Remove Before Flight

Defect-Free Software and Agile Development in SPARK 2014

Martin Becker

Chair of Real-Time Computer Systems (RCS)
Technical University of Munich

Presented at
Frama-C & SPARK Day 2017, Paris
May 30th, 2017

Lehrstuhl für
Realzeit-Computersysteme

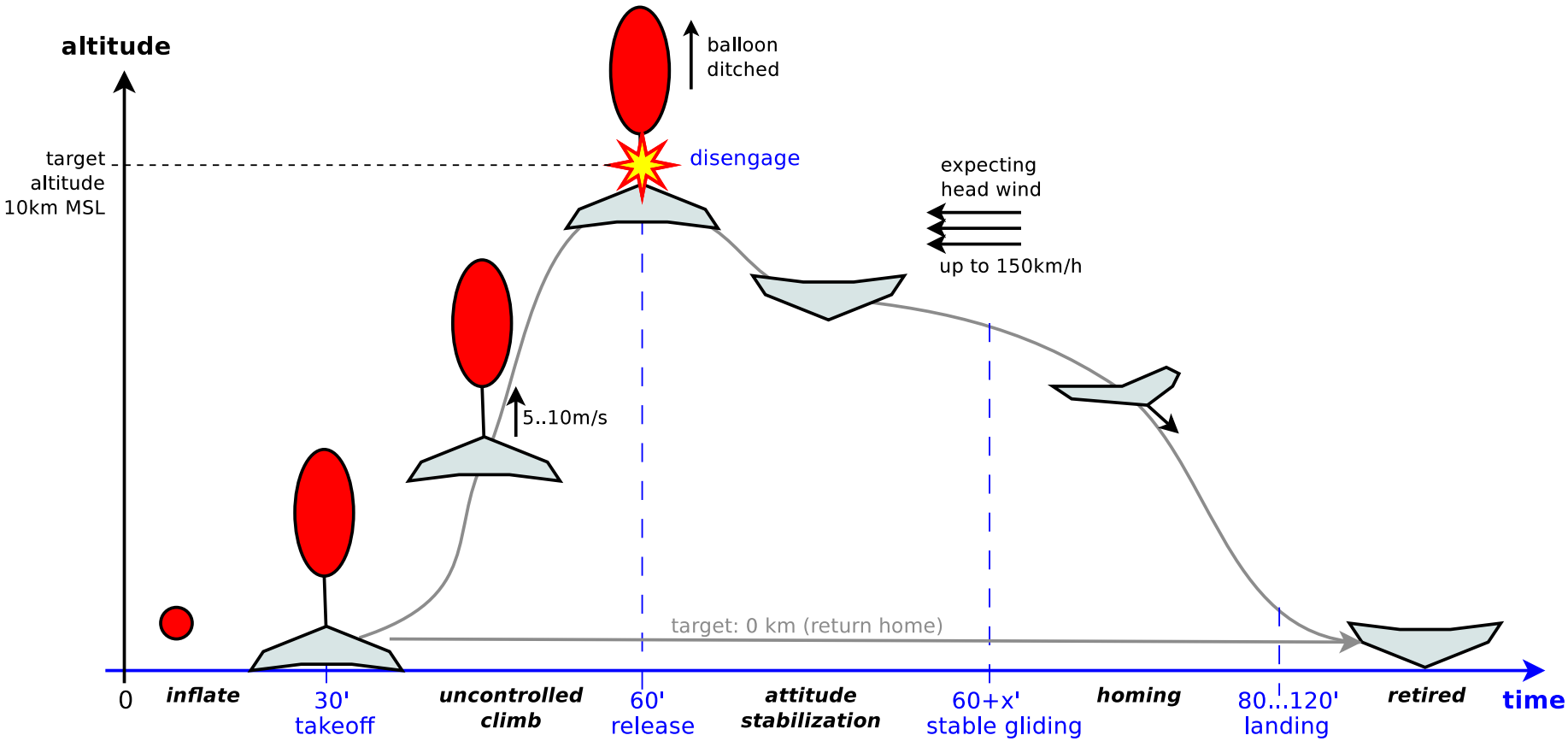
A Wish

Does it work with ‘‘bugs’’?
Which defects are removed, and when?

Outline

Briefing (Introduction)

Mission – A Novel Weather Balloon



Need to: monitor ascend, unhitch from balloon, return to take-off location (& record weather data)

Glider

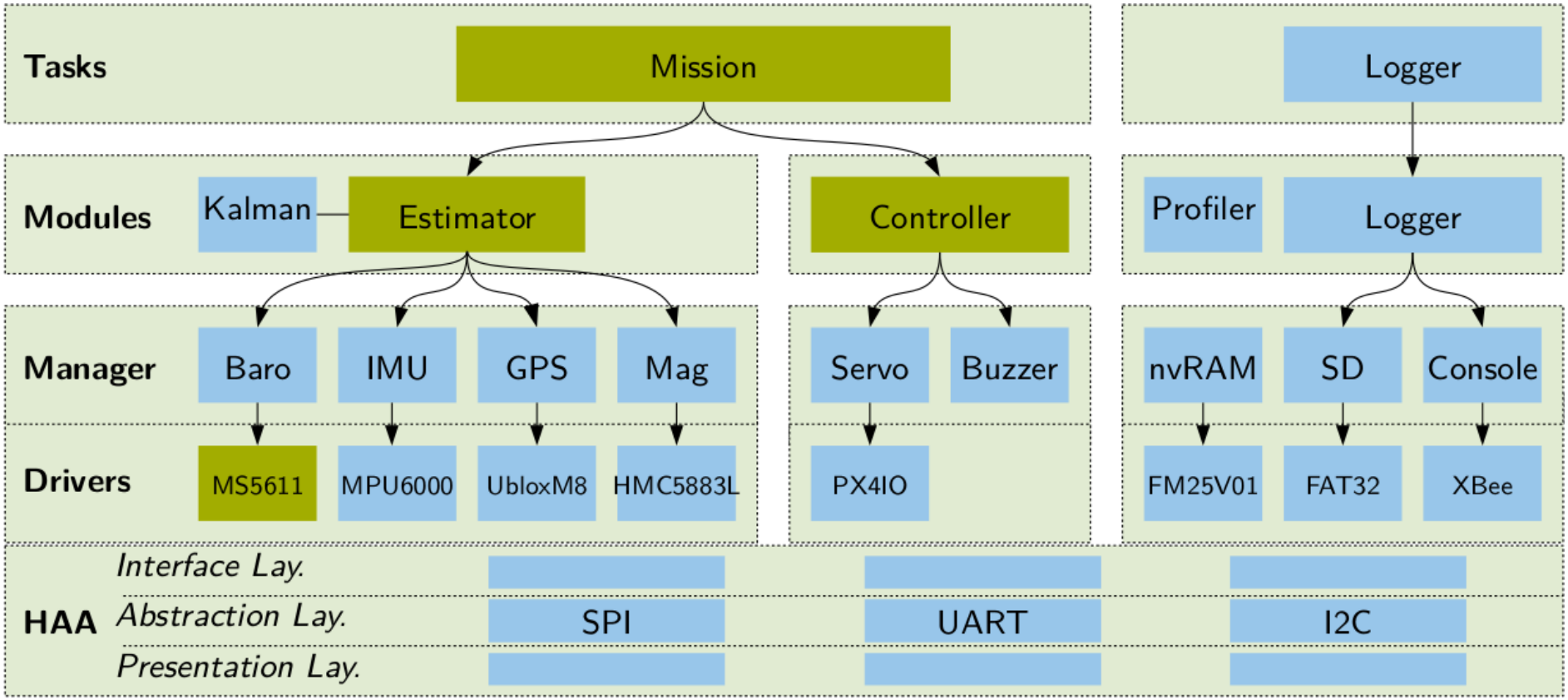
- high-altitude micro glider
  - $>$ 10km altitude, 30-100km/h air speed
  - non-motorized vehicle, MToW 400g
- need a flight stack
  - estimate attitude, stabilize flight, navigate w/ GPS, ...

