Preservation of Lyapunov-Theoretic Proofs: From Real to Floating-Point Arithmetic

Vivien Maisonneuve, Olivier Hermant, François Irigoin

Preservation of Lyapunov-Theoretic Proofs: From Real to Floating-Point Arithmetic

Vivien Maisonneuve, Olivier Hermant and François Irigoin
MINES ParisTech, France
Email: {vivien.maisonneuve, olivier.hermant, francois.irigoin}@cri.mines-paristech.fr

***Abstract*—In a paper, Feron presents how Lyapunov-theoretic proofs of stability can be migrated toward computer-readable and verifiable certificates of control software behavior by relying of Floyd's and Hoare's proof system.**

However, Lyapunov-theoretic proofs are addressed towards exact, real arithmetic and do not accurately represent the behavior of realistic programs run with machine arithmetic. We address the issue of preserving those proofs in presence of rounding errors resulting from the use of floating-point arithmetic: we present an automatic tool, based on a theoretical framework the soundness of which is proved in Coq, that translates Feron's proof invariants on real arithmetic to similar invariants on floating-point numbers, and preserves the proof structure. We show how our methodology allows to verify whether stability invariants still hold for the concrete implementation of the controller.

We study in details the application of our tool to the open-loop system of Feron's paper and show that stability is preserved out of the box. We also translate Feron's proof for the closed-loop system, and discuss the conditions under which the system remains stable.

I. Introduction

Stability constitutes an essential attribute of control systems, especially when human safety is involved, as in medical or aeronautical domains. Modern system developments, such as adaptive control technologies, rely on robust stability and performance criteria as the primary justification for their relevance to safety-critical control applications. Motivated by such applications, there exist many theorems that ensure system stability and performance under various assumptions and in various settings. Lyapunov's stability theory plays a critical role in that regard.

The low-level software implementation of a control law can be inspected by analysis tools available to support the development of safety-critical computer programs. The simplest program analysis technique consists of performing several simulations, sometimes including a software or hardware representation of the controlled system in the loop. However, simulations provide information about only a finite number of system behaviors. More advanced methods include model checking and abstract interpretation, e.g. using Astrée [1]. In these methods, inputs are the very computer programs and outputs are certificates of proper program behavior along the chosen criterion. Another possibility is to use theorem-proving techniques, supported by tools such as Coq, Isabelle or PVS [2], [3], [4]. These proof assistants can be used to establish properties of programs and more general mathematical constructs. Model checking, abstract interpretation, and theorem-proving tools are all used to verify safety-critical applications.

In [5], Feron investigates how control-system domain knowledge and, in particular, Lyapunov-theoretic proofs of stability for the high-level theoretical modelization, can be migrated towards computer-readable and verifiable certificates of control software behavior by relying on Floyd's and Hoare's proof system [6], applied to MATLAB pseudo-code (see Sections II and III). Feron presents both open-loop and closed-loop implementations of his controller, i.e. depending whether the system action on the environment and its feedback are modelized or not. But errors resulting from the use of floating-point arithmetic are not considered.

In this paper, we address this issue by presenting an automatic approach to translate Lyapunov-theoretic stability proof invariants on pseudo-code with real arithmetic, as provided in Feron's article, to similar invariants on machine code that take into account rounding errors introduced by floating-point arithmetic. We use them to verify whether the stability conditions still hold, in which case system stability with floating-point numbers is established. This approach is based on a generic theoretical framework which soundness is proved in Coq, described further in this document, and is implemented as an automatic tool.

This document is organized as follows. First, we describe the second-order dynamical system example used by Feron in [5], with the corresponding controller. Next, Feron's analysis of the open-loop controller with real numbers is presented. In the next sections, we introduce our generic translation scheme and its implementation as a Python library, and use it thereafter to rewrite Feron's analysis with floating-point arithmetic. Finally, the case of the closed-loop system is handled. The document concludes with a discussion on the generality of this approach.

II. Motivating Example by Feron

We consider the first system described in the article of Feron [5]. It is a dynamical system composed of a single mass and a single spring shown in Figure 1.

The position input y of the mass is available for feedback control. The signal y_d is the reference signal, that is, the desired position to be followed by the mass.

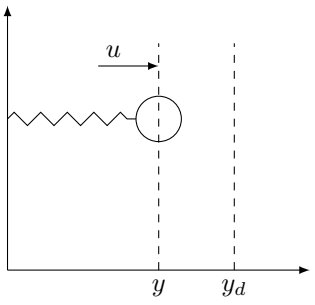

Figure 1. Mass-spring system

A discrete-time MATLAB implementation of the controller, using real numbers, is provided in [5]. The source code is shown below. The process to produce this code from the system modeling is covered in detail in Feron's paper, but in not necessary to understand this document.

```
1  Ac = [0.4990, -0.0500; 0.0100, 1.0000];
2  Bc = [1; 0];
3  Cc = [564.48, 0];
4  Dc = -1280;
5  xc = zeros(2, 1);
6  receive(y, 2); receive(yd, 3);
7  while (1)
8      yc = max(min(y - yd, 1), -1);
9      skip;
10     u = Cc*xc + Dc*yc;
11     xc = Ac*xc + Bc*yc;
12     send(u, 1);
13     receive(y, 2);
14     receive(yd, 3);
15     skip;
16 end
```

In this code, the `skip` statement is a null operation: when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically to be surrounded by invariants, but no code needs to be executed.

Apart from the mechanical system state observation y and the desired system output y_d, variables in this code are:

- $x_c = \binom{x_{c1}}{x_{c2}} \in \mathbb{R}^2$ is the discrete-time controller state;
- $y_c \in [-1, 1]$ is the bounded output tracking error, i.e. the input $(y - y_d)$ is passed through a saturation function to avoid variable overflow in the controller;
- $u \in \mathbb{R}$ is the mechanical system input, i.e. the action to be performed according to the controller.

Constants A_c, B_c, C_c and D_c are the discrete-time controller state, input, output and feedthrough matrices. The commands `send` and `receive` are basically I/O: they respectively send and receive data given in the commands first argument through a specific channel given by the commands second argument.

III. Open-Loop Stability Proof with Reals

The stability proof of this system relies on Lyapunov theory. In simple terms, a system is *Lyapunov stable* if all states x_c reachable from an initial starting state belonging to a bounded neighborhood V of an equilibrium point x_e remain in V.

Lyapunov theory provides constraints that must be satisfied by such a V. On linear systems, they are equations that can be solved using linear matrix inequalities [7]. Commonly, V is an ellipsoid.

In this case, to prove Lyapunov stability, we need to show that at any time, x_c belongs to the set $\mathcal{E}_P$ chosen by Feron according to Lyapunov's theory:

$$\mathcal{E}_P = \{x \in \mathbb{R}^2 \mid x^{\mathrm{T}} \cdot P \cdot x \leq 1\}, P = 10^{-3} \begin{pmatrix} 0.6742 & 0.0428 \\ 0.0428 & 2.4651 \end{pmatrix}$$

using $\mathcal{E}_P$ as the stability neighborhood V.

