Some future challenges in the validation of control systems

E.Goubault[1], M. Martel[1], S. Putot[1]

1: DTSI/SOL, CEA/Saclay, 91191 Gif-sur-Yvette

Abstract: Starting with our work on the characterization of the imprecision error in programs using floating-point-numbers, by abstract interpretation, this paper shows that there are numerous perspectives, if one wants to fully qualify the numerical quality of control systems, as found in the aeronautical and automotive industry, for instance. Some very common functions (e.g. integrators) are hard to statically analyse, because their numerical correctness depend on a fine-grained specification of the classes of input signals they handle. This gets even more complex in the case of e.g. PID controllers, which interact in closed loop with an external environment, since their input signals are in part the consequence of their own computation, similarly for the imprecision errors. We show examples of non-trivial bad and good numerical behaviours, discuss the results of our methods, and present our current research directions, that should hopefully help characterize the imprecision error of such control systems.

Keywords: floating-point numbers, imprecision errors, static analysis, abstract interpretation.

1. Introduction

The research work in computer systems at CEA originally started with the N4 generations of nuclear plants controlled by software (and not hardwired as previous generations). In particular, the emergency stop in such nuclear plants is the most critical part, which required huge validation efforts, fostering new research both in conception and verification methods, and still accounting for part of our current developments, under the auspices of IRSN. Our validation activity naturally applies to other critical control systems, such as the ones in the automotive and aeronautics industry. We will most notably take as an example of our current research the Fluctuat tool, currently evaluated and used both by IRSN and Airbus.

Fluctuat is a static analyser by abstract interpretation, which helps determine the discrepancy in the numerical computations in a control system, due to the use of an imperfect arithmetic, the IEEE 754 floating-point numbers (and more recently, fixed-point numbers as well), instead of using the ideal real numbers. It takes as input a piece of software (written in ANSI C, or in assembler, for the TMS320C3X), some assertions describing the range and precision of potential numerical inputs to this program, and gives as a result an estimate of the range and precision of all variables of the program, at some location in the program, for all possible inputs as specified, and without executing it. Furthermore, this estimate, as guaranteed by the general theory of abstract interpretation [7], is "sure", meaning that it is always an over-approximation of the set of possible values and imprecision errors that may arise during all potential (maybe infinite) executions of the program. Both the underlying theory and the tools themselves are described in [2], [4], [1], [3]. They have been successfully applied to some representative industrial control systems, but these case studies lead us to consider new challenges, both on the practical and theoretical sides.

2. FLUCTUAT

2.1 The tool

The aim of Fluctuat is to either detect automatically a possible catastrophic loss of precision, and its source, or to prove that the precision of all computations remains in an acceptable range.

Indeed, the origin of the main losses of precision is most of the time very localized. Fluctuat relies on semantics that decompose the error between the results of the same computation achieved respectively with floating-point and real numbers, in a sum of error terms corresponding to the elementary operations of this computation, and a higher order term, most of the time negligible, that agglomerates higher order errors. We give in this section a short overview of how floating-point operations are interpreted with the first semantics we implemented, based on this idea.

Let F be either the set of simple or double precision floating-point numbers. Let $\uparrow_o : R \to F$ be the function that returns the rounded value of a real number, with respect to the rounding mode *o*.

Then we define the function $\downarrow_o : R \to F$ that returns the round-off error by

$$\forall r \in R, \quad \downarrow_o (r) = r - \uparrow_o (r).$$

Assume that the control points of a program are annotated by unique labels $l \in L$, and that $\Im$ denotes the union of L and a special word hi used to denote all terms of order higher or equal to 2. We

represent a variable *x* at some point in the program, by a sum

$$x = f^x + \sum_{l \in \Im} \omega_l^x \varepsilon_l \ .$$

In this equation, f^x is the floating-point number approximating the value of *x*. A term $\omega_l^x \varepsilon_l$ denotes the contribution to the global error of the first-order error introduced by the operation labelled *l*, $\omega_l^x \in R$ being the value of this error term and ε_l a formal variable labelling operation *l*.

The result of an arithmetic operation $\bullet^{l_i}$ contains the propagation of existing errors on the operands, plus a new round-off error term $\downarrow_o (f^x \bullet f^y)\varepsilon_{l_i}$. For addition and subtraction, the errors are added or subtracted component-wise :

$$x +^{l_i} y = \uparrow_o (f^x + f^y) + \sum_{l \in \Im} (\omega_l^x + \omega_l^y)\varepsilon_l + \downarrow_o (f^x + f^y)\varepsilon_{l_i}$$

The multiplication introduces higher order errors:

$$x \times^{l_i} y = \uparrow_o (f^x f^y) + \sum_{l \in \Im} (f^x \omega_l^y + f^y \omega_l^x)\varepsilon_l$$
$$+ \sum_{l_1 \in \Im, l_2 \in \Im} \omega_{l_1}^x \omega_{l_2}^y \varepsilon_{hi} + \downarrow_o (f^x f^y)\varepsilon_{l_i}.$$

The analyser is built on this idea, using intervals to get computable supersets of the coefficients, in an abstract interpretation framework [7]. The use of intervals allows on one hand to consider sets of values for variables, and on the other hand to include the rounding errors committed by the analysis. Indeed, static analysis consists in computing some properties of a program without executing it, for possibly large or infinite sets of inputs. Here, we compute a superset of all possible values f^x and errors ω_l^x for each variable *x* at any iteration of the loops, on the nodes of the programs to analyze. These computations are implemented using MPFR [5], a library that allows floating-point computations with arbitrary precision.