Embedded Target

- ‘‘Pixhawk’’ autopilot, 168MHz ARM Cortex-M4(F)
  - 2MB Flash, 256kB RAM, SD Card Interface, NVRAM
  - Co-Processor with ‘‘failsafe’’ functionality for servo output
  - (cross-verification)
- 1x GPS, 1x Barometer, 1xMEMS+Gyro, 1x Magnetometer, 2x Acuator, 1x Buzzer
- Ada/SPARK Runtime System: Ravenscar Small Footprint
  - determinstig & analyzable multi-threading with constraints
  - RTS must be ported from similar Cortex-M4 (F409)
- also: based on AdaCore’s Driver Library/Bareboard Code
- need: custom drivers for our hardware, attitude estimator, mission handling, homing functionality

3 months time, 2 developers

Anatomy of the Flight Stack



- Flight-critical (“mission”) and non-flight-critical (“logging”) tasks communicate via message queue (protected object)
- goal: full SPARK coverage everywhere except HAA + **isolation of tasks**

Pre-Flight Checks (Development Process)

The Role of (Integration) Testing

- avoid loss of airframe (money and time)
- Germany: must not fly above 100m GND, final launch requires permission, insurance and manpower
- some low-altitude testing (risky and of limited use)

The Role of (Integration) Testing

- avoid loss of airframe (money and time)
- Germany: must not fly above 100m GND, final launch requires permission, insurance and manpower
- some low-altitude testing (risky and of limited use)

The Role of (Integration) Testing

- avoid loss of airframe (money and time)
- Germany: must not fly above 100m GND, final launch requires permission, insurance and manpower
- some low-altitude testing (risky and of limited use)
- goal: ‘‘de-bug’’ before going for any flight
  - unit verification through static analysis (SPARK2014, GNATprove), separation of criticality,
  - at least: no exceptions during flight, keep flight-critical task alive

The Role of (Integration) Testing

- avoid loss of airframe (money and time)
- Germany: must not fly above 100m GND, final launch requires permission, insurance and manpower
- some low-altitude testing (risky and of limited use)
- goal: ‘‘de-bug’’ before going for any flight
  - unit verification through static analysis (SPARK2014, GNATprove), separation of criticality,
  - at least: no exceptions during flight, keep flight-critical task alive

As little as possible, only when software is stable.

The SPARK 2014 Programming Language

- language intended for formal verification, successor of SPARK
  - strongly typed, imperative, object-oriented
- adopted syntax of Ada 2012 $\Leftrightarrow$ both languages can be mixed within an application
- SPARK 2014 = ''constrained subset of Ada 2012'':
  - no access (pointers), no aliasing, no exception *handling*, ...
  - need for proving the absence of exceptions
- user can specify functional contracts, data flow contracts and assertions
  - first-order logic, executable semantics, IEEE-754 Floats
- static analysis (i.a., deductive theorem proving)
  - proof for absence of errors (exceptions + failing contracts)
- Used Implementation: AdaCore's SPARK 2014, version GPL 2016 (partially Pro 17)

Debugging Goals

Ways of Finding Defects

1. most by static analysis (each developer & nightly runs)
   - replace unit testing (no harness + fixture, no env. sim.)
   - poinpoints defects and some amount of under-specification
2. few by integration testing
   - defects which were missed by static analysis
   - defects which require context beyond source code
   - logging of exceptions: poinpoint infections, no chasing through cause-effect chains
   - defects of three kinds:
     - masking defects during analysis / careless usage of verifier
     - ignoring failed proofs / wrong process
     - wrong ‘‘verification fixes’’ (saturation) / incomplete specification
3. none during operation
   - nevertheless: logging of exceptions & in-air reset

Nightly Verification Builds

Nightly ‘‘deep’’ verification runs with long timeouts

- Jenkins server on Intel Xeon E5-2680 Octa-Core, 2.7GHz, 16GB RAM
- GNATprove works *all cores*, allowing for high steps

Output: Verification summary, sent to all developers

- complete verification log (`file.adb:13:12: overflow check might fail (e.g., when ...)`)
- gnatprove.out (type of property, % verified, which prover)
- **a custom verification summary in tabular form:**

Nighly Verification Table

compilation unit	props success	flows success	props proven	ents	cover	num props
px4io.protocol	100	100	54	1	100	54
bounded_image	100	100	35	11	100	35
magnetometer	100	100	18	7	100	18
servo	100	100	10	9	100	10
mpu6000.register	100	100	1	1	100	1
ublox8.driver	99.0	100	98	28	100	99
px4io.driver	98.4	100	124	19	100	126
main	97.4	100	38	4	100	39
mission	95.6	100	86	25	100	90
units.navigation	85.4	100	129	16	100	151
hil.devices.nvram	76.2	100	16	10	100	21
ms5611.driver	75.4	100	135	45	100	179
controller	67.3	86.9	111	46	100	165
kalman	65.0	100	91	18	100	140
imu	50.7	100	70	20	100	138
nvram	100	100	15	27	92.6	15
logger	96.9	100	62	21	90.5	64
ulog	98.5	100	133	14	85.7	135
mpu6000.driver	74.6	100	126	29	79.3	169
...	...	...	...	..	...	...

Nighly Verification Table

- different view than GNATprove report
  - success (% discharged)
  - coverage (%SPARK ‘‘on’’) per unit
  - absolute counts
  - ...
- ordered by coverage, then success
- goal: promote flight-critical units to the top
  - developers work on critical parts first
- decision taken every morning
- code base grows quickly, but is kept in check

Latest version: separation by task (‘‘flight-critical’’ table vs ‘‘non-critical’’ table)

Request to Taxi (Verification Outcome)

Experiences

Catching Missing Requirements

An effective way to avoid overflows and range errors?

```
1 baro_alt : Altitude_Type;
  if Float (alt_int) >= Altitude_Type'Last then
     baro_alt := Altitude_Type'Last;
  elsif Float (alt_int) <= Altitude_Type'First then
     baro_alt := Altitude_Type'First;
6 else
     baro_alt := Float (alt_int16);
  end if;
```

Even better (create a generic and use it all over the place):

```
  function Sat_Cast_Alt is new Saturated_Cast (Altitude_Type);
2 baro_alt := Sat_Cast_Alt (alt_int16);
```

Experiences

Catching Missing Requirements

Consider this[1]:

```
data_rx : Byte_Arr (0..91) := Read_From_Device (GPS_UART_0);
-- ...
subtype Lat_Type is Angle_Type range -90.0 .. 90.0;
Lat : Lat_Type := Sat_Cast_Lat(Float(data_rx(28..31))*1E-7);
```

Static analysis: no range errors. However:

[1] code simplified

Experiences

Catching Missing Requirements

Consider this[1]:

```
1 data_rx : Byte_Arr (0..91) := Read_From_Device (GPS_UART_0);
  -- ...
  subtype Lat_Type is Angle_Type range -90.0 .. 90.0;
  Lat : Lat_Type := Sat_Cast_Lat(Float(data_rx(28..31))*1E-7);
```

Static analysis: no range errors. However:

- failure of GPS device $\Rightarrow$ silently ignored
- missing error handling for component/subsystem failure
- majority of such cases was missing a requirement

Saturation as means to ‘‘silence the prover’’ should be avoided

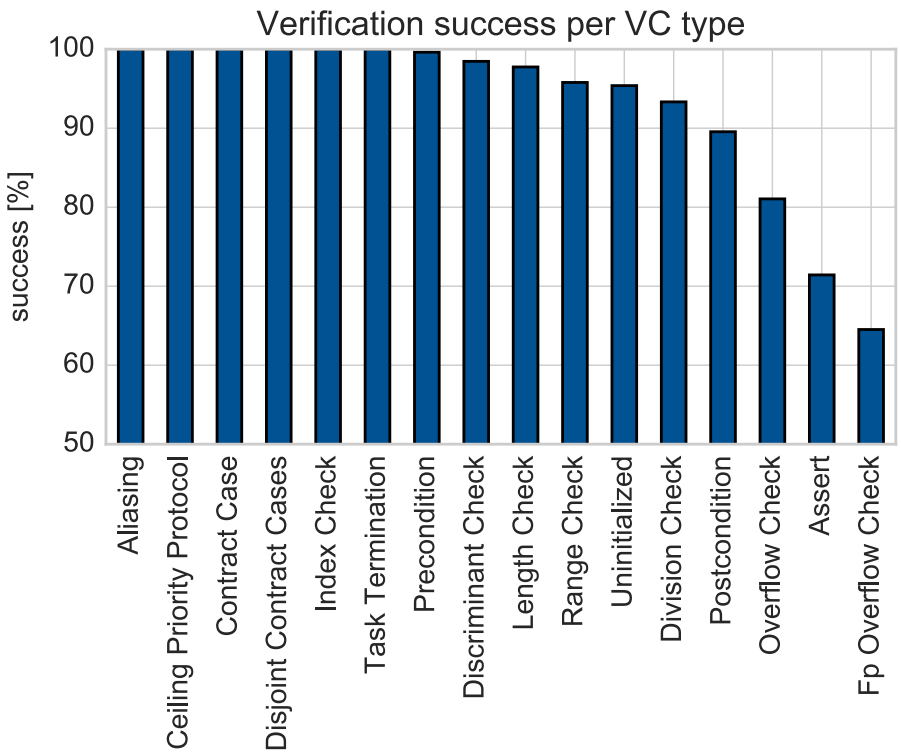
[1] code simplified

Attacking the Floats

- using the GNAT dimensionality system:

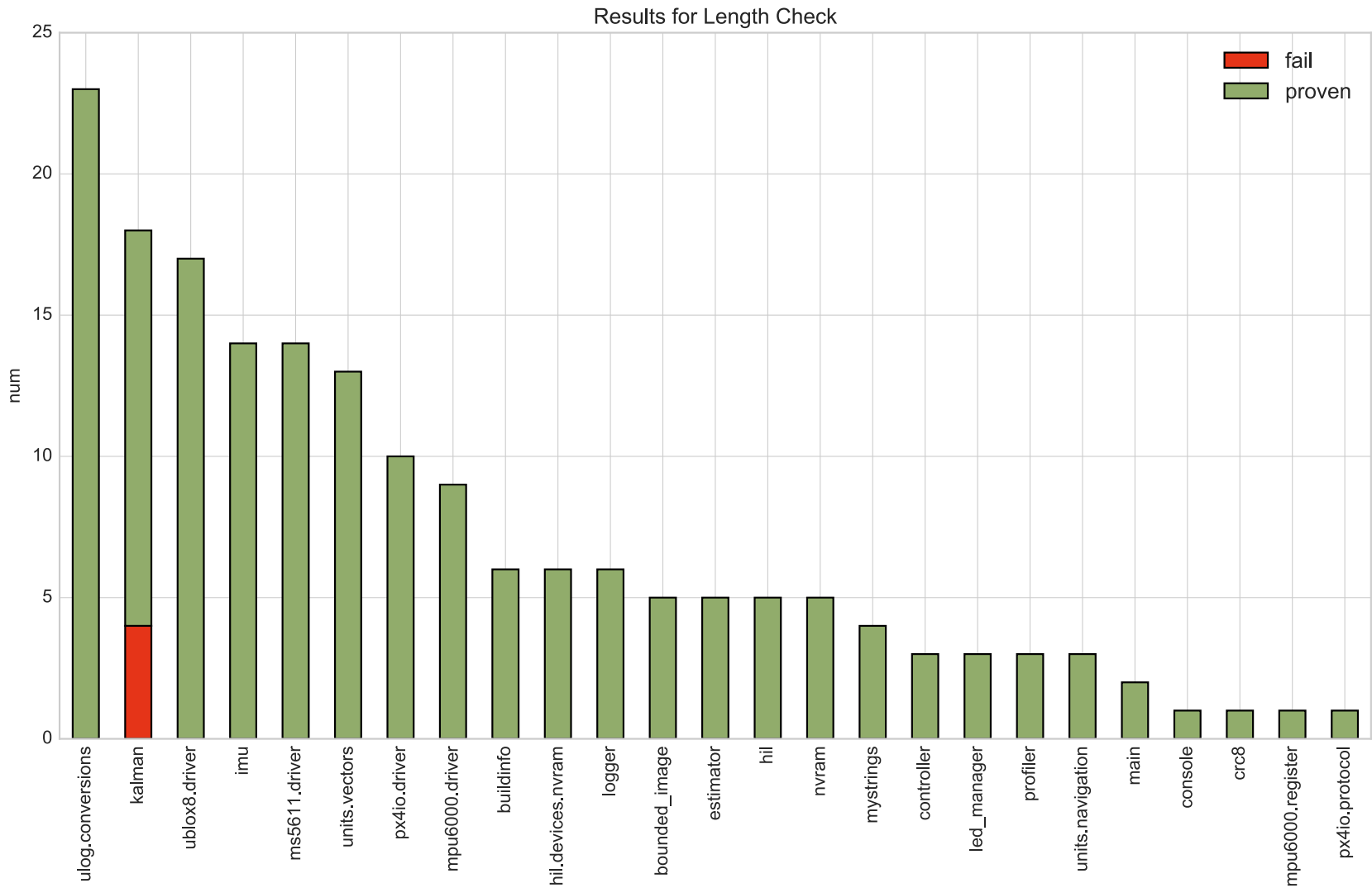
```
1 angle : Angle_Type := 20.0 * Degree;
  dt : Time_Type := 100.0 * Milli * Second;
  rate : Angular_Velocity_Type := dt/angle; -- comp. err.
```

- majority of floats range-limited
- how hard can it be?



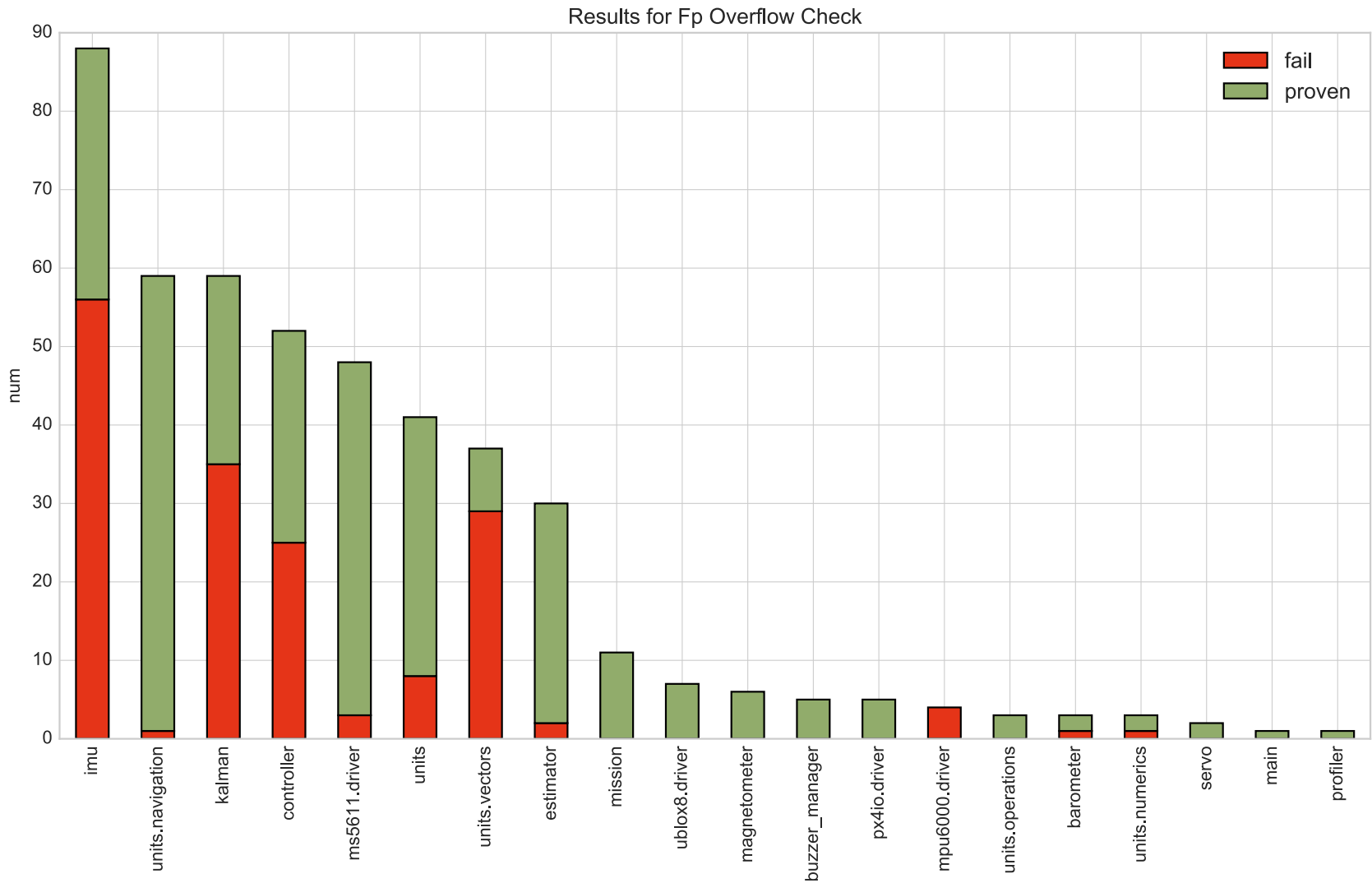
Attacking the Floats

Floats are hard to verify (but possible)