This set is the full ellipse shown in Figure 2, centered around 0 and slightly slanted:

$$x_c \in \mathcal{E}_P \iff 0.6742x_{c1}^2 + 0.0856x_{c1}x_{c2} + 2.4651x_{c2}^2 \leq 1000.$$

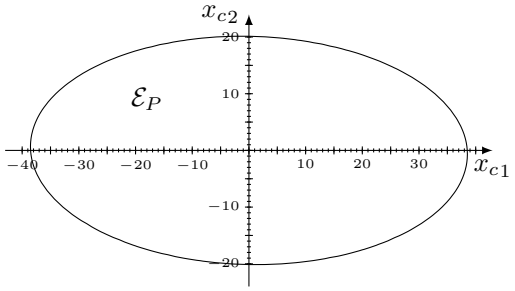

Figure 2. The stability domain $\mathcal{E}_P$

A stability proof of the controller is provided in [5], using Floyd-Hoare program annotation technique [6]: each program instruction comes with an invariant. The program annotated by Feron is reproduced below.

```
5  xc = zeros(2,1);
   % x_c ∈ E_P
6  receive(y, 2); receive(yd, 3);
   % x_c ∈ E_P
7  while (1)
       % x_c ∈ E_P
8      yc = max(min(y - yd, 1), -1);
       % x_c ∈ E_P, y_c^2 ≤ 1
9      skip;
       % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10     u = Cc*xc + Dc*yc;
       % (x_c; y_c) ∈ E_{Q_μ}, u^2 ≤ (C_c D_c)·Q_μ^{-1}·(C_c D_c)^{-1}
11     xc = Ac*xc + Bc*yc;
       % x_c ∈ E_R, R = [(A_c B_c)·Q_μ^{-1}·(A_c B_c)^T]^{-1},
       % u^2 ≤ (C_c D_c)·Q_μ^{-1}·(C_c D_c)^{-1}
12     send(u, 1);
       % x_c ∈ E_R
13     receive(y, 2);
       % x_c ∈ E_R
```

```
14    receive(yd, 3);
      % x_c ∈ E_R
15    skip;
      % x_c ∈ E_P
16  end
```

Most of the proof relies on algebraic arguments. For example, the invariant loosening on Line 9:

```
      % x_c ∈ E_P,   y_c^2 ≤ 1
9     skip;
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
```

means

$$x_c \in \mathcal{E}_P \wedge y_c^2 \leq 1 \Longrightarrow \left(\begin{smallmatrix} x_c \\ y_c \end{smallmatrix}\right) \in \mathcal{E}_{Q_\mu},$$

with $Q_\mu = \left(\begin{smallmatrix} \mu P & 0_{2\times 1} \\ 0_{1\times 2} & 1-\mu \end{smallmatrix}\right)$ and $\mu = 0.9991$.

The correctness of this assertion stems from the fact that, given any value of $\mu \in [0, 1]$, the domain $\mathcal{E}_{Q_\mu}$ is a solid ellipsoid, centered around 0, and whose intersection with the planes $y_c = 1$ and $y_c = -1$ is equal to $\mathcal{E}_P$. Consequently, the solid bounded cylinder $\mathcal{C} = \{\left(\begin{smallmatrix} x_c \\ y_c \end{smallmatrix}\right) \mid x_c \in \mathcal{E}_P \wedge y_c^2 \leq 1\}$ is included within $\mathcal{E}_{Q_\mu}$ (Figure 3).

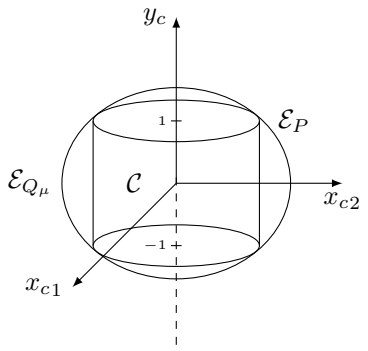

Figure 3. Inclusion of $\mathcal{E}_P$ within $\mathcal{E}_{Q_\mu}$

The next two invariants

```
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10    u = Cc*xc + Dc*yc;
      % (x_c; y_c) ∈ E_{Q_μ}, u^2 ≤ (C_c  D_c) · Q_μ^{-1} · (C_c  D_c)^{-1}
11    xc = Ac*xc + Bc*yc;
      % x_c ∈ E_R, R = [(A_c  B_c) · Q_μ^{-1} · (A_c  B_c)^T]^{-1},
      % u^2 ≤ (C_c  D_c) · Q_μ^{-1} · (C_c  D_c)^{-1}
```

also rely on similar algebraic arguments and theorems. Other invariants are trivial. Finally, only the very last loosening

```
      % x_c ∈ E_R
13    skip;
      % x_c ∈ E_P
```

i.e. $\mathcal{E}_R \subset \mathcal{E}_P$, that "closes" the loop, is not purely algebraic since its validity relies on the numerical parameters A_c, B_c, C_c, D_c. This assertion needs to be checked to ensure the correctness of the proof statements. This can be done either numerically, or algebraically for at most two-dimensional systems like this one.

We have checked this proof using Mathematica [8]. Our proof notebook is available online[1].

IV. Proof Scheme with Limited-Precision Numbers

We would like to check that the stability proof still holds on a realistic controller device, using limited-precision numbers (for instance, but not necessarily, floating-point numbers). When using limited-precision arithmetic, the values of constants are slightly altered and calculations are likely to produce rounding errors.

In absolute terms, it is impossible to switch from real to limited-precision numbers without affecting the behavior of the controller. Thus, the stability proof cannot be preserved in the general case. On the other hand, the program still can be stable if rounding errors are small enough and the final inclusion $\mathcal{E}_R \subset \mathcal{E}_P$ leaves them enough room. We study how proof invariants can be tweaked so that they apply to a limited-precision semantic. Our goal is to derive from the proof scheme for real numbers a proof scheme suited for limited-precision arithmetic, whose correctness, although not guaranteed, can be checked as easily as in the original proof.

A. Formalism

In this section, an abstract formalism is defined to handle a program an its proof, with both real and limited-precision arithmetic.

```
% d
i
% d' = p(d, i)
```

Figure 4. Abstract scheme of invariant propagation on the original program

Let X be the (finite) set of variables in the program.

1) Values: Let $\mathbb{F} \subset \mathbb{R}$ be the targeted set of limited-precision representations of real numbers (e.g., floating-point numbers). We assume that each real number $c \in \mathbb{R}$ has a finite-precision representation $\bar{c} \in \mathbb{F}$ and that the property (1) is satisfied:

$$c \in \mathbb{F} \Longrightarrow \bar{c} = c. \tag{1}$$

Throughout the rest of the document, we will use the uniform notation $\mathbb{K}$ to refer to either the set $\mathbb{R}$ or $\mathbb{F}$.

