Implementation and Analysis of the Nose-Gear Velocity Example in SPARK

Rod Chapman, Altran Praxis

Agenda

- Introduction
- Method
- Results
- Conclusions

Introduction

- Goals and Constraints
  - Do something useful and possibly interesting.
  - Do not attempt to solve every aspect of the problem.
  - Time available: very little, so focus on one or two aspects.

Introduction

- Observation
  - Two problems really:
    - A calculation problem – given inputs, calculate the “right” value of velocity.
    - A concurrency problem – dealing with the independent timing and phasing of the interrupt handler and the polling periodic task.

- Idea
  - *Don’t* try to solve both of these problems at the same time.
  - Aim for separation of concerns.

The Idea

- A simple implementation was constructed by Yannick Moy of AdaCore.
  - Written in SPARK, and subject to information flow analysis.
  - NOT subjected to formal verification for runtime errors or any other property.

- Goal: strengthen this implementation to develop a proof of type-safety for it. Record discoveries along the way.

Method

1. Read and analyse the requirements statement. Look for ambiguity, inconsistency and incompleteness.

2. Attempt type-safety proof for the original solution.

3. Analyse each failed proof, either:
    1. Strengthen the solution (e.g. add a precondition).
    2. Re-code it.
    3. Document an assumption.
    4. Question the requirements.

4. Repeat until proof complete (with a list of assumptions) and/or show-stopping bug is found.

5. Compare results with CSL's Certification Standard.

Reading the v1.0 Requirements

- (At this point, I hadn't read Jeff's pseudo-code, so I was starting with a relatively clean sheet…)
- Reading v1.0 (July) of the requirements statement…
- A few questions:
  - "16-bit counters" – this seems like an implementation-detail, not a requirement! Are 16 bits enough for the required range and precision?
  - What set (of integer values) is represented by each "counter"? Negative values perhaps?
  - What actually is the required range and precision of the answers? R1 calls for an "accurate estimate"…what's that?

Reading the v1.0 Requirements

- What does "increment" mean?
  - X + 1?
  - (X + 1) mod 65536?

- "The *circumference*...is also available...a compile time constant NG_WHEEL_*DIAMETER*."
  - Eh? Diameter and Circumference are usually different...

- The type (real, floating point, integer?) and units (mph, kmh?) of estimatedGroundVelocity are not specified.

Reading the v1.0 Requirements

- The “worked example”
  - Uses 3.14 for Pi – is this really accurate enough?
  - Says *Diameter* of the wheel is 22 inches…
  - Yields velocity in *miles per hour*.
- Let’s stick with these assumptions for now…
- Final comment: reading *requirements*, you should *not* be able to tell what the author’s favourite programming language is…

The Original Implementation

- Constructed by Yannick Moy at AdaCore.

- Makes some simplifying assumptions.
  - In particular, calculate instantaneous velocity in response to every “click” from the wheel every time.

  - Avoids issues of timing and relative phase and timing of the interrupt and the update function. Addresses the calculcation problem, but *not* the concurrency problem (yet…)

  - Simply compute the velocity from the time elapsed between each “click” assuming the sensor reliably generates a “click” for every rotation of the wheel.

The Original Implementation

```
-- Assume "counters" are integers in the range 0 .. 65535
-- with "modulo 65536" arithmetic operators.
type Counter_16_Bits is mod 2 ** 16;

-- Assume pre-defined "Float" has sufficient range
-- and precision for now.
subtype Distance is Float;  --  in inches
subtype Time is Float;   --  in millisecs
subtype Velocity is Float;  --  in mph
```

The Original Implementation

```
NGRotations : Counter_16_Bits;

NGClickTime : Counter_16_Bits;

Millisecs : Counter_16_Bits;

EstimatedGroundVelocity : Velocity;

NG_WHEEL_DIAMETER : constant := 22;

PI : constant := 3.14;

procedure ComputeEstimatedGroundVelocity;
--# global in     NGRotations, NGClickTime;
--#            out EstimatedGroundVelocity;
--#         in out State;
```

The Original Implementation

```
ThisNGRotations := NGRotations;
ThisNGClickTime := NGClickTime;

DistanceRolled := RotationToInch
  (ThisNGRotations - SavedNGRotations);

TimeElapsed := Float (ThisNGClickTime - SavedNGClickTime);

EstimatedGroundVelocity :=
   IpmsToMph (DistanceRolled / TimeElapsed);

SavedNGRotations := ThisNGRotations;
SavedNGClickTime := ThisNGClickTime;
```

- Easy huh? Well…err….no…

Proof of The Original Implementation

- Proof using the SPARK Toolset for “type safety” (aka “no runtime errors”, “no exceptions”) yields 16 Verification Conditions, only 5 of which (31.25%) are proved automatically by our Theorem Prover.
- Not very impressive so far…
- What’s going on?