The analyser has been finely tuned: it comprises an alias analyser, specific fix point iteration schemes, precise widening operators, mechanisms for automatically subdividing input interval values etc.

Modular integer arithmetic is also considered, as well as casts between floating-point and integer, and bitwise operations on integers. Potential error terms are propagated between floating-point and integers.

A language extension is understood by the analyser, that allows the user to specify ranges of possible values and errors of inputs (and soon, the ranges of the gradient of values over time), instead of giving them fixed values.

Finally, the analysis relying on this domain does not use correlation between variables, and this may lead to large over-approximations. We thus have recently proposed weakly relational domains based on the same idea of keeping track of the origin of errors, but that use linear correlations between variables [20]. The results on the example presented in the next section were obtained with this relational analysis for the computation of values. The relational analysis for the computation of errors is being currently implemented.

2.2 Examples

Consider for example the following piece of code that computes the inverse of an input by a Newton iterative method.

```
double xi, xsi, A, temp;
signed int *PtrA, *Ptrxi, cond, exp, i;
double epsilon = e-10 ;
A = __BUILTIN_DAED_DBETWEEN(20.0,30.0);
PtrA = (signed int *) (&A);
Ptrxi = (signed int *) (&xi);
exp = (signed int) ((PtrA[0] &
0x7FF00000) >> 20) - 1023;
xi = 1; Ptrxi[0] = ((1023-exp) << 20);
cond = 1; i = 0;
while (cond) {
  xsi = 2*xi-A*xi*xi;
  temp = xsi-xi;
  cond = ((temp > epsilon) || (temp < -
epsilon));
  xi = xsi;
  i++;
}
```

The special assertion

```
A = __BUILTIN_DAED_DBETWEEN(20.0,30.0)
```

tells the analyzer that double precision input A can take its value between 20.0 and 30.0. Then the operation

```
PtrA = (signed int *) (&A)
```

casts the double precision number A into an array of two integers. Then, the exponent of the input is got from the first integer of the array, by bitwise operations. Thus an initial estimate of the inverse, necessary for a good convergence of the Newton algorithm, is got from the exponent. Then a non-linear iteration is computed until the difference between two successive iterates is bounded by epsilon.

In the framework of embedded systems, it is very important to qualify the behaviour of such an algorithm for all possible inputs. The first crucial point is to bound the possible number of iterations for any input in a range (in order to satisfy real-time constraints). And the second point that may be of importance is to see if the termination criterion is sensible, in the sense that if it is too low, the precision required may be obtained on the floating-point result, but not for the real result.

For example here, for all possible inputs between 20.0 and 30.0, our analyser finds that the algorithm always terminates in a number of iterations between 5 and 9, and states that the floating-point value of the output is in the interval [3.333333e-2,5.000000e-2] with a maximum error between the real and the floating-point computation in [-4.21443e-13,4.21443e-13]. We can note that the analyzer does not overestimate here the number of iterations, as executions show that respectively 5 iterations with an input equal to 20.0, and 9 iterations with an input equal to 30.0, are needed. Also, for the values, their bound indeed correspond to a close approximation of the inverse of the inputs. As for the error, it is for the moment slightly over-estimated, but we are currently working on a relational domain that will improve the results.

It should be noted that this error is not the absolute error between the result and the real value of the inverse. Fluctuat bounds the error due to the use of floating-point numbers, and cannot consider the error due to the algorithm itself, which for example here gives only an approximation of the inverse of the input.

The language of assertions understood by Fluctuat also allows to perturb the input with an initial error. For example if the input now has an error between –e-5 and e-5, then we get that the output has its error in [-9.95e-4, 9.94e-4] so the initial error is propagated without particular amplification, as the algorithm is stable (even if again the error is over-estimated).

In this short example where errors come from a few lines, we only deal with value and global error, but for larger scale errors, the error graphs showing the origin of the main errors in the source code is useful in order to look more precisely at these few problematic lines.

Let us now examine the same example but using simple precision floating-point numbers instead of double precision. The analyser does not manage to prove convergence of the algorithm in a finite number of iterations, the number of iterations obtained is possibly infinite. However, this may either be due to the fact that the algorithm indeed cannot converge, or to the fact that the precision of the analysis is not sufficient. In order to have a more refined insight on that point, a possibility is to use a special mode of the analyzer we call symbolic execution. This mode allows to see the behaviour (evolution of value and errors), still with our abstract semantics, but for one particular value of the input (instead of a set of inputs like intervals), potentially perturbed by an error interval. The symbolic execution is also often less costly than static analysis. We have thus tried symbolic execution for 1000 input values in the range [20.0,30.0], and for the first 517 values tried, indeed the number of iteration remains bounded in [5,7]. However, for the 518^{th} value, A = 25.18, the algorithm does not converge, and the difference between two successive iterates alternates between -3.725290e-09 and 3.725290e-09. This behaviour was confirmed by actual execution on machine (decomposing every arithmetic operation in order to avoid the use of registers for intermediate computations).

