Towards an industrial use of FLUCTUAT on safety-critical avionics software

David Delmas[1], Eric Goubault[2], Sylvie Putot[2], Jean Souyris[1], Karim Tekkal[3] and Franck Védrine[2]

1. Airbus Operations S.A.S., Avionics & Simulation Products
2. CEA LIST, Laboratory for the Modelling and Analysis of Interacting Systems
3. Digiteo

FMICS 2009, Eindhoven

Outline

- Industrial context at Airbus
- The IEEE-754 standard
- The FLUCTUAT static analyser
- Automating the accuracy analysis of basic operators
- Conclusions and future work

Industrial context at Airbus

- Formal verification of certified avionics software
  - for certification credit on several projects
    - StackAnalyzer & aiT WCET (AbsInt)
    - CAVEAT (CEA-LIST)
  - among next candidates for industrial transfer
    - ASTRÉE (ENS + AbsInt)
    - FLUCTUAT (CEA-LIST)
- Numerical computations in control programs
  - control algorithms are
    - correct by design in the realm of (ideal) real numbers
    - implemented with (finite-precision) floating-point numbers
  - need to ensure stability & accuracy of control programs
    - freedom from run-time errors
    - imprecision should be negligible compared to computer I/O (sensors, actuators: imprecision bounds are known)

Accuracy and sensitivity analyses on control programs

- Model-based approach
  - automatic code generation from SCADE/Simulink models
  - library of basic (numerical) operators
- Part of the system accuracy assessment:
  for each operator, sound & precise analysis of
  1. potential loss of accuracy
  2. potential propagation of errors from inputs to outputs (i.e. sensitivity)
  3. behavior of the underlying algorithm
     - bounding the number of iterations of an iterative algorithm
     - functional proof (both in real and floating-point numbers)
- Activities conducted on Airbus's fly-by-wire computers
  - since floating-point numbers are embedded (1990s)
  - combination of testing and sophisticated intellectual analyses
  - but costly and error-prone
  - to be partly automated with FLUCTUAT

Floating-point numbers (IEEE 754 norm)

- Limited range and precision : potentially inaccurate results or run-time errors
- A few figures for simple precision normalized f.p. numbers :
  - largest $\sim 3.40282347 * 10^{38}$
  - smallest positive $\sim 1.17549435 * 10^{-38}$
  - max relative rounding error $= 2^{-23} \sim 1.19200928955 * 10^{-7}$
- Consequences:
  - potentially non intuitive representation error:
    $\frac{1}{10} = 0.0001100110011001100110011 \cdots$ (binary)
    $\Rightarrow float(\frac{1}{10}) = 0.100000001490116119 4 \cdots$ (decimal)
  - absorption : $1 + 10^{-8} = 1$ in simple precision float
  - associative law not true : $(-1 + 1) + 10^{-8} \neq -1 + (1 + 10^{-8})$
  - cancellation: loss of relative accuracy if subtracting close nbs

In real world : costly or catastrophic examples

- 25/02/91: a Patriot missile misses a Scud and crashes on an american building : 28 deads.
  - the missile program had been running for 100 hours, incrementing an integer every 0.1 second
  - but 0.1 not representable in a finite number of digits in base 2
  - Drift on 100 hours $\sim 0.34s$: location error $\sim 500m$
- Explosion of Ariane 5 in 1996 (conversion of a 64 bits float into a 16 bits integer : overflow)
- An index of the Vancouver stock exchange in 1982
  - truncated at each transaction : errors all have same sign
  - within a few months : lost half of its correct value
- Sinking of an offshore oil platform in 1992 : inaccurate finite element approximation

Validation of numerical programs

- Conception designed in real numbers: what is the impact of a finite precision implementation, what is a correct program using floating-point numbers?
  - No run-time error, such as division by 0, overflow, etc
  - The program does compute something "not too far" from what is expected (=the result computed in real numbers)
  - No problematic control-flow difference between real and floating-point computation (same nb of iterations)
  - Can we also prove the algorithm correct (method error) ?

Householder scheme for square root computation

Execution of Householder scheme

Our choice for validation

- Static analysis by abstract interpretation: results are sound, but possibly over-approximated
- We consider the program in two ways:
  - a specification: using the idealized semantics in real numbers
  - an implementation: using the actually executed semantics, in IEEE 754 floating-point numbers
- We bound ranges of values and errors due to the use finite precision, for all variables
- We decompose the errors on their causes (LOC)

FLUCTUAT

- Takes source C code (most of ANSI C, except union types and dynamic memory allocation most notably),
- With assertions (for instance range of values and imprecision on input, but also range of gradient of evolution of values)
- Parameterizable (precision/time, domains, architecture dependent features, etc)
- Gives, fully automatically, characterization of ranges/errors, and describe the origins of errors: identification of pieces of code with numerical difficulties
- But also, in some cases, weak functional proof of algorithms
- Is/has been used for a wide variety of codes (automotive, nuclear industry, aeronautics, aerospace) of size up to about 50000 LOCs (on laptop PCs 1Gb)