Iteration 2: Implementation-Defined constants and types

- By default, the SPARK toolset does *not* assume *anything* about the range of values and/or the precison of the pre-defined types like Integer and Float.
  - These are *implementation-defined* in SPARK, so the VC-Generator can be told the "right" values with a configuration file.
  - These have to be checked very carefully to make sure they're valid for the *target hardware.*

Iteration 2: Implementation-Defined constants and types

- Let’s assume 32-bit signed 2’s complement “Integer”, and IEEE 32-bit “Float”…
- Now we get 11 of 16 VCs proved (68.75%)
- Our normal rule-of-thumb is that well-written SPARK code should score >95% VCs proved automatically for type-safety, so still some room for improvement…

Iteration 3: Data Validity

- Who says that the pattern of bits you get from an input device is really a valid value according to the type system of the programming language?
  - This certainly isn't the case for floating-point values, for example, where you might read a NaN value.
  - Even if you say an integer type is N-bits, a compiler and/or hardware might read more than N bits – enormous care is required.

- So…The VCG makes *no assumption at all* about the validity of external data unless told otherwise.

Iteration 3: Data Validity

- In this case, though, we're OK. *Every* pattern of 16 bits is a valid value for type Counter_16_Bits, assuming the generated code and hardware really do read exactly 16-bits.

- We tell this to the VCG with an additional contract.

- Now we get 13 VCs (81.25%) proved automatically.

Iteration 4: Numeric ranges

- Normal coding practice in SPARK says that you should *never* use the pre-defined types like "Float" anyway, since their range and precision are implementation-defined.
  - You should say what you want!

- So…what range should Velocity have?
  - What's the smallest observable velocity?
  - What's the largest?
  - What about negative value?
  - How is "zero velocity" handled?

Iteration 4: Numeric ranges

- This train of thought led to many discoveries…
- Let's assume the highest observable velocity is 1 wheel rotation in 1 millisecond.
- Important constant
  - 1 inch per millisecond = 56.818181 miles per hour.
- Therefore, MaxV = Diameter * Pi * 56.818181

  = 3924.99 mph (which we assume is beyond the physical capability of the airframe when it's on the ground…)

Iteration 4: Numeric ranges

- What about the minimum observable velocity?
- Let's assume 1 wheel rotation in 65535 milliseconds (65.535 seconds).
- Wheel circumference is 22*Pi = 69.08 inches, so this is about 1 inch per second.
- MinV = 0.0598863 mph.

- Realization: below MinV, this system is unreliable and not to be trusted. Zero velocity is not measurable at all.

Iteration 4: Numeric ranges

- Realization: below MinV, this system is unreliable and not to be trusted. Zero velocity is not measurable at all.

- Example: click N occurs at T1 = 0. Click N+1 occurs 65538 milliseconds later. The “16 bit” clock now reads “2”, so we’re doing 1962 mph, right? ☹

- Observation: “16 bit” timer is insufficient and/or just plain wrong for this problem.

Iteration 4: Numeric ranges

- What about negative velocity?
- Can the plane roll backwards?

- Well..probably – how many times have you parked your car facing uphill and momentarily forgotten to put the hand-brake on?

- Does the sensor “click” on a backwards-going wheel turn? It seems to be impossible to tell the difference…

Iteration 4: Numeric ranges

```
Ipm_To_Mph : constant := 56.818181;

Slowest_Measureable_Velocity_Ipm : constant Velocity :=
    Velocity (Circumference / Inches (Counter_16_Bits'Last));

Fastest_Measureable_Velocity_Ipm : constant Velocity :=
  Velocity (Circumference);

subtype Velocity_Ipm is Velocity range
   Slowest_Measureable_Velocity_Ipm ..
   Fastest_Measureable_Velocity_Ipm;

subtype Velocity_Mph is Velocity range
    Slowest_Measureable_Velocity_Ipm * Ipm_To_Mph ..
    Fastest_Measureable_Velocity_Ipm * Ipm_To_Mph;
```

Iteration 4: Numeric ranges

- Numeric accuracy…
- Now we know the range of Velocity_Mph, we can determine the density of the model numbers of type Float in that range, and decide if its precision is good enough…
- Let's not go there today, though…

Iteration 4: Numeric ranges

- With a bit of further simplification, this implementation yields 14 VCs, of which 13 (92.8%) are proven automatically.

- The one left, though, is troublesome, arising from the line of code:

```
V_Ipm := Velocity_Ipm (Float (DistanceRolled) / TimeElapsed);
```

- This expression needs to yield a value in the range of Velocity_Ipm (not too big, not too small…), must not overflow, and must not divide by zero.

