mission timeline and key stages:

- **Inflate**: Initial stage.
- **Takeoff**: 30' climb.
- **Uncontrolled Climb**: Initial ascent.
- **Release**: 60' climb stabilized.
- **Disengage**: Balloon reaches target altitude.
- **Homing**: Up to 150 km/h headwind expected.
- **Stable Gliding**: 60+x' stable gliding.
- **Landing**: 80-120' landing.
- **Retired**: Mission concluded.

**Altitude**

- **Target Altitude**: 10 km MSL.

**Time**

- **0**: Inflate.
- **30'**: Takeoff.
- **60'**: Release.
- **60+x'**: Stable gliding.
- **80-120'**: Landing.
- **Retired**: Mission completed.

**Other Notes**

- **5-10 m/s ascent rate.**
- **Target: 0 km (return home).**
Introduction to the Scenario

- **Sensors**
  - Baro
  - IMU
  - GPS
  - Mag

- **Flight Controller**

- **Servo left**

- **Elevon left**

100mm
The Roll of System Testing

- Full system tests, including external effects (wind, etc.)
- Risky and high effort (Time & Money) ⇒ as little as possible
- Germany: Must not fly above 100m AGL ⇒ limited
Development Process

1. Fast: before compiling
2. Normal: Continuous Integration with git
3. Nightly “deep” verification runs with long timeouts
Finding Defects – Expectation

- most by static analysis (each developer & nightly runs)
  - replace unit testing
  - identify under-specification
- few by system testing
  - defects which were missed by static analysis
  - defects which require context beyond source code
  - logging of exceptions: no reproduction issues
- none during operation
  - nevertheless: logging of exceptions & in-air reset
Ada & SPARK

ISO-8652
<defines>

Ada 2012 Language

<compiles>

GNAT compiler
by Adacore

SPARK 2014 Language

<verifies>

GNATprove
by Adacore
Verification Goals

We want to formally verify

**Absence of run-time errors**
Division by zero, overflows

**Integration Correctness**
Valid inputs and outputs

**Functional Behavior**
Input to output relation

**Information Flow**
Global variables, Input to output dependencies

**Physical Dimensions**
Compliance with physical laws
Verification Goals

We want to formally verify

- **Absence of run-time errors**
  - Division by zero, overflows

- **Integration Correctness**
  - Valid inputs and outputs

- **Functional Behavior**
  - Input to output relation

- **Information Flow**
  - Global variables, input to output dependencies

- **Physical Dimensions**
  - Compliance with physical laws
package MyPack with SPARK_Mode is

subtype Percentage is Natural;

Global_Ratio : Percentage;

procedure set_ratio( alt, maxalt : Integer )
  is begin
    Global_Ratio := alt * 100 / maxalt;
  end set_ratio;
end MyPack;
package MyPack with SPARK_Mode is
  subtype Percentage is Natural;
  Global_Ratio : Percentage;

procedure set_ratio( alt, maxalt : Integer ) is begin
  Global_Ratio := alt * 100 / maxalt;
end set_ratio;
end MyPack;

Mypack:9:35 medium: overflow check might fail
Mypack:9:41 medium: divide by zero might fail
Mypack:9:41 medium: range check might fail
package MyPack with SPARK_Mode is

  subtype Percentage is Natural;

  Global_Ratio : Percentage;

procedure set_ratio( alt, maxalt : Integer ) with
    Pre => alt,maxalt > 0 and alt < Integer'Last/100
is begin
  Global_Ratio := alt * 100 / maxalt;
end set_ratio;

  set_ratio( 42, 62 );
end MyPack;
package MyPack with SPARK_Mode is

    subtype Percentage is Natural;

    Global_Ratio : Percentage;

    procedure set_ratio( alt, maxalt : Integer ) with
    Pre => alt,maxalt > 0 and alt < Integer'Last/100
    is begin
        Global_Ratio := alt * 100 / maxalt;
    end set_ratio;

    set_ratio( 42, 62 );

end MyPack;
Absence of Run-Time Errors

```haskell
package MyPack with SPARK_MODE is

  subtype Tar_Alt is Integer range 10 .. 10_000;
  subtype Alt is Integer range 0 .. 100_000;
  subtype Percentage is Natural;

  Global_Ratio : Percentage;

  procedure set_ratio( val : Alt; max : Tar_Alt ) is begin
    Global_Ratio := val * 100 / max;
  end set_ratio;

end MyPack;
```
Absence of Run-Time Errors

```haskell
package MyPack with SPARK_Mode is

  subtype Tar_Alt is Integer range 10 .. 10_000;
  subtype Alt is Integer range 0 .. 100_000;
  subtype Percentage is Natural;

  Global_Ratio : Percentage;

  procedure set_ratio( val : Alt; max : Tar_Alt )
  is begin
    Global_Ratio := val * 100 / max;
  end set_ratio;

end MyPack;
```