Moreover, even in cases when the algorithm seems to converge properly, the accuracy of the result may be questionable : for example, for input A = 25.46999931335449219, the algorithm converges in 6 iterations, and the difference between two floating-point iterates is zero. But the analyzer also gives the information that there is an rounding error on this difference equal to 2.0199e-09, which is 20 times greater than the stopping criterion epsilon. One can thus wonder at the meaning of the criterion in this case.

Finally, if we relax the stopping criterion for example to epsilon=1e-7, then Fluctuat is able to find that the algorithm will always converge with between 4 and 12 iterations.

3. The complexity of control systems

Fluctuat mostly considers as for now only one part of the complexity of the problem. Typically, the inputs to the programs we want to check come, at some point, from an external input. As for now, we only deal with the interface between the physical and the software worlds at the level of the discretization of physical quantities (say, input from analogical sensors, measuring physical quantities such as speed, acceleration, position in space, temperature, etc.), which are made of real numbers in general, into integers (through quantization) or fixed-point or floating-point numbers. This is just one part of the

problem: one also has to consider the discretization in time, or sampling of the data (as considered in "hybrid systems" theory [8] but in general to a lesser extent than what we are planning to do). As for now, this is done in a rather simple way, partly because we mainly deal with synchronous systems, and partly because most of the systems we have been studying are somehow quite robust to the types of signal they can handle.
For instance, in the previous example, we specified the range of the values and errors of inputs, and this was enough to get a good estimate of the numerical quality of the code. In the very near future, we will also have assertions to prescribe the range of the gradient of the inputs over time, which is a first step towards a better formalisation of the physical environment.

This is quite new to static analysis, since the need has not been felt so much up to now. Typically, as reported in Astrée's experiments (static analyser for run-time errors developed by ENS and X [6]), simple range assertions on cyclic inputs, not even giving some constraints on their time evolution, can be sufficient for proving absence of (software) bugs.

In the case of precision analysis, this is not true for even simple pieces of code, typical of components in control systems. Integrators are such components. They generally take as input a value, add it up to a current value, and use thresholds to limit the value they compute. Integrators are called in general with new inputs over time, cyclically. Indeed, it is very simple to prove (automatically, in a static analyser, for instance using the interval abstraction) that the variable containing the result will not overflow, if the threshold mechanism is well designed. Now, when it comes to the potential drift between the floating-point and the real values, things are much more difficult: imagine that the floating-point value representing the input signal always has a negative bias with respect to its actual physical value. This bias might add up substantially and lead to a very important imprecision error (of the order up to the value of the threshold itself). The difficulty is in general that the "real signal" has to be known quite precisely in order to find out that the rounding (quantization) and sampling introduce, for instance, a zero average bias. We give and explain below examples of such phenomena, and ideas about lines of research concerning static analysis in such contexts:

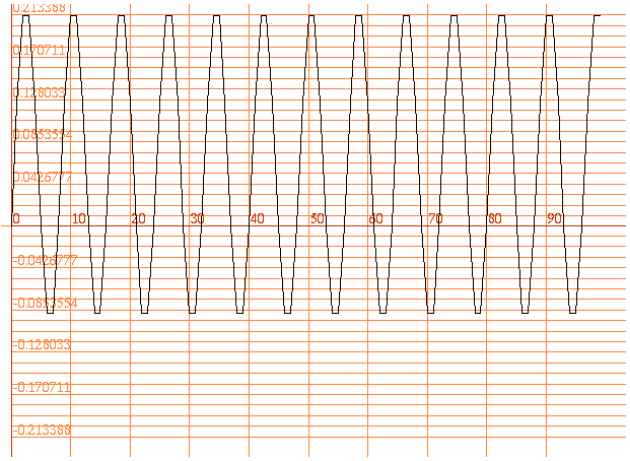
```
#define SUP 20
#define INF -20
#define h 1/8.0
#define N 100

static float intgrx=0.0;

void intgr(float xi) {
  intgrx += xi*h;
  if (intgrx > SUP)
    intgrx = SUP;
  if (intgrx < INF)
    intgrx = INF;
}

float f(int i) {
  …
}

void main() {
  int i;
  for (i=0;i<N;i++)
    intgr(f(i));
}
```

In the code above, `intgr` is a function that integrates (using the rectangle method) a given function `f` depending on a sample time `i*h` (`h` being the sampling step). Integration is carried out up to some threshold defined by the interval `[INF,SUP]`. Suppose `f(i)=cos(2`π`ih)`. This can be the result of the sampling at times ih of an external physical environment agreeing with the following ordinary differential equation (ODE) – for instance coming from the modelling of some electronic oscillator circuit with negligible impedance, or after the transient period:

$$\frac{d^2u}{dt^2} + 4\pi^2 u = 0$$

With initial conditions u(0)=1 and $\frac{du}{dt}(0) = 0$. Then for h any power of 2, it is easy to see that:

I. `intgrx` is finite on any trace of execution (and for every `N`); indeed, it is of the order of $\frac{1}{2\pi}\sin(2\pi x)$ when `h` is small enough, as seen in the figure below, for $h = \frac{1}{8}$:



II. the imprecision error due to the floating-point truncation error, in the round to nearest mode is also bounded.

For point I, this is only true if the input signal is sampled at a sufficient rate. For instance, if `h=1`, the result would diverge to infinity, but any sampling with $h \leq \frac{1}{2}$ would do (this is related to Shannon's theorem).