Iteration 4: Numeric ranges

- This observation open a whole can of worms:
  - Perhaps velocities and/or accelerations should be filtered, “sanity checked” or bounded in some way?
  - How should it deal with acceleration from standing start and deceleration towards zero?
  - Can the sensors exhibit stuck-at faults, so it might appear that no time is passing or no distance being travelled?
  - Should the system actually maintain a moving-average of velocity based on the last N readings? What should N be?
  - And so on and so on…

Reading the v1.1 Requirements

- November 7th: find and read the v1.1 requirements spec.
- See comments earlier.
- Plus…v1.1 re-enforces the tricky matter of dealing with the unpredictable phase and timing of the update function relative to the hardware interrupt.
- Introduces the "estimatedGroundVelocityIsAvailable" flag where "True" seems to be "non-zero value".
  - Why is a requirements statement polluted with low-level and seemingly language-specific detail?

Iteration 5: Robustness

- We need to guard the division suitably to avoid division-by-zero and range overflow.
  - DistanceRolled and TimeElapsed must not be zero *and*
  - The number of wheel rotations registered must be less than or equal to the number of milliseconds since last click. For example:
    - 1 rotation in 2 milliseconds is OK.
    - 2 rotations in 1 millisecond is not allowed, since we'd be going too fast…

Iteration 5: Robustness

```
Rotations     := ThisNGRotations - SavedNGRotations;
Clicks_Elapsed := ThisNGClickTime - SavedNGClickTime;

if (Clicks_Elapsed /= 0) and
  (Rotations in Valid_Non_Zero_Rotation_Count) and
  (Rotations <= Clicks_Elapsed) then

   Distance_Rolled := RotationToInch (Rotations);
   Time_Elapsed    := Time (Clicks_Elapsed);

   V_Ipm := Velocity_Ipm (Float (Distance_Rolled) / Time_Elapsed);
   EstimatedGroundVelocity := IpmToMph (V_Ipm);
   EstimatedGroundVelocityAvailable := True;

   SavedNGRotations := ThisNGRotations;
   SavedNGClickTime := ThisNGClickTime;
else
   EstimatedGroundVelocity := Velocity_Mph'First;
   EstimatedGroundVelocityAvailable := False;
end if;
```

Iteration 5: Robustness

- Now we get 17 VCs.
  - 16 proven fully automatically.
  - 1 requires a user-defined lemma to “help” the prover.

- This gives us a solid understanding of what’s required to solve the calculation problem and the assumptions that we’ve made along the way.

The Concurrency Problem

- Not really had enough time to address this.

- But…this is a common pattern:
  - Interrupt handler (low latency, fast, non-blocking) samples raw inputs in response to click, and deposits in a “protected” (mutually exclusive access) buffer.
  - A “periodic” task samples latest data from buffer and calculates current estimated velocity. This is placed in another protected state to be visible to other tasks.

- BUT..still lots of interesting issues. How much history to store? What if sensors fail? Averaging or filtering? Etc. etc.

Comparison with the CSL Certification Standard.

- SPARK generates evidence that *directly* addresses the following objectives:
  - M (Source code is accurate)
  - N (Source code complies with standards)
  - R (Object code robustness)

Comparison with the CSL Certification Standard.

- SPARK also *indirectly* contributes to the following objectives (i.e. makes them easier, prevention of defects and/or reduction in repetition of steps)
  - K (Source code complies with the software requirements)
  - O (Source code is compatible with the target computer)
  - P (Output of integration is complete and correct)
  - S (Object code is compatible with the target computer)
  - V (Verification procedures are correct)

Comparison with the CSL Certification Standard.

- On the C130J MC project (DO-178B Level A), the main contribution of SPARK was *indirect*.
  - Reduction of pre-test defect rate through prevention of common errors and exposure of specification defects (e.g. incomplete Parnas tables).
  - See Sutton & Middleton "Lean Software Strategies" book.

Comparison with the CSL Certification Standard.

- But…with DO-178C
  - What if we qualify the SPARK toolset as a “super verification tool”?
    - What credit can we take?
    - What subsequent process adjustments can we make?
  - Is SPARK a “formal notation”?
    - We think “Yes”
  - Does static analysis of SPARK exhibit “property preservation” (relative to a realistic set of real-world assumptions)?
    - We hope so! This is a principal design goal of SPARK…
- We hope to work with customers to address these issues soon.

Further work…

- Some ideas for things to do:

1. Complete the implementation using RavenSPARK tasking with an interrupt handler, protected objects and a periodic task that runs the update function. Investigate and report on concurrency issued raised.

2. Translate Jeff's pseudo-code into SPARK and analyse as before.

3. Start again from scratch. Re-consider the design of the timer hardware for a start…

Conclusions

- A formal implementation style *forces you to think* really hard about an implementation, its properties, and the assumptions on which your "proof" is built.

- You are forced to write down and possibly question assumptions.

- A *sound* verification tool forces you to think about *all* the corner cases.

Altran Praxis Limited

20 Manvers Street
Bath BA1 1PX
United Kingdom
Telephone: +44 (0) 1225 466991
Facsimile: +44 (0) 1225 469006
Website: www.altran-praxis.com

Email: rod.chapman@altran-praxis.com