FLUCTUAT static analyzer: models float as real + error

```
float x,y,z;
x = 0.1; // [1]
y = 0.5; // [2]
z = x+y; // [3]
t = x*z; // [4]
```

$$
\begin{aligned}
f^x &= 0.1 + 1.49e^{-9}\ [1] \\
f^y &= 0.5 \\
f^z &= 0.6 + 1.49e^{-9}\ [1] + 2.23e^{-8}\ [3] \\
f^t &= 0.06 + 1.04e^{-9}\ [1] + 2.23e^{-9}\ [3] - 8.94e^{-10}\ [4] - 3.55e^{-17}\ [ho]
\end{aligned}
$$

$\Rightarrow$ Then abstraction for each term (real value and errors)

Sound abstraction based on Affine Arithmetic

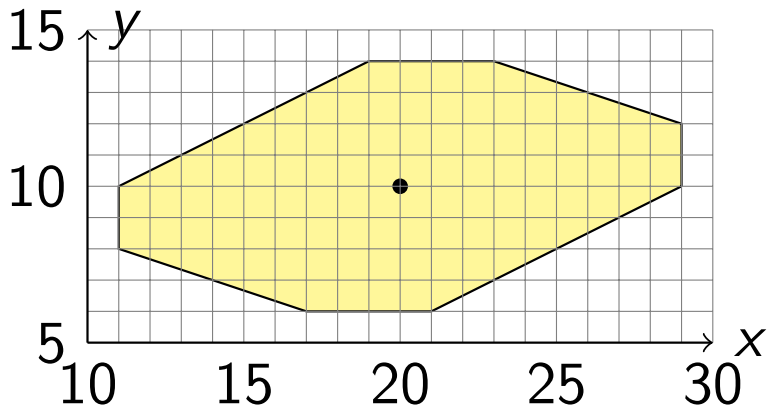
- The *real value* of variable x is represented by an affine form $\hat{x}$:

$$\hat{x} = x_0 + x_1\varepsilon_1 + \ldots + x_n\varepsilon_n,$$

where $x_i \in \mathbb{R}$ and the ε_i are independent symbolic variables with unknown value in $[-1, 1]$.

- Sharing ε_i between variables expresses *implicit dependency*: concretization as a zonotope

$$\begin{aligned} \hat{x} &= 20 - 4\varepsilon_1 + 2\varepsilon_3 + 3\varepsilon_4 \\ \hat{y} &= 10 - 2\varepsilon_1 + \varepsilon_2 - \varepsilon_4 \end{aligned}$$



Abstract domain based on affine arithmetic

- *Assignment* of a variable x whose value is given in a range $[a, b]$ introduces a noise symbol ε_i :

$$\hat{x} = \frac{(a+b)}{2} + \frac{(b-a)}{2}\varepsilon_i.$$

  - functional abstraction: link to the inputs via the noise symbols, allowing sensitivity analysis and worst case generation

- *Addition* is computed componentwise (no new noise symbol):

$$\hat{x} + \hat{y} = (\alpha_0^x + \alpha_0^y) + (\alpha_1^x + \alpha_1^y)\varepsilon_1 + \ldots + (\alpha_n^x + \alpha_n^y)\varepsilon_n$$

- *Non linear operations* : approximate linear form (Taylor expansion), new noise term for the approximation error
- Efficient join operator (on-going work for a better meet operator)

Back to the Householder scheme

Householder

Automating the accuracy analysis of basic operators with FLUCTUAT

- Types of codes to be analysed
  - C functions or macro-functions: polynomial approximations, interpolators, digital filters, signal integrators, etc.
  - challenges for precise static analysis
    - very large input ranges, to address all operational contexts
    - (implicit) unbounded loop for digital filters
    - sophisticated bitwise operations
      (WCET, determinism & traceability constraints)
      Example of challenging feature for (sound & precise) static analysers: a conversion from `double` to `int` avoiding unexpected compiler-generated constant sections

```
int i, j;
double f, Var, Const;
...
j=0x43300000; ((int*)(&Const))[0]=j; ((int*)(&Var))[0]=j;
j=0x80000000; ((int*)(&Const))[1]=j; ((int*)(&Var))[1]=j^i;
f = (double)(Var-Const);
```

Results obtained for every basic operator

- Input ranges guaranteeing absence of run-time errors
- Analysis results proving that each operator
  - can introduce only negligible rounding errors
  - cannot significantly amplify errors on inputs
- Compared to the legacy method (intellectual analysis)
  - the analysis process is faster & easier
  - the figures obtained are
    - usually of the same magnitude
    - sometimes much less pessimistic (up to one order of magnitude)
- In some cases, a functional proof of the underlying algorithm is also achieved (in real and floating-point numbers)

Analysing a polynomial approximation of the arctangent

Arctangent

Conclusions & future work

- Accuracy analysis of basic operators partly automated
  - very precise results
  - rather systematic analysis process
- Next step: industrial use within operational development teams
  - industrialisation of the tool ongoing
  - qualification as a verification tool wrt. DO-178B/C?
- Future work
  - interoperation with ASTRÉE
  - beyond local analyses of basic operators
    - assess the accuracy of independent system-level functions
    - analyses of SCADE sub-models (servo-loops)