Mypack:11:35 info: overflow check proved
Mypack:11:41 info: division check proved
Mypack:11:41 info: range check proved
Verification Goals

We want to formally verify

- **Absence of run-time errors**: Division by zero, overflows
- **Integration Correctness**: Valid inputs and outputs
- **Functional Behavior**: Input to output relation
- **Information Flow**: Global variables, input to output dependencies
- **Physical Dimensions**: Compliance with physical laws
Verification Goals

We want to formally verify

Absence of run-time errors
Division by zero, overflows

Integration Correctness
Valid inputs and outputs

Functional Behavior
Input to output relation

Information Flow
Global variables, Input to output dependencies

Physical Dimensions
Compliance with physical laws
package MyPack with SPARK_Mode is

  subtype Percentage is Natural;

  Global_Ratio : Percentage;

procedure set_ratio( alt, maxalt : Integer )
  is begin
    Global_Ratio := alt * 100 / maxalt;
  end set_ratio;

end MyPack;

Mypack:9:19 info: initialization of "Global_Ratio" proved
We want to formally verify

- **Absence of run-time errors**
  Division by zero, overflows

- **Integration Correctness**
  Valid inputs and outputs

- **Functional Behavior**
  Input to output relation

- **Information Flow**
  Global variables, input to output dependencies

- **Physical Dimensions**
  Compliance with physical laws
Verification Goals

We want to formally verify

- **Absence of run-time errors**
  Division by zero, overflows

- **Integration Correctness**
  Valid inputs and outputs

- **Functional Behavior**
  Input to output relation

- **Information Flow**
  Global variables, input to output dependencies

- **Physical Dimensions**
  Compliance with physical laws
Dimension Checking

Scientific: angular rate = 20 deg / 100 ms = 200 deg/s

C Program:

```c
1  Float angle = 20;
2  Float dt    = 0.1;
3
4  Float rate = dt / angle;
```

SPARK Program:

```spar
1  angle : Angle_Type := 20.0 * Degree;       -- Value: 0.524
2  dt    : Time_Type := 100.0 * Milli * Second; -- Value: 0.100
3  rate  : Angular_Velocity_Type := dt / angle;
4
5  yaw   : Angle_Type := 20.0;
```
Dimension Checking

Scientific: angular rate = 20 deg / 100 ms = 200 deg/s

C Program:

```c
1 Float angle = 20;
2 Float dt = 0.1;
3 Float rate = dt / angle;
```

SPARK Program:

```sp
1 angle : Angle_Type := 20.0 * Degree; -- Value: 0.524
2 dt    : Time_Type := 100.0 * Milli * Second; -- Value: 0.100
3 rate  : Angular_Velocity_Type := dt / angle;
4 yaw   : Angle_Type := 20.0;
```

Mypack:3:17 dimensions mismatch in assignment
Mypack:3:17 expected dimension [A.T**(-1)], found [T.A**(-1)]
Mypack:5:17 warning: assumed to be "20.0 Rad"
Verification Goals

We want to formally verify

- **Absence of run-time errors**
  - Division by zero, overflows

- **Integration Correctness**
  - Valid inputs and outputs

- **Functional Behavior**
  - Input to output relation

- **Information Flow**
  - Global variables, input to output dependencies

- **Physical Dimensions**
  - Compliance with physical laws
Verification Goals

We want to formally verify

- **Absence of run-time errors**
  Division by zero, overflows

- **Integration Correctness**
  Valid inputs and outputs

- **Functional Behavior**
  Input to output relation

- **Information Flow**
  Global variables, Input to output dependencies

- **Physical Dimensions**
  Compliance with physical laws
-- Functional Requirement
function FR_poshold_iff_no_course() return Boolean is (
  (Have_Course and G_state.mode /= MODE_POSHOLD) or
  (not Have_Course and G_state.mode = MODE_POSHOLD)
) with Ghost;

-- Functional Requirement
function FR_arrive_iff_near_target() return Boolean is (if (Have_Home_Position and Have_My_Position) then
  (dist2home < TARGET_R and G_state.mode = MODE_ARRIVED) or
  (dist2home >= TARGET_R and dist2home <= 2.0*TARGET_R) or
  (dist2home > 2.0*TARGET_R and G_state.mode /= MODE_ARRIVED)
else G_state.mode /= MODE_ARRIVED
) with Ghost;

-- Update the controller mode, depending on state
procedure Update_Homing() with
  Post => FR_poshold_iff_no_course and FR_arrive_iff_near_target;
Verification Goals

We want to formally verify

**Absence of run-time errors**
Division by zero, overflows

**Integration Correctness**
Valid inputs and outputs

**Functional Behavior**
Input to output relation

**Information Flow**
Global variables, Input to output dependencies

**Physical Dimensions**
Compliance with physical laws
Final GNATprove Results

- SPARK subprogram coverage: 82%
Final GNATprove Results

![Graph showing total properties and proven properties over time]