2) Functions Symbols: We consider a set of function symbols $\mathcal{F}_\mathbb{R}$ on real numbers. Each real function symbol $f \in \mathcal{F}_\mathbb{R}$ is associated to a finite-precision counterpart $\bar{f} \in \mathcal{F}_\mathbb{F}$. Functions symbols in $\mathcal{F}_\mathbb{R}$ and $\mathcal{F}_\mathbb{F}$ will be simply referred as *functions* when there will be no ambiguity.

Each function symbol f in $\mathcal{F}_\mathbb{R}$ or $\mathcal{F}_\mathbb{F}$ has an *arity* $n \in \mathbb{N}$ and can be *evaluated* on its domain:

$$f \in \mathcal{F}_\mathbb{K} \Longrightarrow \mathrm{eval}(f) \in \mathbb{K}^n \to \mathbb{K}. \tag{2}$$

[1] Mathematica source file is available at: http://www.cri.ensmp.fr/people/maisonneuve/lyafloat/resources/lyafloat_stability.nb, and the corresponding PDF file at: http://www.cri.ensmp.fr/people/maisonneuve/lyafloat/resources/lyafloat_stability.pdf.

Functions f and $\bar{f}$ have the same arity:

$$\text{arity } f = \text{arity } \bar{f}.$$

As for now, no other assumption is made as to the behavior of $\bar{f}$ compared to f.

Notice that function symbols are not identified to their evaluations. The reason is that functions with similar evaluations on real arithmetic may behave differently on a limited-precision paradigm, e.g. function $x \mapsto 2^{100} + x - 2^{100}$ compared to the identity function: $x \mapsto x$.

3) Valuations: A *valuation* v on $\mathbb{K}$ is a function that maps variables in X to values in $\mathbb{K}$. The set of valuations on $\mathbb{K}$ is noted $\mathcal{V}_{\mathbb{K}}$.

$$\mathcal{V}_{\mathbb{K}} \ni v : X \to \mathbb{K}.$$

Notice that $\mathcal{V}_{\mathbb{F}}$ is a subset of $\mathcal{V}_{\mathbb{R}}$.

Each valuation v in $\mathcal{V}_{\mathbb{R}}$ is associated to a valuation $\bar{v}$ in $\mathcal{V}_{\mathbb{F}}$, defined as follows:

$$\bar{v} : x \in X \longmapsto \overline{v(x)}. \tag{3}$$

Using this definition, it can easily be shown using (1) that for any valuation $v \in \mathcal{V}_{\mathbb{R}}$,

$$v \in \mathcal{V}_{\mathbb{F}} \Longrightarrow \bar{v} = v. \tag{4}$$

Let v_1 and v_2 be two valuations in $\mathcal{V}_{\mathbb{R}}$. The *distance* dist between v_1 and v_2 is the valuation that maps a variable in X to the distance in absolute value between its associated values in v_1 and v_2:

$$\mathcal{V}_{\mathbb{R}} \ni \text{dist}(v_1, v_2) = x \in X \longmapsto |v_1(x) - v_2(x)|. \tag{5}$$

We say that v_1 is *lower* than v_2, and note $v_1 \leq v_2$, if

$$\forall x \in X, v_1(x) \leq v_2(x). \tag{6}$$

Finally, the *sum* of the two valuations, noted $v_1 + v_2$, is the valuation defined as follows:

$$v_1 + v_2 : x \in X \longmapsto v_1(x) + v_2(x). \tag{7}$$

4) Domains: A *domain* d on $\mathbb{K}$ is a set of valuations on $\mathbb{K}$, i.e. $d \subset \mathcal{V}_{\mathbb{K}}$. Domains are used to represent Floyd's and Hoare's invariants seen in previous sections. The set of domains on $\mathbb{K}$ is noted $\mathcal{D}_{\mathbb{K}}$, with $\mathcal{D}_{\mathbb{F}} \subset \mathcal{D}_{\mathbb{R}}$.

A real domain $d \in \mathcal{D}_{\mathbb{R}}$ can be associated to a limited-precision domain $\bar{d} \in \mathcal{D}_{\mathbb{F}}$ defined by:

$$\bar{d} = \{\bar{v} \,|\, v \in d\}. \tag{8}$$

As for valuations in $\mathbb{F}$, it can be shown that

$$d \in \mathcal{D}_{\mathbb{F}} \Longrightarrow d = \bar{d}. \tag{9}$$

We also define an operation to overapproximate a domain with respect to a valuation. The *extension* of domain $d \in \mathcal{D}_{\mathbb{R}}$ using a positive valuation $v \in \mathcal{V}_{\mathbb{R}}$, noted $d \oplus v$, is defined as follows:

$$d \oplus v = \{v_2 \in \mathcal{V}_{\mathbb{R}} \,|\, v_1 \in d \wedge \text{dist}(v_1, v_2) \leq v\}. \tag{10}$$

5) Expressions: We consider a very simple language with variables, constants and function calls. The *expressions* $\mathcal{E}_{\mathbb{K}}$ of this language are constructed as follows:

$$\begin{aligned} \mathcal{E}_{\mathbb{K}} \ni e ::= {} & x \in X && \text{variable} \\ | \; & c \in \mathbb{K} && \text{constant} \\ | \; & f(e_1, \ldots, e_n) && \text{function call} \\ & \text{with } f \in \mathcal{F}_{\mathbb{K}}, \text{arity } f = n \\ & \text{and } \forall i \in [1, n], e_i \in \mathcal{E}_{\mathbb{K}} \end{aligned}$$

An expression $e \in \mathbb{K}$ can be evaluated in the environment described by the valuation $v \in \mathcal{V}_{\mathbb{K}}$. The result is a value in $\mathbb{K}$.

$$\text{eval}(e, v) = \begin{cases} v(x) & \text{if } e = x \in X \\ c & \text{if } e = c \in \mathbb{K} \\ \text{eval}(f)(c_1, \ldots, c_n) & \text{if } e = f(e_1, \ldots, e_n) \\ \quad \text{with } \forall i \in [1, n], c_i = \text{eval}(e_i, v) & \end{cases}$$

where the notation eval is overloaded to handle expressions.

6) Instructions: An *instruction* is the affectation of an expression value to a variable. The set of instructions is noted $\mathcal{I}_{\mathbb{K}}$:

$$\mathcal{I}_{\mathbb{K}} = X \times \mathcal{E}_{\mathbb{K}}$$

We note $x := e$ the instruction $(x, e) \in \mathcal{I}_{\mathbb{K}}$.