Attacking the Floats

Floats are hard to verify (but possible)



Ways out: SPARK lemma library, CodePeer

More Floats – Unexpected Exception

- CPU goes into reset loop during pre-flight checks
- log: exception in `units.vectors.rotate()`
- debugger (extremely hard to reproduce):

```
 3   package body Units.Vectors with SPARK_Mode is
 4
 5      procedure rotate
 6        (vector : in out Cartesian_Vector_Type;
 7         axis   :        Cartesian_Coordinates_Type;
 8         angle  :        Angle_Type) is
 9         result : Cartesian_Vector_Type := vector;
10
11      begin
12         case (axis) is
13            when X =>
14               result (Y) :=  Cos (angle) * vector (Y) - Sin
```

Units.Vectors.rotate

Messages Locations Debugger Execution <6> Debugger Console <6> Debu

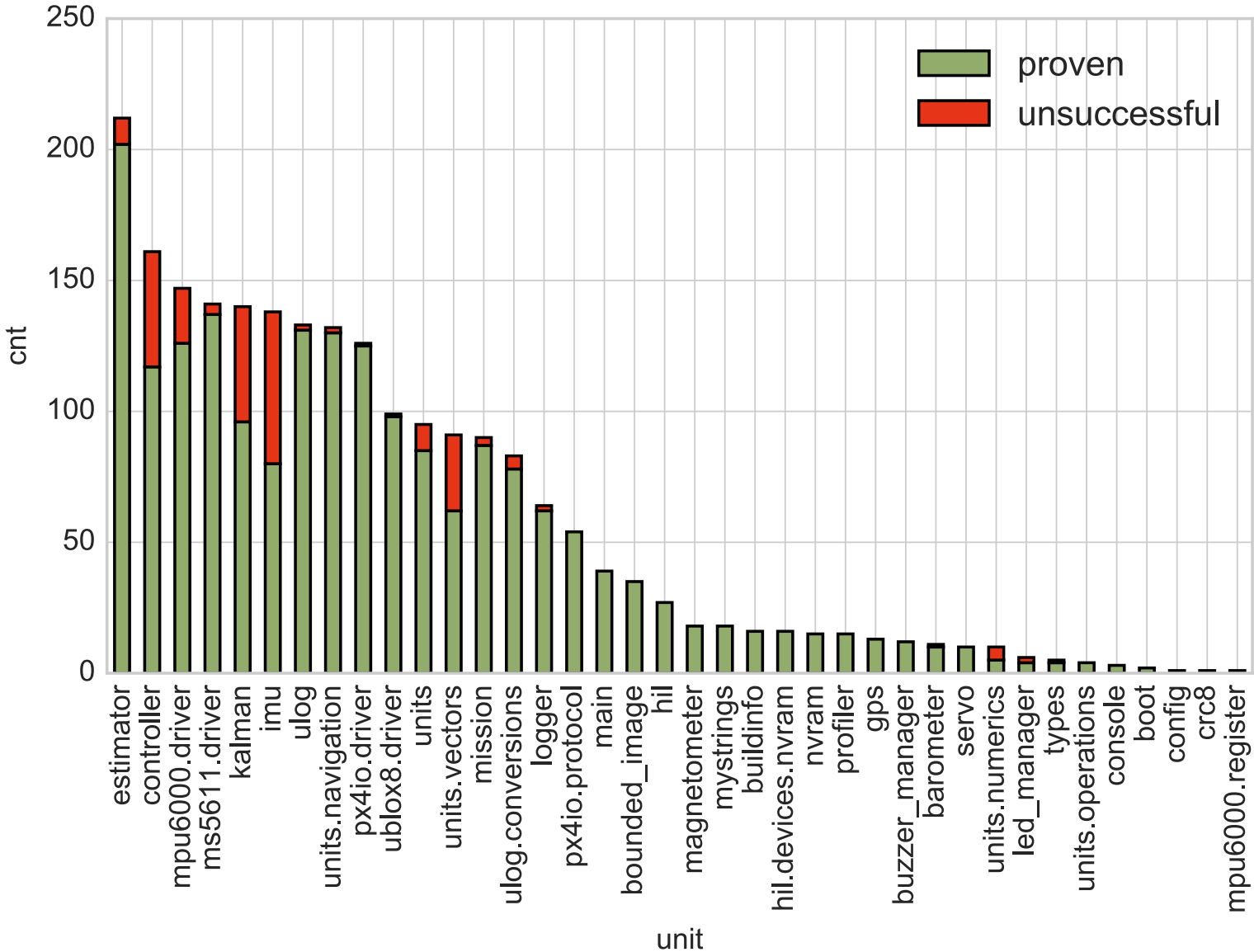
```
#1  0x08015a9c in units.vectors.rotate (vector=..., axis=x, angle=0.00429291604) at
(gdb) info locals
result = (0.00212701317, 0.0383495204, -2.02303554e-38)
(gdb) print angle
$11 = 0.00429291604
```

More Floats – Unexpected Exception

- CPU goes into reset loop during pre-flight checks
- log: exception in `units.vectors.rotate()`
- debugger (extremely hard to reproduce):
- static analysis had proven absence of runtime errors
- after some detour into assembly: run-time system configured incorrectly for target
  - config: FPU produces no denormals (“flush to zero”)
  - reality: FPU *does* produce denormals
  - internal `Float'Valid` check raises exception
- what happened: glider motionless long enough $\rightarrow$ gyro rates very small (2E-38) $\rightarrow$ rotating by small pitch/roll angle (glider level) $\rightarrow$ “numerical underflow”
- fix configuration of run-time system, mismatch gone

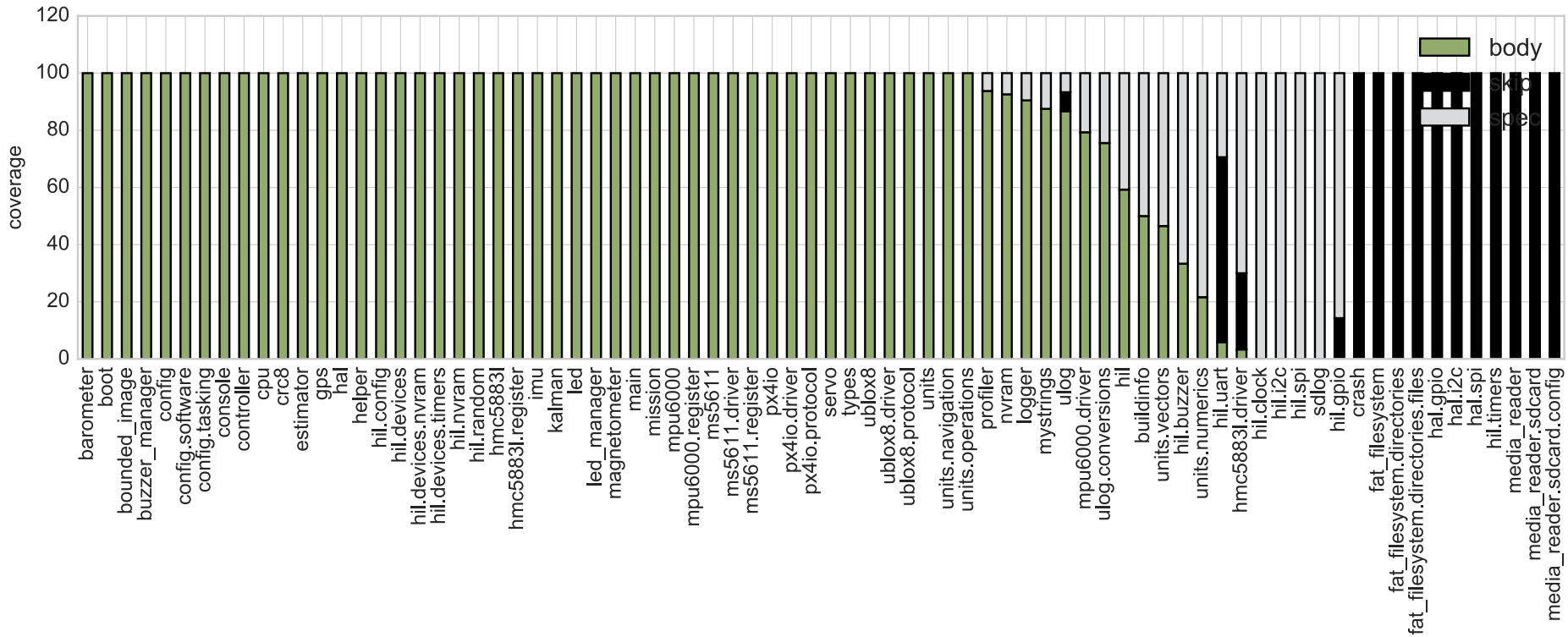
Reports for daily use

Success per Unit:



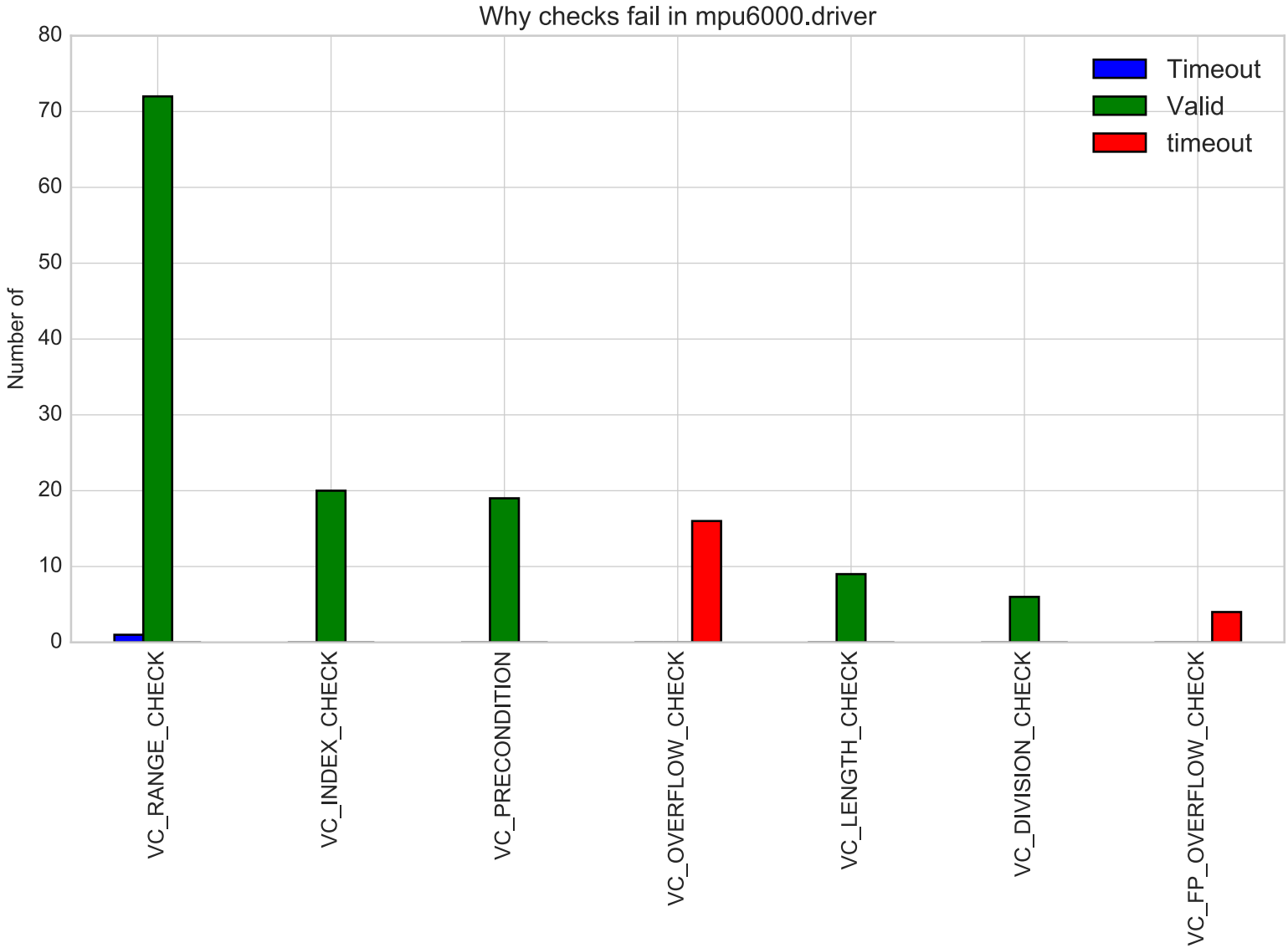
Reports for daily use

Unit SPARK coverage:

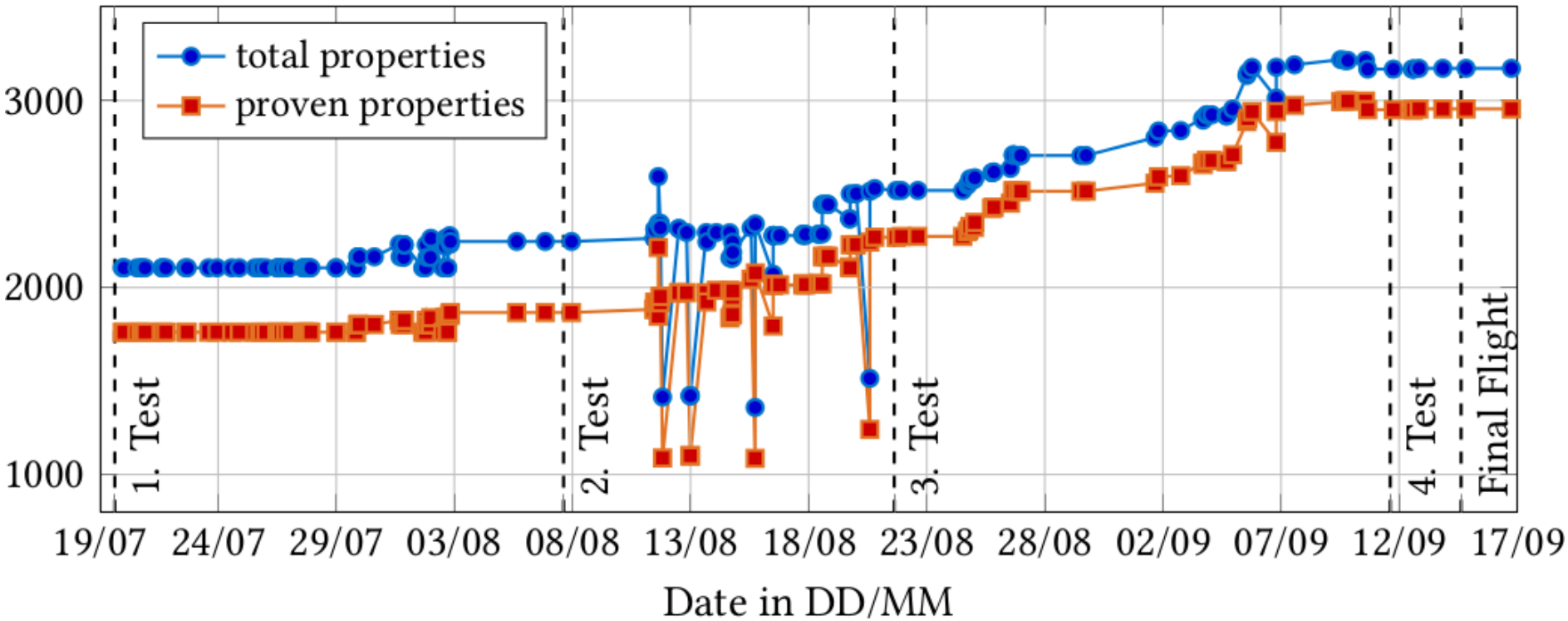


Reports for daily use

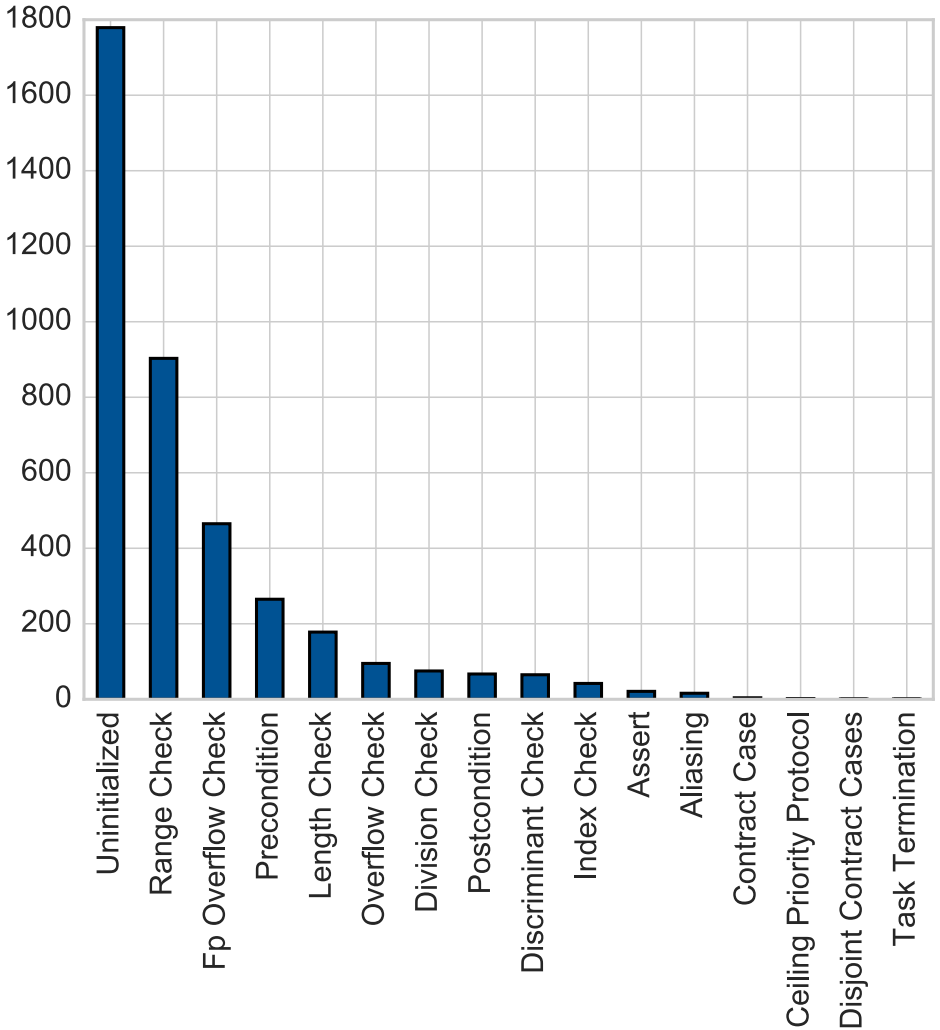
Why checks fail in unit:

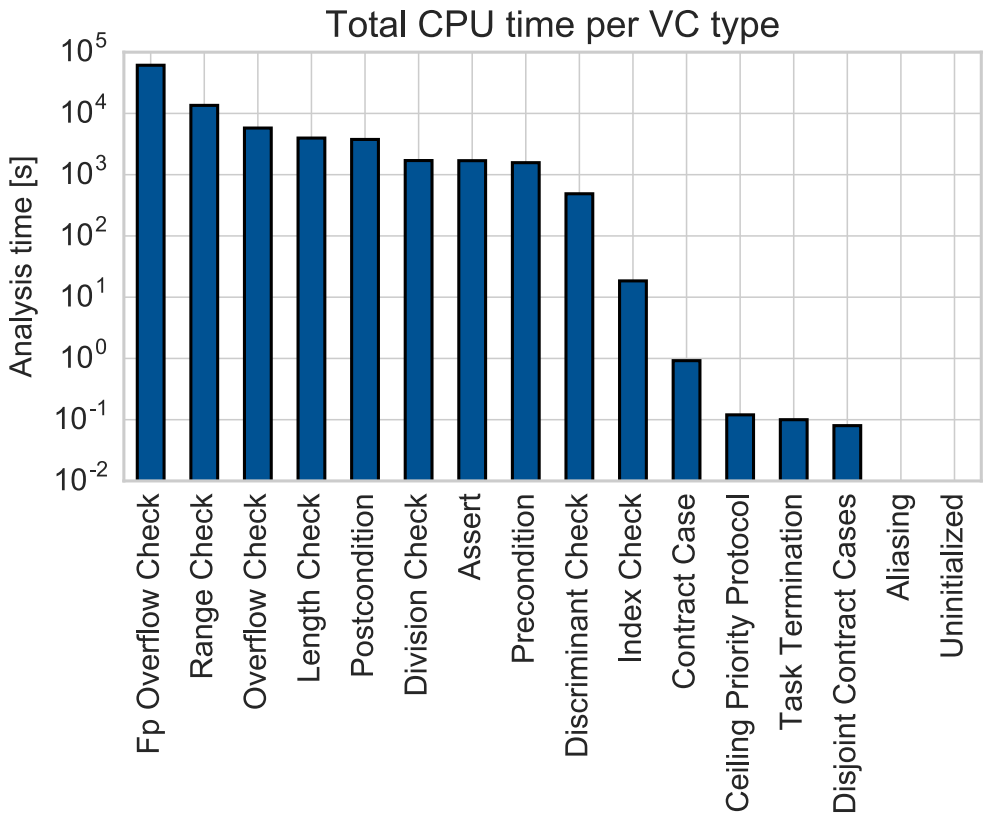


Success vs. Time



Computational Effort





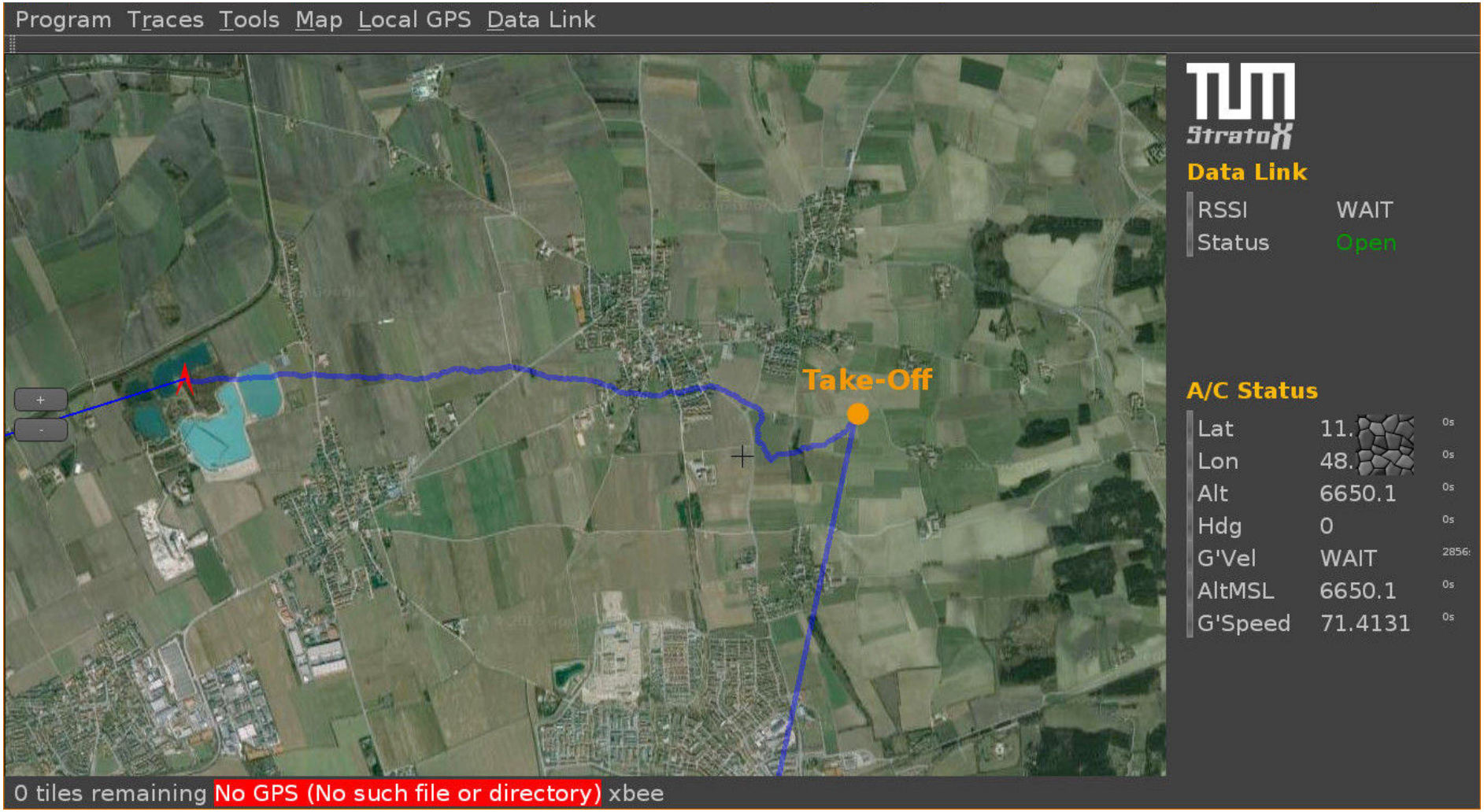
Debugging – Reality

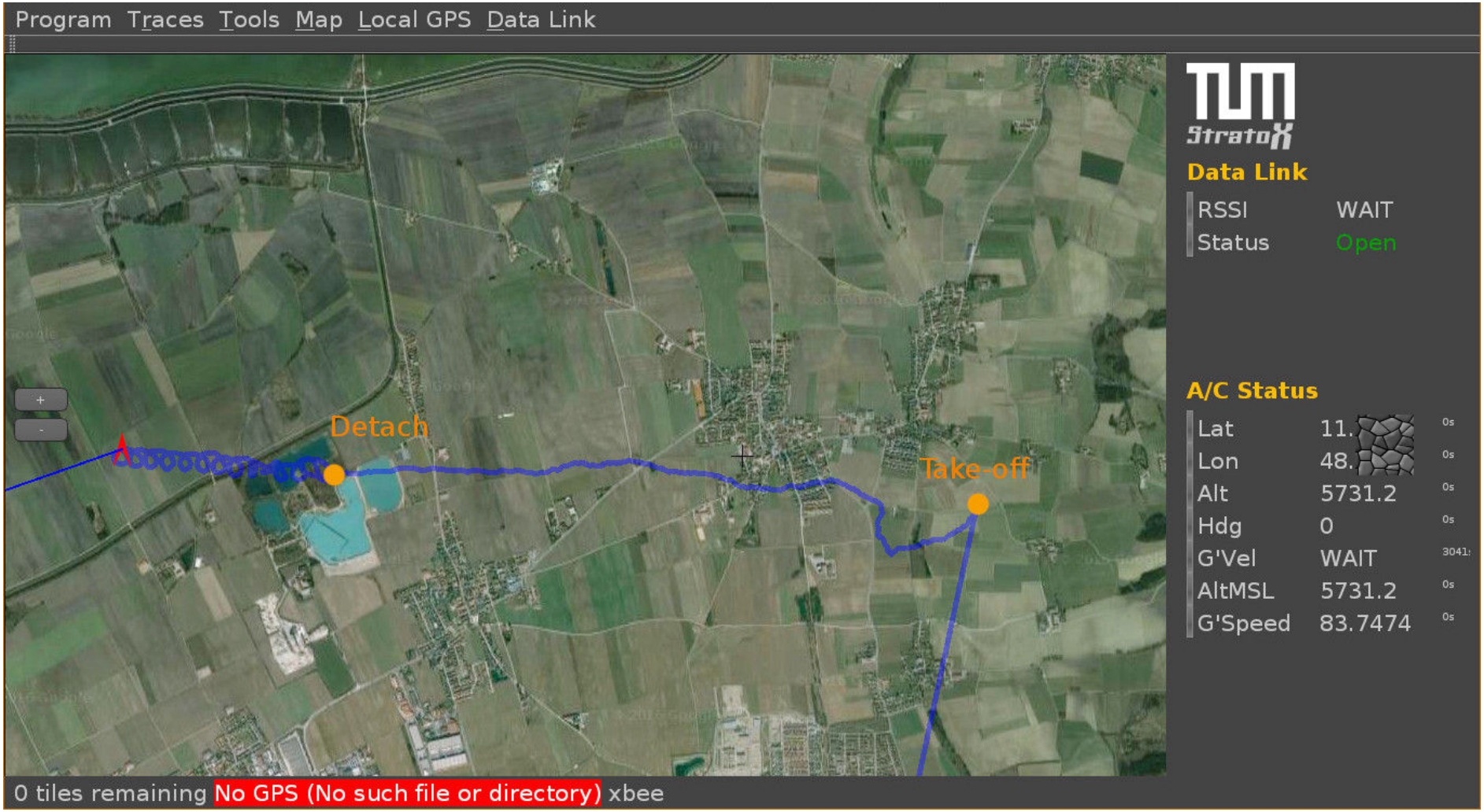
How we found defects

1. most by static analysis (each developer & nightly runs)
   - removed all ‘‘stupid bugs’’
   - identified missing requirements (saturation)
2. few by integration testing
   - masking defects during analysis / careless usage of verifier
   - ignoring failed proofs / wrong process
   - incorrect config of run-time system / violating prerequisite for analysis
3. none during operation?

Ready for Takeoff / En Route (Glider Launch)

Final Flight (1)


Program Traces Tools Map Local GPS Data Link
Take-Off
Data Link
RSSI WAIT
Status Open
A/C Status
Lat 11.
Lon 48.
Alt 6650.1
Hdg 0
G'Vel WAIT
AltMSL 6650.1
G'Speed 71.4131
0 tiles remaining No GPS (No such file or directory) xbee


Program Traces Tools Map Local GPS Data Link
Detach
Take-off
Data Link
RSSI WAIT
Status Open
A/C Status
Lat 11.
Lon 48.
Alt 5731.2
Hdg 0
G'Vel WAIT
AltMSL 5731.2
G'Speed 83.7474
0 tiles remaining No GPS (No such file or directory) xbee


2126m
1919m
2023m
2740m
2841m
3146m
3470m
3572m
3681m
3782m
3882m
3987m
4095m
4197m
4301m
4405m

Lost Contact

- navigation failure, but stable flight
- glider did not return back home
- glider could not be recovered
  - ironically: stand-alone backup localization device had failed
- How to find out what went wrong?

Incident Report (SPARK Forensics)

SPARK Forensics

- self-tests at startup had been passed
- launch and unhitch went according to plan
- Continuously flying **left circles** until landing:
  - stable flight is only possible with healthy airframe + controls
- Possible Explanations: another unexpected exception, or no GPS fix, or lost home position, or in-air reset with unsuccessful resume, or sensor error or numerical error, or, or, or ...