In order to illustrate point II, for $h = \frac{1}{8}$, the sampling sequence is of period 8:

$$1,\ \frac{\sqrt{2}}{2},\ 0,\ -\frac{\sqrt{2}}{2},\ -1,\ -\frac{\sqrt{2}}{2},\ 0,\ \frac{\sqrt{2}}{2};$$

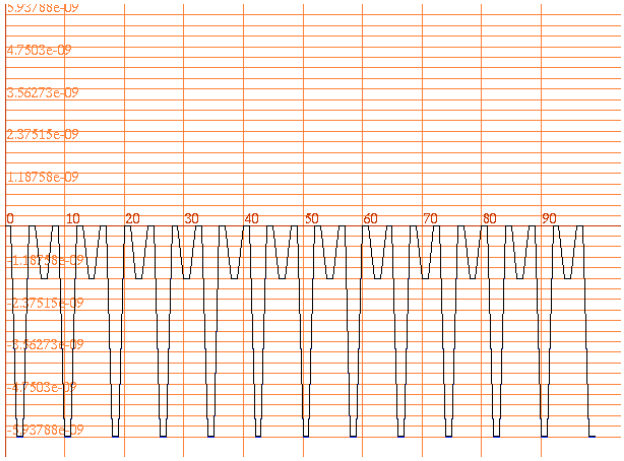
the imprecision error coming only from the rounding of $\frac{\sqrt{2}}{2}$ (the corresponding floating-point number is slightly less than the real number, by about 1.21e-8). The rounding error for $1+\frac{\sqrt{2}}{2}$ is of about –5.938e-9. Also, it is a property of the IEEE 754 standard [16] that, in particular, for $\frac{1}{2} \leq x \leq 1$, (1+x)-x=1 in the rounding to the nearest mode. Hence:

$$\left(1+\frac{\sqrt{2}}{2}\right)-\frac{\sqrt{2}}{2}=1$$

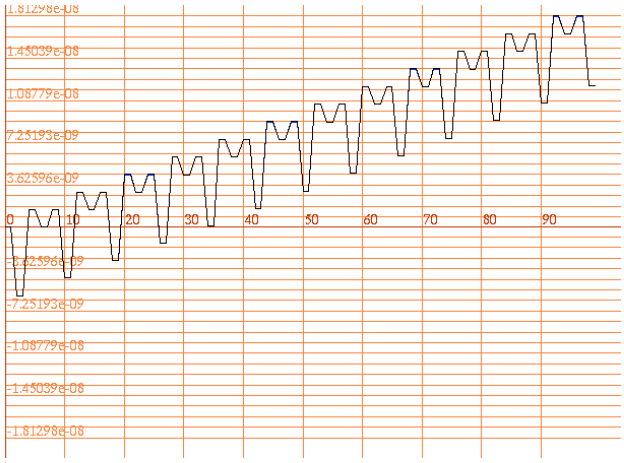
This means that the imprecision error on `intgrx` is the following sequence of period 8:

0, -5.938e-9; -5.938e-9, 0, 0, -1.51e-9, -1.51e-9, 0,

which is bounded indeed, as can be seen in the figure below:



Of course any unfortunate error in the sampling time (or more general irregular sampling times, such as the ones considered in works in applied mathematics, such as [19]), or due to the sensor, might entail an important drift between the computed integral and the real integral. For instance, suppose that at every period (of 8 samples), sample number 4 (which should be taken at time (8k+3)`h` is in fact taken at time (8k+3)h+ε where ε is of the order of 1.71e-8. For this sampling time t, `f(t)` is equal to the floating-point number, closest to the real number $-\frac{\sqrt{2}}{2}$, hence the imprecision error is zero for the computation of `f` at this precise time. But this produces a positive drift between the real number and the floating-point computation of the integral, which will take, after a finite number of iterations, the real number to the `SUP` threshold whereas the floating-point number will be periodic always less than 0.214 approximatively. See the figure below to see the slow drift of the imprecision error:



The fact is that any simple static analyser, based on standard intervals for instance, will be able to prove I, but II will be in general quite hard. Fluctuat is able to find that if the sampling is exact for a given `h`, and for the precise signal we have been looking at (with

an extremely precise estimate of the solution of the ODE, using for instance the algorithm [13]) then "unfolding" the loop $\frac{1}{h}$ times, periodically, will allow for giving very accurate bounds on the imprecision error. For general input sampling times, or more general types of signals, it will have no other solution than bound the imprecision error by the whole interval size `SUP-INF`. It will be right (not that pessimistic since this might really happen) but one might hope for better results, since in general an integrator is just a small part, buried deep in a control command code. This calls for the following:

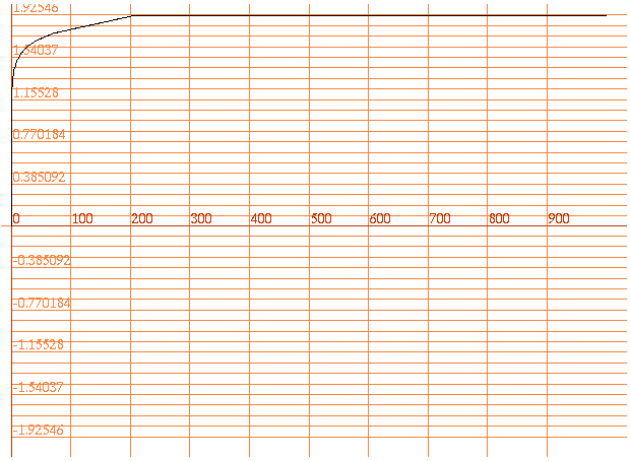
- a careful study of the imprecision error in the sampling of some classical (but general) classes of signals, or of solutions to some classical ODEs
- probabilistic estimates of the imprecision error in more general cases, in order to be less pessimistic, and give an indication of the "average case", since the worst case might be very often erroneous.