As for expressions, an instruction $(x := e) \in \mathcal{I}_{\mathbb{K}}$ can be evaluated in an environment $v \in \mathcal{V}_{\mathbb{K}}$. The result is a valuation in $\mathcal{V}_{\mathbb{K}}$.

$$\text{eval}(x := e, v) = \left(y \in X \longmapsto \begin{cases} \text{eval}(e, v) & \text{if } x = y \\ v(y) & \text{if } x \neq y \end{cases} \right)$$

This definition can be extended to evaluate the image of a domain $d \in \mathcal{D}_{\mathbb{K}}$, the result being also a domain of $\mathcal{D}_{\mathbb{K}}$.

$$\text{eval}(x := e, d) = \{\text{eval}(x := e, v) \,|\, v \in d\}$$

7) Invariant Propagation: An invariant propagator, or simply *propagator*, is a function p that takes as input a domain in $\mathcal{D}_{\mathbb{K}}$, known as *precondition*, an instruction in $\mathcal{I}_{\mathbb{K}}$, and returns a domain in $\mathcal{D}_{\mathbb{K}}$ called *postcondition*.

$$p : \mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}} \longrightarrow \mathcal{D}_{\mathbb{K}}$$

A propagator is expected to return a valid postcondition for a given precondition and instruction. Formally, this can be described by the correctness condition:

$$\forall (d, i) \in \mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}}, \text{eval}(i, d) \subset p(d, i). \tag{11}$$

The set of (correct) propagators is noted $\mathcal{P}_{\mathbb{K}}$.

Propagators are an abstract representation of the arguments used to propagate Floyd's and Hoare's invariant en route to prove a program. Unlike these arguments, propagators have to be defined on the whole domain $\mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}}$, that is, for any precondition and instruction. To make propagators total functions, the universal postcondition can always be returned if the domain is outside the scope of the corresponding argument, for instance when the hypothesis of some Lyapunov theorem is not verified.

B. Translation Scheme

The general translation scheme that is implemented is the following (see Figure 5). Each instruction i in the abstract controller code with real numbers is turned into an corresponding instruction $\bar{\imath}$ on the concrete controller, where real constants and operators are replaced by their limited-precision counterparts. A post-condition $\bar{d}'$ on the limited-precision side is computed using the same proof argument p as in the original code, "enlarged" to include the error term bound v_{err} that may occur in limited-precision arithmetics in $\bar{\imath}$. It is computed on an "intermediate" instruction form $\tilde{\imath}$ which meaning will be explained later. The precondition d might have been previously replaced by $\bar{d}$ as the postcondition of a preceding instruction.

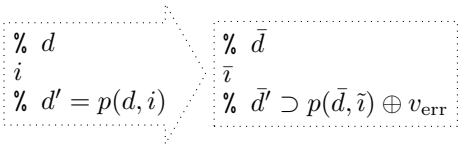

Figure 5. General translation scheme

The error term bound v_{err} depends on the operators of the instruction $\bar{\imath}$ and on the information on program variables given by $\bar{d}$. The limited-precision arithmetic error is not necessarily bounded even when the variables are, e.g. considering division. In this case, we are not able to compute a suitable $\bar{d}'$ and the proof translation attempt fails.

Also, $\bar{d}'$ will be used as a precondition in the next instruction. An effort has to be made so that $\bar{d}'$ has a similar shape to d', in order to fit with the proof argument p' used in this instruction, if possible. Thus, $\bar{d}'$ is an overapproximation of $p(\bar{d}, \bar{\imath}) \oplus v_{\text{err}}$ in the domain of p'. In consequence, the automated analysis falls short if such a $\bar{d}'$ cannot be found.

C. Translation Steps

In this section, we consider an instruction $i = (x := e)$ in $\mathcal{I}_{\mathbb{R}}$, along with a precondition d and a postcondition d' in $\mathcal{D}_{\mathbb{R}}$. d, i and d' are linked through a propagator p in $\mathcal{P}_{\mathbb{R}}$:

$$d' = p(d, i). \tag{12}$$

As outlined in Section IV-B, we study how the instruction i can be translated into an equivalent instruction $\bar{\imath} \in \mathcal{I}_{\mathbb{F}}$ on limited-precision arithmetic, while using information on the precondition d and propagator p of i to compute an interesting postcondition $\bar{d}'$ for $\bar{\imath}$ (given a modification $\bar{d}$ of d). This is a two-step process:

1) First, the program constants $c \in \mathbb{R}$ are converted into their limited-precision representations $\bar{c} \in \mathbb{F}$ while keeping absolute-precision functions, giving an instruction $\tilde{\imath} \in \mathcal{I}_{\mathbb{R}}$;
2) Second, the functions $f \in \mathcal{F}_{\mathbb{R}}$ that appear in the code are changed into their limited-precision counterparts $\bar{f} \in \mathcal{F}_{\mathbb{F}}$.

The resulting code relies on limited-precision arithmetic only, which is what is being sought. The translation scheme is described in Figure 6.

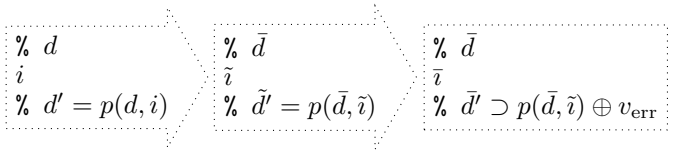

Figure 6. Refined translation scheme

1) Converting Constants: We construct an expression $\tilde{e} \in \mathcal{E}_{\mathbb{R}}$, obtained from e by converting its constants from $\mathbb{R}$ to $\mathbb{F}$. Formally, $\tilde{e}$ is defined recursively as follows:

$$\tilde{e} = \begin{cases} x & \text{if } e = x \in X \\ \bar{c} & \text{if } e = c \in \mathbb{R} \\ f(\tilde{e}_1, \ldots, \tilde{e}_n) & \text{if } e = f(e_1, \ldots, e_n) \end{cases} \tag{13}$$

We let $\tilde{\imath} = (x := \tilde{e})$.

Let $\bar{d} \in \mathcal{D}_{\mathbb{F}}$ be the precondition corresponding to d after converting both constants and functions in the preceding instructions of the program. Let $\tilde{d}' = p(\bar{d}, \tilde{\imath}) \in \mathcal{D}_{\mathbb{R}}$. According to Equation 11, $\tilde{d}'$ is a valid postcondition for precondition $\bar{d}$ and instruction $\tilde{\imath}$.

2) Converting Functions: The next step is to convert real functions f that appear in the expression into their limited-precision counterparts $\bar{f}$, as defined in Section IV-A2. The resulting expression $\bar{e} \in \mathcal{E}_{\mathbb{F}}$ is obtained with:

$$\bar{e} = \begin{cases} x & \text{if } e = x \in X \\ \bar{c} & \text{if } e = c \in \mathbb{R} \\ \bar{f}(\bar{e}_1, \ldots, \bar{e}_n) & \text{if } e = f(e_1, \ldots, e_n) \end{cases} \tag{14}$$

Compared to the definition of $\tilde{e}$ (Equation 13), we just have turned functions symbols f into $\bar{f}$. As previously, we also let $\bar{\imath} = (x := \bar{e})$.

Unlike $\tilde{\imath}$, instruction $\bar{\imath}$ relies on different functions than i (taken from $\mathcal{F}_{\mathbb{F}}$ instead of $\mathcal{F}_{\mathbb{R}}$): the underlying invariant propagation argument in propagator p is very likely not to be applicable to it. In this case, $p(\bar{d}, \bar{\imath})$ would fall back on the universal domain, which of course is valid but prevents us from doing any interesting proof further in the program.