- Total properties
- Proven properties

Date in DD/MM:

- 19/07
- 24/07
- 29/07
- 03/08
- 08/08
- 13/08
- 18/08
- 28/08
- 02/09
- 07/09
- 12/09
- 17/09

Final Flight
Final GNATprove Results

Totals of verified properties

Absence of run-time errors
1487 / 1711 (86.9%)

Integration Correctness
277 / 282 (98.2%)

Functional Requirements
2 / 2 (100%)

Information Flow
1539 / 1540 (99.9%)

Physical Dimensions
?? / ?? (100%)
Final GNATprove Results

Total CPU time per VC type

Analysis time [s]

10^5
10^4
10^3
10^2
10^1
10^0
10^{-1}
10^{-2}

Fp Overflow Check
Range Check
Overflow Check
Length Check
Postcondition
Division Check
Assert
Precondition
Discriminant Check
Index Check
Contract Case
Ceiling Priority Protocol
Task Termination
Disjoint Contract Cases
Aliasing
Uninitialized
Final Flight on 2016-09-14

Target Altitude: 6100 m AGL
Final Flight on 2016-09-14

A/C Status
- Lat: 11.7479
- Lon: 48.1952
- Alt: 5731.2
- Hdg: 0
- G'Vel: WAIT
- AltMSL: 5731.2
- UTC: 14:16:47
Finding Defects – Reality

- most by static analysis (each developer & nightly runs)
  - removed all stupid bugs
  - identified under-specification
- few by system testing
  - masking defects during analysis
  - ignoring failed proofs
  - incomplete specification
- one during operation
  - faulty but non-crashing behavior
  - missed during system testing
  - unverified assumptions about sensor data (beyond code)
Conclusion

- Very little debugging work
  - Practically no exceptions during system testing
  - No issues with reproduction and isolation of failures

- SPARK tools work very well
  - Defect detection with almost no additional effort
  - Results are precise: MyPack\texttt{:9:35}: overflow check might fail
  - Effective multi-threading: separation of critical tasks
  - Verification automation as continuous integration with git
  - Verification of physical dimensions
  - Floats are difficult but possible
  - Verification of high-level behavior is difficult but possible

code released to open source: https://github.com/tum-ei-rcs/StratoX
“I think you should be more explicit here in step two.”

from *What’s so Funny about Science?* by Sidney Harris (1977)
References

- Adacore, SPARK 2014 user guide, version 18.0.
Formal Verification

Abstract Interpretation
- no missed bugs
- automatic

Model Checking
- no false alarms
- automatic

The Holy Grail
- no missed bugs
- automatic

Coverity
- HP Fortify
- MISRA-C Tools

Astrée (C)
- Frama-C/Value (C)
- CodePeer (Ada)

SPARK (Ada)
- Frama-C/Jessie (C)
- Frama-C/WP (C)
- ACL2 (Lisp)
- Isabelle, HOL, Coq

Deductive
- no false alarms
- no missed bugs

CBMC (C)
- KLEE (C)
my_sub(a : Integer) is
begin
add(a);
sub(a);
end my_sub;

preconditions

postconditions

precondition
add(x : Integer) is
begin
...
end add;

postcondition

precondition
sub(x : Integer) is
begin
...
end sub;

postcondition
-- Float Underflow

```haskell
function Sin ( x : Float ) return Float with
    Post => Sin'Result in -1.0 .. 1.0; -- OK

pragma Assert ( ( Sin(x) )**2 in -1.0 .. 1.0 ); -- Might fail
```
### Final GNATprove Results

<table>
<thead>
<tr>
<th>SPARK Analysis</th>
<th>Total</th>
<th>Flow</th>
<th>Interval</th>
<th>Proved</th>
<th>Justified</th>
<th>Unproven</th>
</tr>
</thead>
<tbody>
<tr>
<td>Data Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Flow Dependencies</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Initialization</td>
<td>1540</td>
<td>1510</td>
<td>366</td>
<td>1117</td>
<td>4</td>
<td>224</td>
</tr>
<tr>
<td>Non Aliasing</td>
<td>16</td>
<td>16</td>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Run-Time Checks</td>
<td>1711</td>
<td></td>
<td>366</td>
<td>1117</td>
<td>4</td>
<td>224</td>
</tr>
<tr>
<td>Assertions</td>
<td>15</td>
<td></td>
<td></td>
<td>15</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Functional Contracts</td>
<td>282</td>
<td></td>
<td>277</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>LSP Verification</td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td>3564</td>
<td>1526</td>
<td>366</td>
<td>1409</td>
<td>33</td>
<td>230</td>
</tr>
</tbody>
</table>

Subprogram Coverage: 538 / 1227 (43.8%) 538 / 654 (82%)

Proven Properties: 3334 / 3564 (93.5%)

Proven Run-Time Errors: 1487 / 1711 (86.9%)