For instance, for our last statement: if we model the perturbation of the sampling time as independent gaussian random variables with zero average and small variance, this will result in a zero average and small variance difference between the real integral of the perturbed discrete signal and the real integral of the periodic discrete signal. We conjecture that under some mild assumptions, this will also be the case between the real integral and the floating-point integral of the perturbed discrete signal.

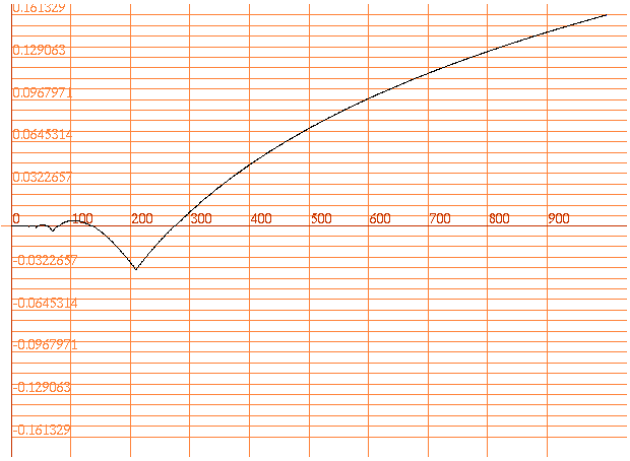
As for the former point, one can readily see that there are functions that always behave badly with respect to the estimate of their integrals. For instance, if `f` is a decreasing function converging towards zero, but whose integral (from 0 to x, any positive real) is not bounded, like

$$f(x) = \frac{1}{x+1}$$

the drift between the real number and the floating-point number computation of its integral is *always* very important, as we will show shortly. Below are the pictures of the computed integral in the floating-point numbers, and, respectively, of the imprecision error made for 10000000 iterations:



Then for the imprecision errors (the last part of the curve is actually increasing for ever, until the `SUP` threshold):

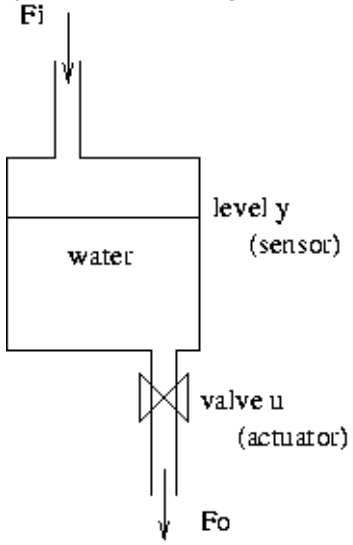


Consider any sampling of such a function `f`, such that for all N, there is always an infinite number of samples after N (plus infinity is an accumulation point of the sequence of samples). Then, because `f` is converging towards 0, and as it is always positive (it is decreasing towards zero), there is an X such that for all x greater than X, `f(x)` is less than half of the smallest positive floating-point number (of a given type). Hence for all subsequent samples, the signal will be rounded to zero, hence the integral as computed by our code, in the floating-point numbers, will converge in a finite time, to a finite number. Notice now that the integral of `f` from X to any further t will diverge to plus infinity when t goes to infinity. For a decreasing and positive function, the discrete integral as given by the algorithm programmed in our code, always gives an upper-approximation (in the *real numbers*), of the real integral, hence will also diverge to plus infinity. This entails that the imprecision error can grow arbitrarily high (up to the thresholds given by the code).

4. Future challenges

This gives but a simplified view of the problems one might encounter when trying to automatically verify typical PID control systems [12], very much used in industrial codes. More generally, in most control systems, feedback loops are created, linking the output of the code to the input, at some further step, in a more or less direct way. For instance, the output of a control system goes through actuators, that interact with the physical environment, and in turn, will modify the values that the sensors will input to the control system, later on. We have already seen in the last section that the verification of such systems can only be precise enough if the "semantics" of the physical environment is modelled accurately, for instance using models involving some ODE or PDE. But this involved only the "separate" resolution of ODEs before trying to solve the (discrete) abstract semantical equations given by the code under analysis. For programs which interact also through actuators on the environment (that is, *every* real life control system), this is not enough: we need to be able to solve the discrete and continuous equations jointly. This poses new challenges to the field of automatic validation of systems, integrated in their environment, since methods for solving these discrete and continuous systems are in general fairly different! It is already interesting to note that some control theorists have also begun to make the way in the reverse direction, integrating software in their models of the physical world [9], [10]. It is probable that the two communities will meet on a joint solution.