Functional Arithmetic Errors: To circumvent this issue, we propose when it is possible to "enlarge" the invariant $\tilde{d}' = p(\bar{d}, \tilde{\imath})$ found in Section IV-C1 to take into account arithmetic errors introduced when converting functions from $\mathcal{F}_{\mathbb{R}}$ to $\mathcal{F}_{\mathbb{F}}$. Such errors are called functional arithmetic errors, but we will refer to them as *arithmetic errors*, or simply errors, later in this document.

We define the arithmetic error on an expression $\tilde{e} \in \mathcal{E}_{\mathbb{R}}$ and a valuation $v \in \mathcal{V}_{\mathbb{F}}$. The result is a value in $\mathbb{R}$.

$$\text{err}(\tilde{e}, v) = |\,\text{eval}(\bar{e}, v) - \text{eval}(\tilde{e}, v)| \tag{15}$$

This definition is extended to define the arithmetic error of an instruction $\tilde{\imath} = (x := \tilde{e})$ in $\mathcal{I}_{\mathbb{R}}$ on a valuation $v \in \mathcal{V}_{\mathbb{F}}$. The result is a valuation in $\mathcal{V}_{\mathbb{R}}$ defined as follows:

$$\text{err}(x := \tilde{e}, v) = \left(y \in X \longmapsto \begin{cases} \text{err}(\tilde{e}, v) & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases} \right).$$

Finally, a valuation $v_{\text{err}} \in \mathcal{V}_{\mathbb{F}}$ is an *upper bound* of the arithmetic error of instruction $\tilde{\imath}$ on a domain $d \in \mathcal{D}_{\mathbb{F}}$ if:

$$\forall v \in d, \text{err}(\tilde{\imath}, v) \leq v_{\text{err}}.$$

Now, let us consider an instruction $\tilde{\imath}$ in $\mathcal{I}_{\mathbb{R}}$ surrounded by a precondition $\bar{d}$ and a postcondition $\tilde{d'} = p(\bar{d}, \tilde{\imath})$ in $\mathcal{D}_{\mathbb{R}}$:

$$\text{eval}(\tilde{\imath}, \bar{d}) \subset \tilde{d'}$$

Let $v_{\text{err}} \in \mathcal{V}_{\mathbb{F}}$ be an upper bound of the arithmetic error of instruction $\tilde{\imath}$ on domain $\bar{d}$. Then, the implication

$$\forall \bar{d'} \in \mathcal{D}_{\mathbb{F}}, \tilde{d'} \oplus v_{\text{err}} \subset \bar{d'} \Longrightarrow \text{eval}(\bar{\imath}, \bar{d}) \subset \bar{d'} \qquad (16)$$

holds. In other words, if $\tilde{d'}$ is a postcondition for an instruction $\tilde{\imath}$ on real numbers, on a given precondition domain $\bar{d}$, and $\bar{d'}$ is a superset of $\tilde{d'}$ loose enough to support the arithmetic error v_{err}, then $\bar{d'}$ is a valid postcondition for $\bar{\imath}$, the limited-precision counterpart of instruction $\tilde{\imath}$.

Proving this relation raises no particular difficulty, but is a bit too long to fit in this document. A proof in Coq, limited to binary functions (which include all the functions that we need in this paper), was implemented and is available online.

V. Open-Loop Stability Proof Scheme with Floating-Point Numbers

We would like to check whether the stability proof in Feron's program still holds on a controller implemented with floating-point numbers. In this section, we use the IEEE Standard for Floating-Point Arithmetic [9] encoded on 64 bits[2], as most of today's floating-point units do. In this standard, both addition and multiplication are correctly rounded depending on the active rounding mode, which allows to bound the rounding error depending on the values of operands. The case of alternative numeric representations is discussed in Section VII.

A. Automatic Translation

We have developed a program to automatically perform these computations, following the translation scheme framework described in the previous section. It is a module called LyaFloat, written in Python and built upon Python libraries SymPy (to handle symbolic mathematics) and Mpmath (for arbitrary-precision floating-point arithmetic).

Here is the listing of a script that automatically computes the floating-point output invariant ellipsoid $\overline{\mathcal{E}_R}$ (in variable `ERbar`) corresponding to $\mathcal{E}_R$ in the original proof, and tests whether $\overline{\mathcal{E}_R} \subset \mathcal{E}_P$. Two cases are possible:

- either $\overline{\mathcal{E}_R} \subset \mathcal{E}_P$, then the program is Lyapunov-stable on a floating-point architecture;
- or $\overline{\mathcal{E}_R} \not\subset \mathcal{E}_P$: as $\overline{\mathcal{E}_R}$ was obtained through over-approximations, we cannot conclude about the program behavior.

[2]The procedure that follows would be exactly the same with 32-bit floating-point numbers, only with different numerical results.

```
from lyafloat import *
# Parameters
setfloatify(constants=True, operators=True,
       precision=53)

# Definition of E_P
P = Rational("1e-3") * Matrix(rationals(
       ["0.6742 0.0428", "0.0428 2.4651"]))
EP = Ellipsoid(P)

# Definition of E_Q_mu
mu = Rational("0.9991")
Qmu = mu * P
Qmu = Qmu.col_insert(2, zeros(2, 1)).
       row_insert(2, zeros(1, 3))
Qmu[2,2] = 1 - mu
EQmu = Ellipsoid(Qmu)

# Symbols
xc1, xc2, yc = symbols("xc1 xc2 yc")
Xc = Matrix([[xc1], [xc2]])
Yc = Matrix([[yc]])
Zc = Matrix([[xc1], [xc2], [yc]])

# Constant matrices
Ac = Matrix(constants(
       ["0.4990 -0.0500", "0.0100 1.0000"]))
Bc = Matrix(constants(["1", "0"]))
Cc = Matrix(constants(["564.48 0"]))
Dc = Matrix(constants(["-1280"]))

# Definition and verification of E_R
AcBc = Ac.col_insert(Ac.cols, Bc)
R = (AcBc * Qmu.inv() * AcBc.T).inv()
ER = Ellipsoid(R)
print("ER included in EP :", ER <= EP)

# Computation and verification of E_R_bar
i = Instruction({Xc: Ac * Xc + Bc * Yc},
       pre=[Zc in EQmu], post=[Xc in ER])
ERbar = i.post()[Xc]
print("ERbar =", ERbar)
print("ERbar included in EP :", ER <= EP)
```

In this open-loop case, our program LyaFloat is able to check the inclusion. Thus, the stability of the open-loop system with a 64-bit IEEE 754 compliant implementation is formally proven to hold using our proof translation scheme.

Notice that the precision of the floating-point arithmetic can be set in `setfloatify`. This allows us to check that the controller is still stable on a 32-bit only architecture, or to compute that the minimum required precision is 17 bits. This can be useful if, instead of adapting the controller code and/or its proof to the hardware, we have to select a controller device that suits the proof.