Luckily, high-level behavior of glider was encoded and verified in SPARK

Encoded High-Level Behavior (simplified)

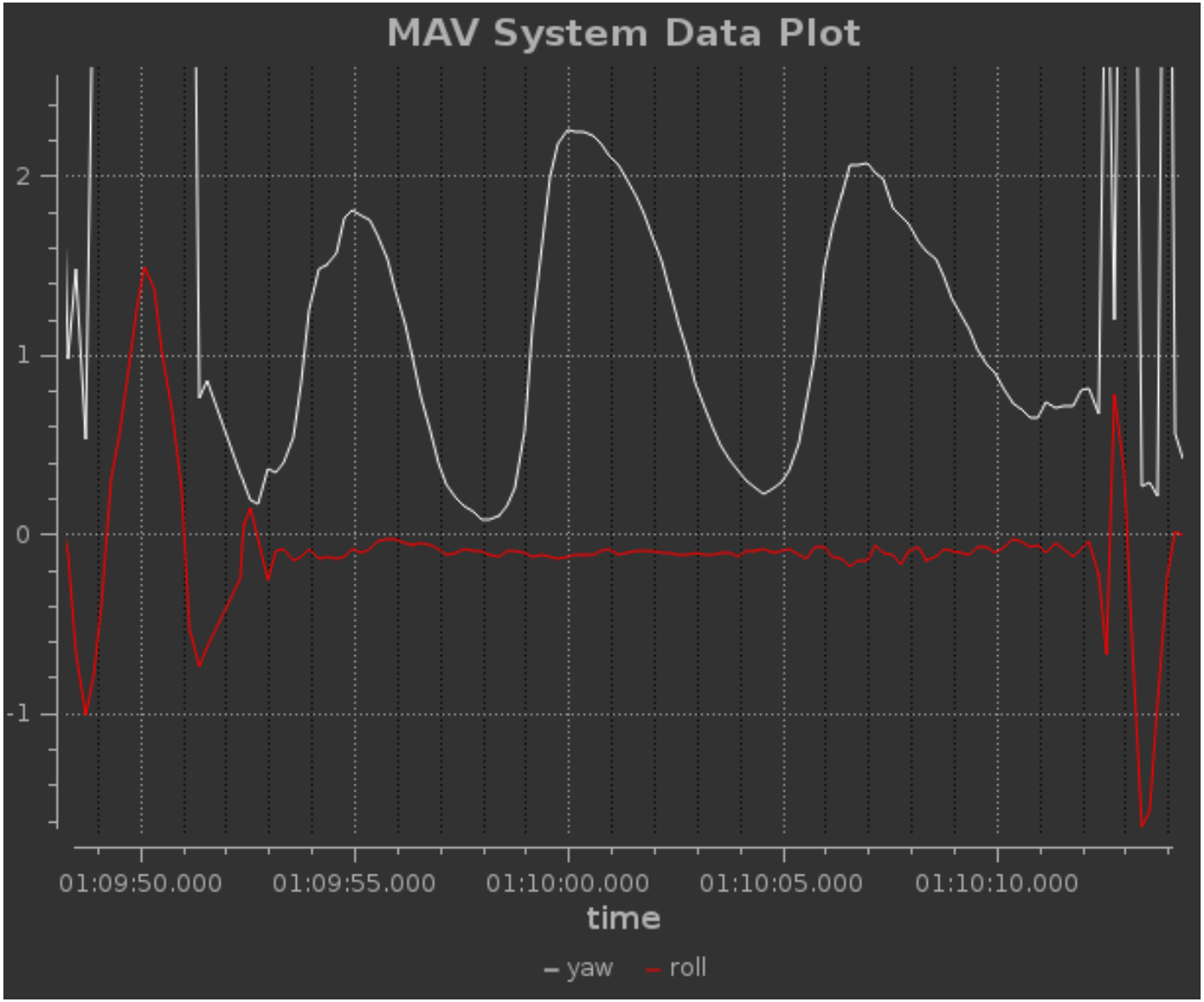
```
procedure Update_Homing with
  Global => (Input => (G_Object_Pos, G_Target_Pos),
             In_Out => (G_state), Output => Ctrl_Mode),
  Depends => (G_state => (G_state, G_Object_Pos, G_Target_Pos)) is
begin
   G_state.once_had_my_pos := G_state.once_had_my_pos or have_my_pos;
   if have_my_pos and then have_home_pos then
      if G_state.distance_to_home < Config.TARGET_ZONE_RADIUS then
         Ctrl_Mode := MODE_ARRIVED;
      else
         Ctrl_Mode := MODE_HOMING;
      end if;
   elsif have_home_pos and then (not have_my_pos and G_state.once_had_my_pos) then
      Ctrl_Mode := MODE_COURSEHOLD; -- temporarily lost nav => hold course
   else
      pragma Assert (not have_home_pos or not G_state.once_had_my_pos);
      Ctrl_Mode := MODE_POSHOLD; --  don't know where to go => hold position
   end if;
end Update_Homing;
```

```
procedure Compute_Target_Attitude with
  Global => (Input  => (G_state, Ctrl_Mode, G_Object_Att, Ada.Real_Time.Clock_Time),
             Output => (G_Target_Att)),
  Contract_Cases => ((Ctrl_Mode = MODE_COURSEHOLD) =>
                       G_Target_Att.Yaw = G_Target_Att_Prev.Yaw, -- roll open
                     (Ctrl_Mode = MODE_HOMING) =>
                       G_Target_Att.Yaw = G_state.course_to_home, -- roll open
                     (Ctrl_Mode in MODE_POSHOLD | MODE_UNKNOWN) =>
                       G_Target_Att.Roll = -Config.CIRCLE_TRAJECTORY_ROLL, -- yaw open
                     (Ctrl_Mode = MODE_ARRIVED) =>
                       G_Target_Att.Roll = Config.CIRCLE_TRAJECTORY_ROLL), -- yaw open
  Post => G_Target_Att_Prev = G_Target_Att;
```

Encoded High-Level Behavior

- enforcing exactly one out of four behaviors
  $\{$`MODE_HOMING`, `MODE_COURSEHOLD`, `MODE_POSHOLD`, `MODE_ARRIVED`$\}$
- conditions for circling left:
  - `MODE_POSHOLD`: only if no home position or never had GPS fix
    $\Rightarrow$ no, mission starts only with these (acoustic and optical pre-flight checks from glider)
  - `MODE_COURSEHOLD`: no, would fly straight $\Rightarrow$ glider had GPS fix
  - `MODE_ARRIVED`: no, would circle right $\Rightarrow$ no calculation error
  - `MODE_HOMING`: yes, could circle left
  - permanent in-air resets: extremely unlikely, as this causes twitches (trajectory smooth). Even if, last values would be restored from NVRAM.
- $\Rightarrow$ Glider was in homing mode. Only two explanations left:
  1. mag/compass broken (would have been detected by BIST)
  2. mag/compass provided unexpected output (e.g., ‘‘cannot find north’’)

Reviewing Previous Flight Logs...



Suggested explanation: Magnetic distortion in avionics bay

Debriefing (Conclusion)

Debugging – Reality

Ways of Finding Defects

1. most by static analysis (each developer & nightly runs)
   - removed all ‘‘stupid bugs’’
   - identified under-specification
2. few by integration testing
   - masking defects during analysis / careless usage of verifier
   - ignoring failed proofs / wrong process
   - wrong ‘‘verification fixes’’ (saturation) / incomplete specification
   - incorrect config of run-time system / violating prerequisite for analysis

Debugging – Reality

Ways of Finding Defects

1. most by static analysis (each developer & nightly runs)
   - removed all ‘‘stupid bugs’’
   - identified under-specification
2. few by integration testing
   - masking defects during analysis / careless usage of verifier
   - ignoring failed proofs / wrong process
   - wrong ‘‘verification fixes’’ (saturation) / incomplete specification
   - incorrect config of run-time system / violating prerequisite for analysis
3. *one* during operation
   - faulty but non-crashing behavior
   - missed during integration testing
   - unverified assumptions about sensor data
   - beyond context of source code

Looking back

- writing code takes more time in SPARK 2014 (cmp.: C)
  - but some agility can be applied to speed up progress (write first $\rightarrow$ fix critical $\rightarrow$ fix uncritical)
- very little debugging work
  - practically no exceptions during system testing
  - no working through cause-effect chains
  - no minimization of problem cases
  - no isolation of failure causes
  - no reproduction issues
- concentrate on getting functionality right which is not modeled in SPARK
  - effect of Kalman Filter
  - evolution of sensor data
  - underspecification (what happens after GPS goes silent?)

Conclusion

- SPARK 2014 and tools work very well
  - priceless execution semantics of annotations
  - many defects are avoided with almost no additional effort
  - no excuses for uninitialized variables
  - floating-point numbers are usable, but need some work (instantiate lemmas, refactoring of code, longer time to verify)
  - testing can be significantly reduced (not system testing)
  - high-level behavior can and should be encoded
- Ravenscar+SPARK: easy and effective multi-threading, separation of criticality, must-have
- not evaluated in detail: OO (not for us) and flow dependencies (for being a cousin of ‘‘const hell’’)
- code being released open source

Reached Parking Position – Questions?

(Do not re-install after landing)

To appear: *Development and Verification of a Flight Stack for a High-Altitude Glider in Ada/SPARK 2014*, M.Becker, E.Regnath, S.Chakraborty, Computer Safety, Reliability and Security (SAFECOMP), 36th International Conference, Trento, Italy, 2017.