Take the following example**.** Water is poured into a tank of water at a given rate Fi, and a controller can act on the output flow Fo, possibly faster than the output flow (as shown in the figure below):



Hence the level u of the tank is governed by the ODE:

$$\frac{du}{dt} = F_i - F_o$$

The controller will try to adjust the outgoing flow through an action on the valve, so that it can make the level reach and stabilize at the objective level `yc`. It will take as input, at periodic time ticks, the current level of the water (not knowing the input flow `Fi` of course) so that to determine whether it has to open more or close more the valve. Typical controllers (see [12]) take as new value of the flow `Fo` at the time tick i, controlled by the value u_i, a coefficient `K` multiplied by e_i, the difference between the current level `y` and the objective level `yc`**.** These very simple controllers are called proportional controllers (P controllers). They can be controlled also with an extra term, using a correction term based on the time derivative of e_i (to have a PD controller). Finally, one can also add up a factor of the integral over time of e_i (to have a general PID controller). We refer the reader to [12] for instance, for more about the respective interests of these different controllers. Note that PID controllers are heavily used in industry in the large, this is not a purely academic example.

An implementation of a PID controller can be easily encoded as follows, using the sum of e_i over time to compute the integral (see our integrator of Section 3) and $e_i - e_{i-1}$ as a simple discretization of the time derivative of e_i. Our PID controller is implemented in the `main` function below; it implements the following iterative scheme:

$$u_i = K\left(e_i + \frac{T}{\tau_I}\sum_{j=0}^{i} e_j + \frac{\tau_D}{T}\left(e_i - e_{i-1}\right)\right)$$

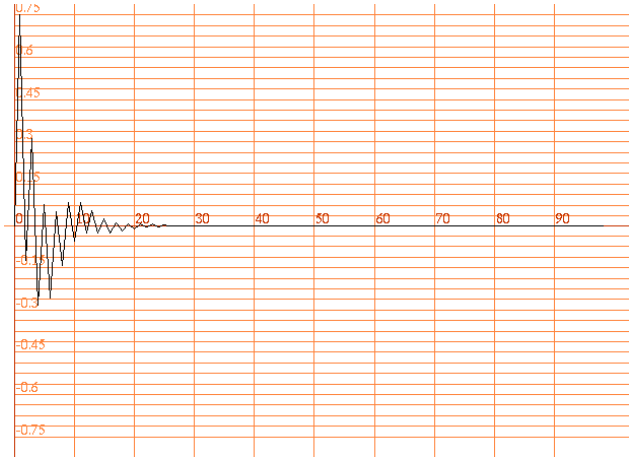
The different variables involved in the code below are:

- yn is the current value (at sample time `n`) of the level of the tank,
- `taui` is the integration time of the PID controller,
- `taud` is the derivation time of the PID controller,
- `K` is the gain of the controller,
- `yc` is the level value that the controller should converge to**,**
- `ui` (as computed by the `main` function), is the value of the flow of water that the controller imposes on the valve, at time `i`.

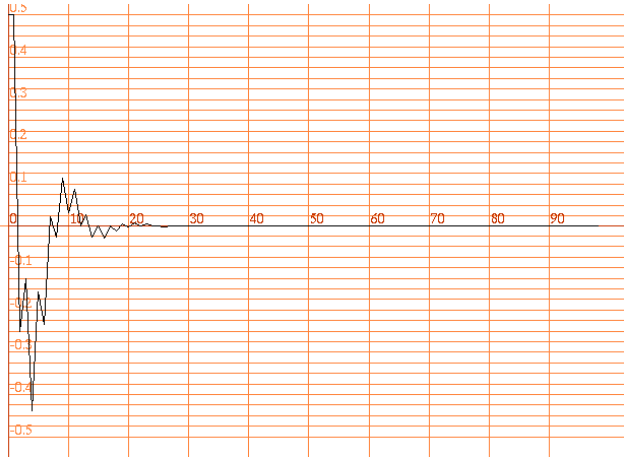
```
typedef double NUM;
static NUM yn = 0;
NUM ui = 0;

NUM y(int i) {
  yn += ui;
```

```
  return yn;
}

void main() {
  NUM yi, yc;
  NUM K;
  NUM T;
  NUM taui;
  NUM taud;
  NUM ei, sumej, epi;
  int i;
  T = 1;
  taui = 1;
  taud = 1;
  K = .5;
  yc = .5;
  yi = y(0);
  epi = yc-yi;
  sumej = epi;
  for (i=1;i<100;i++) {
    yi = y(i);
    ei = yc-yi;
    sumej = sumej+ei;
    ui = K*(ei+sumej*T/taui+taud/T*(ei-
epi));
    epi = ei;
  }
}
```

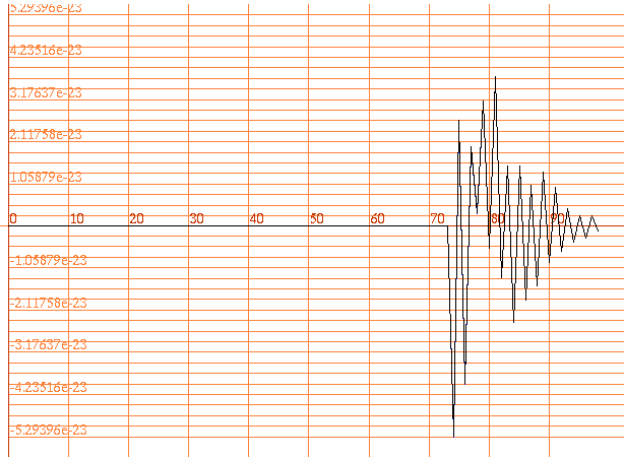
This example is complex for a static analyser, as already was the integrator in Section 3, which it contains. So we only study here the numerical behaviour for one execution only, using Fluctuat and the same semantics as explained in Section 1 (this is a symbolic execution in the corresponding abstract domain). The graph below shows the value of the command `ui` over time, as automatically given by Fluctuat :



while the figure below pictures the evolution over time of the measure `ei` of the difference between the current level of water and the objective level:



Notice that numerical experiments show that this scheme is well behaved in general, as shown in our particular run of the algorithm, in the figure below (imprecision error of the difference between the level `yi` objective `yc` over time):



Notice that the errors at the end are going to zero, and are transient, due to the very low value of `ei=yi-yc` at these iterations (around the ulp(1) from the 70th iteration on). The first iterates are actually computed exactly (the floating-point number is equal to the real number) because the first few iterates are of the form 2^{-n} (the first ones are 0.5, 0.5, -0.25, -0.125 etc.) and thus the integral, derivative and gain computation (since the gain is also of this form: 0.5) have a small finite bit expansion in the floating-point number format. It is only after about 70 iterations, when close enough to the ulp, that some bits are lost in the rounding to nearest mode, but this still converges towards 0.

Unfortunately, this is but an approximation of the system we should prove correct (in particular, as for the imprecision errors in the control mechanism). In the real system, function `y` is a sampling at some more or less precise times of the current level of the

tank, modelled (in the real numbers) as the solution of the differential equation above. We can make two remarks here:

- Fo is equal to the current flow command (given by `ui`) at the last clock tick – which is only a floating-point number approximating the value of the real command that should have taken place. Also the actuator (the valve) may not be very precise, and a new error is created there. This affects at the next sampling time the value of `y`. Because of the feedback loop, the imprecision error of the computation of `ui` will affect later inputs, possibly drifting dramatically (there is an integrator, as in last section, in the code!). But now, the characterization of the input signals, even probabilistically, is very hard to make, since the input depends on the computation.
- Here, the ODE describing the external physical world is very simple, it is easy to simulate its solution in the same formalism as for the controller code (this is what we did here to analyse the code through our current version of Fluctuat). This is unfortunately not the case in general, and one has to design mixed ODE-discrete systems solvers, in order to be able to answer the verification problem (see for instance [13] or [11]).

Notice that once again, it is the integrator part of the code which creates the main problem for Fluctuat: if we just use a P or PD controller, then we can prove automatically that there is no big imprecision error, in the particular case when the outside physical world is simulated by a C function as the one we had for the PID controller.

5. Conclusion

We have seen that Fluctuat is able to assert the precision of many floating-point computations in industrial, representative, control systems. Our current and future work is driven by two related objectives: to improve the precision of the analyses and to deal with larger classes of problems. These directions often are convergent since, for example, a more accurate domain enables the analyser to treat new classes of algorithms while improving the precision of the results in general. Obviously, as for other static analysers, a trade-off between precision and performances has to be carefully chosen.

A main difficulty comes from the fact that programs contain information on the evolution of floating-point values but not on the error terms. For example, the threshold performed by the integrator of Section 3 does not allow one to limit the error on the accumulator. So, in order to enable Fluctuat to infer such properties, we aim at designing new domains. For example, we are currently introducing linear correlations between variables, to limit the usual drawbacks of interval arithmetic (wrapping effect) on the error terms [20]. We also plan to compute information on the derivatives of the errors with respect to the floating-point values, by (safe) automatic differentiation, to correlate the floating-point values to the error terms.

New classes of problems have already been mentioned in this article. The analysis of hybrid systems, described in Section 4, is quite a challenging direction. Besides the definition of the analysis itself, it requires the design of safe numerical algorithms to find sure solutions to equation systems (typically ODE or PDE). Basically, for the safety and precision of the analysis, these algorithms have to output fine over-approximations and under-approximations of the real solution. For instance, in the case of the integrator of Section 3, a current implementation of the abstract domain of O. Bouissou [13] can already prove the boundedness of the variable `intgrx` for the cosine function, assuming the sampling is done at exact periodic times.

Next, our interest for introducing random variables to model some error terms was illustrated in Section 3 to cope with problems where, in the worst case, some error terms may indeed grow infinitely, for example if round-off errors do not cancel each other, even if this scenario is very unrealistic. In these cases, we plan to use new analyses based on probabilistic abstract domains [17]. Another mid-term objective is to analyse mathematically more difficult problems. More precisely, we aim at designing new analysis frameworks to cope with numerical intensive codes, used for simulations (by opposition to control systems). However, the properties ensuring the stability of this kind of algorithms usually are much more complicated than in the case of control systems. Special purpose domains, possibly specific to some classes of numerical methods will probably be needed for these applications.

Another important research direction concerns the validation of the conformance of an implementation with respect to a model and this problem can be considered at different levels of abstraction. In practice, control systems usually are designed using high-level, often block diagram-based, languages like SCADE of SIMULINK and, then, are translated into C or assembler (by hand or automatically). If C code is used, it ultimately has to be compiled into assembler.