In the remainder of this section, we describe the operations performed by our tool to compute $\overline{\mathcal{E}_R}$. It follows the two-step scheme described in Section IV-C. Alternatively to our program, an implementation of this proof using Mathematica is available online: see Footnote 1.

B. Converting Constants

The first step when translating Feron's program is to convert real constants into floating-point numbers. The first lines of code in the program:

```
1 Ac = [0.4990, -0.0500; 0.0100, 1.0000];
2 Bc = [1; 0];
3 Cc = [564.48, 0];
4 Dc = -1280;
```

become, assuming rounding to nearest value:

```
1 Ac = [0.4989999999...6552734375,
        -0.0500000000...2705078125;
         0.0100000000...2880859375,
         1.0000];
2 Bc = [1; 0];
3 Cc = [564.4800000000...5830078125, 0];
4 Dc = -1280
```

Theses matrices $\overline{A_c}$, $\overline{B_c}$, $\overline{C_c}$ and $\overline{D_c}$ will be used instead of the original matrices A_c, B_c, C_c and D_c in the sequel of the proof.

Apart from constants, the proof scheme for the first part of the program is unchanged:

```
5 xc = zeros(2,1);
  % x_c ∈ E_P
6 receive(y, 2); receive(yd, 3);
  % x_c ∈ E_P
7 while (1)
      % x_c ∈ E_P
8     yc = max(min(y - yd, 1), -1);
      % x_c ∈ E_P, y_c^2 ≤ 1
9     skip;
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10    u = Cc*xc + Dc*yc;
```

C. Converting Functions

The second step of our proof translation scheme is to convert function symbols from real to floating-point. We show how it is done instruction by instruction.

1) Invariant on u: The next instruction in the original proof scheme is:

```
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10    u = Cc*xc + Dc*yc;
      % (x_c; y_c) ∈ E_{Q_μ}, u^2 ≤ (C_c D_c) · Q_μ^{-1} · (C_c D_c)^{-1}
```

First of all, matrices C_c and D_c must be replaced by their floating-point counterparts $\overline{C_c}$ and $\overline{D_c}$ both in the program instruction Line 10 and the ensuing invariant. This invariant relies only on algebraic arguments and does not depend on the values in the matrices, it still holds considering exact arithmetic operations. But this is not sufficient: indeed, this instruction is a sum of matrix multiplications, i.e. a set of additions and multiplications on floating-point numbers that yield rounding errors.

We can notice that entering this instruction, the values of matrices $\overline{C_c}$, $\overline{D_c}$ and $\mathcal{E}_{Q_\mu}$ are known, and the values of x_c and y_c are bounded by the precondition

$$\begin{pmatrix} x_c \\ y_c \end{pmatrix} \in \mathcal{E}_{Q_\mu},$$

that is

$$0.000673593x_{c1}^2 + 0.000085523x_{c1}x_{c2} + 0.00246288x_{c2}^2 + 0.9991y_c^2 \leq 1. \quad (17)$$

From (17), we deduce:

$$\begin{cases} |x_{c1}| \leq 3 \cdot 10^5 \sqrt{\frac{13\,695}{829\,322\,227\,639}} < 38.5515 \\ |x_{c2}| \leq 10^5 \sqrt{\frac{33\,710}{829\,322\,227\,639}} < 20.1614 \\ |y_c| \leq \frac{100}{\sqrt{9991}} < 1.00046 \end{cases} \quad (18)$$

Here we are able to find algebraic solutions, but this may be impossible with ellipsoids of higher dimension. Still, we would be able to find bounds using numerical methods.

In floating-point arithmetic, rounding errors created by addition and multiplication operators can be bounded when the operands are known or bounded by (18), provided that overflow, underflow, and denormalized numbers do not occur [10], [11].

Here, we need to compute

$$\overline{C_c}x_c + \overline{D_c}y_c = \overline{C_c}_{(0,0)}x_{c1} + \overline{D_c}y_c$$

where all values in the right-hand term are known or bounded. Thus, a constant ε can be computed that bounds the absolute rounding error created when computing u. We obtain:

$$\varepsilon = 5.90 \cdot 10^{-12}$$

This way, starting from the algebraic result obtained on real numbers

$$|u| \leq \sqrt{(C_c\ D_c) \cdot Q_\mu^{-1} \cdot (C_c\ D_c)^{\mathrm{T}}}$$

we can ensure that with floating-point numbers, the inequality holds:

$$|u| \leq \bar{U} = \sqrt{(\overline{C_c}\ \overline{D_c}) \cdot Q_\mu^{-1} \cdot (\overline{C_c}\ \overline{D_c})^{\mathrm{T}}} + \varepsilon$$

which leads to the invariants:

```
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10    u = Cc*xc + Dc*yc;
      % (x_c; y_c) ∈ E_{Q_μ}, u^2 ≤ Ū^2
```

2) Invariant on x_c: The next instruction, considering constant changes, is:

```
11    xc = Ac*xc + Bc*yc;
      % x_c ∈ Ẽ_R̃, R̃ = [(Ā_c B̄_c) · Q_μ^{-1} · (Ā_c B̄_c)^T]^{-1},
```

where $\tilde{R}$ is defined the same way R is, using floating-point terms $\overline{A_c}$, $\overline{B_c}$ instead of the real-valued counterparts A_c, B_c, and $\tilde{\mathcal{E}}_R$ is the ellipsoid built upon $\tilde{R}$. Again, this invariant holds independently of matrices values.

Here, we compute the values affected to $x_c = \binom{x_{c1}}{x_{c2}}$:

$$\begin{cases} \overline{A_{c(0,0)}} x_{c1} + \overline{A_{c(0,1)}} x_{c2} + y_c \\ \overline{A_{c(1,0)}} x_{c1} + x_{c2} \end{cases}$$

Using the same method as above, absolute rounding errors introduced by floating-point operations can be bounded on dimensions x_{c1} and x_{c2} by constants

$$\varepsilon_1 = 7.42 \cdot 10^{-15} \qquad \text{and} \qquad \varepsilon_2 = 3.62 \cdot 10^{-15}.$$

These constants must be taken into account in the postcondition. Then the postcondition can be replaced by

$$\% \ x_c \in \overline{\mathcal{E}_R}$$

where $\overline{\mathcal{E}_R}$ is an ellipse that includes $\widetilde{\mathcal{E}_R}$ plus the rounding error terms (see Figure 7). As mentioned in Section IV-B, replacing the ellipse in the postcondition by another ellipse has the advantage of introducing little change in the stability proof sketch (instead of using a different domain, which would involve using different theorems), which can greatly facilitate tweaking the rest of the proof in longer codes. Formally, $\overline{\mathcal{E}_R}$ must satisfy:

$$\forall x_c \in \widetilde{\mathcal{E}_R}, \forall {x_c}' \in \mathbb{R}^2, \\ |{x_{c1}}' - x_{c1}| \leq \varepsilon_1 \wedge |{x_{c2}}' - x_{c2}| \leq \varepsilon_2 \Longrightarrow {x_c}' \in \overline{\mathcal{E}_R} \quad (19)$$

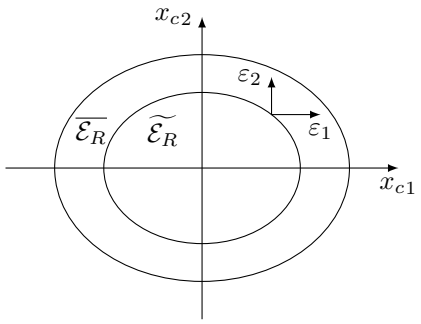

Figure 7. Relation between $\widetilde{\mathcal{E}_R}$ and $\overline{\mathcal{E}_R}$

At the end of the proof scheme, the system is stable with floating-point numbers if and only if the inclusion

$$\overline{\mathcal{E}_R} \subset \mathcal{E}_P$$

holds. To succeed, $\overline{\mathcal{E}_R}$ should be as narrow as possible with respect to Equation (19). This is not a clear criterion, as several shapes are possible for $\overline{\mathcal{E}_R}$ with no clear winner. We propose to define $\overline{\mathcal{E}_R}$ as the smallest homothety of $\widetilde{\mathcal{E}_R}$ centered around 0 that satisfies (19). It can be computed rather easily, for any number of dimension; we give details for two dimensions.

Let a, b, c be the coefficients of $\widetilde{\mathcal{E}_R}$:

$$\widetilde{\mathcal{E}_R} = \{(x_{c1}, x_{c2}) \,|\, a{x_{c1}}^2 + b{x_{c2}}^2 + cx_{c1}x_{c2} \leq 1\}.$$

a, b and c are known, positive values. Then there exists $k \geq 0$ such that

$$\overline{\mathcal{E}_R} = \{(x_{c1}, x_{c2}) \,|\, a{x_{c1}}^2 + b{x_{c2}}^2 + cx_{c1}x_{c2} \leq k\}.$$

As $\overline{\mathcal{E}_R}$ is wider than $\widetilde{\mathcal{E}_R}$, $k \geq 1$. We need a condition on k that guarantees (19).

We consider a point (x_{c1}, x_{c2}) located on the border of $\widetilde{\mathcal{E}_R}$:

$$a{x_{c1}}^2 + b{x_{c2}}^2 + cx_{c1}x_{c2} = 1. \quad (20)$$

By construction, for any values $e_1, e_2 \in \mathbb{R}$ such that $|e_1| \leq \varepsilon_1 \wedge |e_2| \leq \varepsilon_2$ the relation

$$(x_{c1} + e_1, x_{c2} + e_2) \in \overline{\mathcal{E}_R}$$

must hold, that is to say:

$$a(x_{c1} + e_1)^2 + b(x_{c2} + e_2)^2 + c(x_{c1} + e_1)(x_{c2} + e_2) \leq k.$$

It develops into

$$(a{x_{c1}}^2 + b{x_{c2}}^2 + cx_{c1}x_{c2}) + (2ae_1 + ce_2)x_{c1} + \\ (2be_2 + ce_1)x_{c2} + (ae_1^2 + be_2^2 + ce_1e_2) \leq k,$$

that is

$$1 + (2ae_1 + ce_2)x_{c1} + (2be_2 + ce_1)x_{c2} + \\ (ae_1^2 + be_2^2 + ce_1e_2) \leq k$$

due to (20).

Greatest values for the left-hand term are reached with $|e_1| = \varepsilon_1 \wedge |e_2| = \varepsilon_2$, depending on the signs of x_{c1} and x_{c2}. As the ellipse $\widetilde{\mathcal{E}_R}$ is symmetric around the origin point $(0, 0)$, we can set $e_1 = \varepsilon_1$, which leaves only two cases to study. Finally, we numerically verify that greatest values of the term are reached when $e_2 = \varepsilon_2$. This is the only case we detail here.

We can write:

$$1 + \alpha x_{c1} + \beta x_{c2} + \gamma \leq k$$

with values $\alpha = (2a\varepsilon_1 + c\varepsilon_2)$, $\beta = (2b\varepsilon_2 + c\varepsilon_1)$ and $\gamma = (a\varepsilon_1^2 + b\varepsilon_2^2 + c\varepsilon_1\varepsilon_2)$.

We know that x_{c1} and x_{c2} are bounded, thus so is the term $\alpha x_{c1} + \beta x_{c2}$: we can compute a minimum bound δ such that $\alpha x_{c1} + \beta x_{c2} \leq \delta$. So it is sufficient that k satisfies:

$$k \geq 1 + \gamma + \delta.$$

Consequently, the smallest homothety of $\widetilde{\mathcal{E}_R}$ that satisfies (19) is obtained with $k = 1 + \gamma + \delta$; we take it as our definition of $\overline{\mathcal{E}_R}$. The instruction becomes:

```
11    xc = Ac*xc + Bc*yc;
      % x_c ∈ E_R
```

In our case, starting from the ellipse

$$\widetilde{\mathcal{E}_R} = \{(x_{c1}, x_{c2}) \,|\, 0.00269007{x_{c1}}^2 + \\ 0.000341414x_{c1}x_{c2} + 0.00247323{x_{c2}}^2 \leq 1\}$$

we get the following values:

$\alpha = 1.03246 \cdot 10^{-17}, \beta = 1.84829 \cdot 10^{-17}, \gamma = 7.17582 \cdot 10^{-32}$.

Our program finds $\delta = 5.35754 \cdot 10^{-16} \gg \gamma$ and finally

$$k = 1 + 5.35754 \cdot 10^{-16}$$

that gives $\overline{\mathcal{E}_R}$.

3) End of Proof Scheme: Then, what remains of the stability proof scheme becomes:

```
      % x_c ∈ overline(E_R)
12    send(u, 1);
      % x_c ∈ overline(E_R)
13    receive(y, 2);
      % x_c ∈ overline(E_R)
14    receive(yd, 3);
      % x_c ∈ overline(E_R)
15    skip;
      % x_c ∈ E_P
16 end
```

At this stage, the final assertion $\overline{\mathcal{E}_R} \subset \mathcal{E}_P$ holds and is checked successfully.

VI. Closed-Loop Stability Proof Scheme with Floating-Point Numbers

We now show how the proof of state boundedness of the closed-loop system specifications can be migrated to the level of the controller code and executable model of the system. To be more precise, we exploit the invariance of the ellipsoid $\mathcal{E}_P$ to develop a proof of proper behavior, that is, stability and variable boundedness, for the computer program that implements the controller as it interacts with the physical system. Unlike the developments related to the open-loop controller, this proof necessarily involves the presence of a model of the physical system. In [5], Feron chooses to represent the physical system and the computer program by two concurrent programs. The code for controller dynamics is unchanged, same as in Section II. The pseudo-code to represent the physical system dynamics is shown below.

```
1 Ap = [1.0000, 0.0100; -0.0100, 1.0000];
2 Bp = [0.00005; 0.01];
3 Cp = [1, 0];
4 while (1)
5    yp = Cp * xp;
6    send(yp, 2);
7    receive(up, 1);
8    xp = Ap * xp + Bp * up;
9 end
```

In this scheme, the computer program representation of the physical system is to remain unchanged, since it only exists for modeling purposes and does not correspond to any actual program, whereas the controller code is allowed to evolve to reflect the various stages of its implementation.