In any case, one has to assert that an implementation conforms to a model (C w.r.t. SCADE or SIMULINK, assembler w.r.t. C, etc.). These translations are error prone for numerical precision since floating-point computations are very sensitive, for example, to the evaluation order of the operations and to the different numbers of bits available in registers and memory locations. Hence, implementation details may significantly change the quality of a code, even if it mathematically computes what the specification requires in the reals. This makes us investigate different topics: first, the analysis of control systems described by block diagrams is a necessary intermediate objective. Second, we plan to use techniques to compare invariants at different levels of abstractions. This approach was successfully applied to run-time errors [18] but numerical precision introduces new problems, since we cannot expect to obtain exactly the same error terms in the model and the implementation. Finally, we investigate ways to make our analysers (C and assembler) collaborate, at least to analyse C codes with inlined assembler routines and, at longer term, to help solving the conformance problem. For example, under some assumptions, any arithmetic C expression could be translated in assembler before applying a mixed analysis.

Finally, we aim at defining more robust safety properties concerning the numerical precision of programs. Current criteria consist of proving that no error term overpasses a given limit, expressed as an absolute or relative quantity. But such criteria remain weak: for example an error of one percent may be, most of the time negligible, while being critical in some situations. In the context of hybrid systems, we plan to define more robust criteria based on the actions performed by a program on the environment. For example, an alarm has to be activated in the same cases, independently of which arithmetic is used, and independently of the precision of the computations.

Last but not least, in the search for realistic modelling and verification of *systems* involving software, some particular events have to be taken into account: for instance, faults of sensors, or actuators [15], since most control systems are inherently designed to be robust to some forms of accidents, and a complete proof should integrate the proof of the relevant software (and hardware) mechanisms. We advocate that other interactions with software than the ones with the physical world should be considered, namely the interaction with hardware (through OS, or simpler apparatus, like simple drivers). An ultimate goal, would be to include in the proof of a control system the human factor (which gives some inputs to the software in particular, for instance a pilot of a plane, or a driver of a car), but this has unfortunately far less clear scientific grounds, although models could and should include some basic factors like minimum response time of a human being, the possibility of an illegal command issued by the human being (such as turn off the key while driving), which can be formalized.

6. Acknowledgement

The authors acknowledge the contribution of their colleagues to this work, and in particular A. Chapoutot (see [14]) and O. Bouissou.

7. References

[1] Static Analysis-Based Validation of Floating-Point Computations, S. Putot, E. Goubault and M. Martel, follow-up of the seminary on Numerical Software with Result Verification, at Dagstuhl, Germany, LNCS 2991, 2004.

[2] Asserting the Precision of Floating-Point Computations: a Simple Abstract Interpreter, E. Goubault, M. Martel and S. Putot, European Symposium on Programming, LNCS 2305, 2002.

[3] Validation of Assembler Programs for DSPs: A Static Analyzer, M. Martel, Program Analysis for Software Tools and Engineering, ACM Press, 2004.

[4] Propagation of Roundoff Errors in Finite Precision Computations: a Semantics Approach , M. Martel, European Symposium On Programming, LNCS 2305, 2002.

[5] Multiprecision library, http://www.mpfr org

[6] Design and implementation of a special-purpose static program analyser for safety critical real-time embedded software, B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, X. Rival, LNCS 2566, 2003.

[7] Abstract Interpretation: A unified lattice model for static analysis of programs by construction of approximations of fixed points, P. Cousot and R. Cousot, POPL'77.

[8], The algorithmic analysis of hybrid systems, R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P.-H. Ho, X. Nicollin, A. Olivero, J. Sifakis and S. Yovine, TCS, Vol. 138, P. 3-34, 1995.

[9] Safety verification of hybrid systems using barrier certificates, SIAM journal on Systems and Control Theory, S. Prajna and A. Jadbabaie, 1994.

[10] A framework for worst-case and stochastic safety verification using barrier certificates, S. Prajna, A. Jadbabaie, G. Pappas, IEEE transactions on automatic control, 2005.

[11] Guaranteed error bounds for ordinary differential equation, G. F. Corliss, Lectures Notes SERC, 1994.

[12] Elements d'Automatique, M. Depeyrot and P. Faurre, Dunod, 1974.

[13] Analyse statique par interpretation abstraite de systèmes hybrids discrets-continus, O. Bouissou, Technical Report CEA-LIST-DTSI05-301, 2005.

[14] Analyse statique pour la precision numérique, A. Chapoutot,, Master's Thesis, Université Paris 6, 2005.

[15] Distributed Algorithms, N. Lynch, Morgan-Kaufmann Editors, 1996.

[16] What every computer scientist should know about floating-point arithmetic, D. Goldberg, ACM Computing Surveys, Vol. 23, No. 1, ACM, 1991.

[17] An abstract Monte-Carlo Method for the Analysis of Probabilistic Programs, D. Monniaux, Principles of Programming Languages, ACM, 2001.

[18] Invariant Translation-Based Certification of Assembly Codes, X. Rival, International Journal on Software and Tools for Technology Transfer, 2004.

[19] Recursive Non Parametric Spectral Estimation From Irregularly Sampled Observations, A. Rivoira, G. Fleury, IEEE DSP, 2002.

[20] Weakly relational domains for floating-point computation analysis, E. Goubault, S. Putot, NSAD'2005.