Establishing proofs of stability of the closed-loop system at the code level is necessarily tied to understanding the joint behavior of the controller and the plant. The entire state space therefore consists of the direct sum of the state spaces of the controller and the physical system. The approach described in the previous sections is used to document the corresponding system of two processes. One interesting aspect of these processes is their concurrency, which can complicate the structure of the state transitions. However, a close inspection of the programs reveals that the transition structure of the processes does not need to rely on the extensions of Hoare's logic to concurrent programs: one program at a time is running, through the blocking nature of the `receive` primitive.

Feron's stability proof with real numbers is much longer than for the open-loop system. We do not detail it, the interested reader is referred to [5] for full information. To be noticed, the resulting comments are not much more complex than those available from the study of the controller alone. On the good side, as already mentioned, the Hoare formalism is not significantly affected by the concurrent structure of the closed-loop system.

A floating-point representation of the closed-loop system consists of keeping the listing corresponding to the physical system in its original settings, while replacing the controller part with the corresponding floating-point implementation, as we did in Section V. Using similar techniques to the study of the controller alone, proof invariants can be tweaked to take into account constant changes and rounding errors resulting from the use of floating-point arithmetic in the controller and real arithmetic in the plant.

Unfortunately, these invariants are not sufficient to show that the stability condition holds at the end of the loop body in the case of 64-bit floating-point numbers. In this case, our tool cannot prove the system stability on a double-precision floating-point architecture: either the system is not stable with the floating-point based controller, and in this case the proof parameters ($\mathcal{E}_P$, μ, ...) must be chosen more carefully by the controller designer, or the stability holds but we lost it by overapproximating the errors. Tuning the precision of floating-point arithmetic, as described in Section V-A, learns us that the controller stability holds on a quadruple-precision platform (i.e., with 128 bits). Thus, an alternative solution might be to use a microcontroller device with superior precision.

VII. Alternative Limited-Precision Arithmetics

Our general idea is to replace some of the invariants in the original proof scheme by wider ones that include rounding errors, with the hope that the stability condition is strong enough and still holds. This approach is made possible by the fact that the rounding errors introduced by the operations used in the code are bounded on bounded inputs and bounded state variables.

In previous sections, we mostly dealt with the case of a floating-point representation of real numbers. They are not available on all architectures, especially on microcontrollers that are commonly used to implement control systems. In this section, we quickly discuss alternative real-number representations.

- We can deal with fixed-point arithmetic the same way we do with floating-point, as long as we stand far enough from extremal values that can lead to overflows or extremely large error terms;
- Another way to represent real numbers is to use two integers, a numerator and a denominator. Considering that the input values are exact, the elementary operations do not introduce rounding

errors but can easily lead to overflows, e.g. when computing

$$\frac{p_1}{q_1} + \frac{p_2}{q_2} = \frac{p_1 q_2 + p_2 q_1}{q_1 q_2}.$$

A strategy must be used to prevent overflows by introducing approximations: in this case, the question is to quantify the errors introduced by these approximations.

In our example, we exclusively used additions and multiplications: divisions are not involved in linear control. Still, programs with divisions can also be analyzed, if the numerator can be shown to be far enough from zero: it is a supplementary constraint, but it is reasonable to assume that it should be respected on a realistic control system that uses divisions. Differentiable, periodic functions such as (sin) can be computed with an abacus and an interpolation function, thus with bounded error. In the same way, functions not periodic, but restricted to finite domains, can also be approximated. Other functions, such as tangent or square root, could raise more issues.

VIII. Conclusion

In this paper, we described a theoretical framework to translate proof invariants on code with real arithmetic to similar invariants on limited-precision numbers, while preserving the overall proof structure. We focused on the case of Lyapunov-theoretic proofs of stability with floating-point numbers, for which we implemented LyaFloat, a tool that automatically generates correct invariants for floating-point arithmetic from the provided invariants on real arithmetic, attempts to check whether stability holds and computes the required amount of precision needed for this. We used this tool to analyze Feron's motivating proof of stability, thus addressing the issue of arithmetic accuracy when implementing high-level, proved code on a microcontroller.

Many directions are open for future work. A first idea would be to implement the translation program in Coq rather than Python. This would grant an additional layer of safety, at the price of a deeper formalizing of invariant propagators — or even better proving them — and dealing with the quirks of the formalization of floating-point arithmetic, e.g. using the Flocq library [12]. Alternatively, our tool could simply generate invariants in the form of proof terms that could be checked with Coq or another floating-point-compliant proof checker. We also plan to extend the application scope of our tool, enabling it to analyze a wider range of control programs (which means handle more functions and propagation theorems), so that we could apply it to some of those real-life controllers that come with Lyapunov stability proofs [13]. Finally, it would be interesting to be able to deal with different arithmetic paradigms, as discussed in Section VII.

References

[1] P. Cousot, R. Cousot, J. Feret, A. Miné, and X. Rival, "The astrée static analyzer," 2013. [Online]. Available: http://www.astree.ens.fr/

[2] INRIA, "The coq proof assistant," 2013. [Online]. Available: http://coq.inria.fr/

[3] L. Paulson, T. Nipkow, and M. Wenzel, "Isabelle," 2013. [Online]. Available: http://isabelle.in.tum.de/

[4] S. Owre, "PVS specification and verification system," 2013. [Online]. Available: http://pvs.csl.sri.com/

[5] E. Feron, "From control systems to control software," *IEEE Control Systems Magazine*, vol. 30, no. 6, pp. 50–71, Dec. 2010. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=5643477

[6] D. Peled, *Software reliability methods*. Springer, 2001.

[7] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan, *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics, 1994. [Online]. Available: http://epubs.siam.org/doi/abs/10.1137/1.9781611970777

[8] Wolfram Research, "Mathematica," 2013. [Online]. Available: http://www.wolfram.com/mathematica/

[9] IEEE Computer Society, "IEEE standard for floating-point arithmetic," Tech. Rep., Aug. 2008. [Online]. Available: http://ieeexplore.ieee.org/lpdocs/epic03/wrapper.htm?arnumber=4610935

[10] D. Goldberg, "What every computer scientist should know about floating-point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, Mar. 1991. [Online]. Available: http://portal.acm.org/citation.cfm?doid=103162.103163

[11] N. J. Higham, *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[12] S. Boldo and G. Melquiond, "Flocq," 2010. [Online]. Available: http://flocq.gforge.inria.fr/

[13] P. Martin and E. Salaün, "Estimation d'état pour les drones aériens," Tech. Rep., 2009.