Code-Level Formal Verification of Ellipsoidal Invariant Sets for Linear Parameter-Varying Systems

Elias Khalife, Pierre-Loïc Garoche, Mazen Farhood

Code-Level Formal Verification of Ellipsoidal Invariant Sets for Linear Parameter-Varying Systems*

Elias Khalife[1(✉)], Pierre-Loic Garoche[2], and Mazen Farhood[1]

[1] Virginia Tech
Kevin T. Crofton Department of Aerospace and Ocean Engineering
{eliask,farhood}@vt.edu
[2] Ecole Nationale de l'Aviation Civile
pierre-loic.garoche@enac.fr

Abstract. This paper focuses on the formal verification of invariant properties of a C code that describes the dynamics of a discrete-time linear parameter-varying system with affine parameter dependence. The C code is annotated using ACSL, and the Frama-C's WP plugin is used to transform the annotations and code into proof objectives. The invariant properties are then formally verified in both the real and float models using the polynomial inequalities plugin of the theorem prover Alt-Ergo. The challenges of verifying the invariant properties in the float model are addressed by utilizing bounds on numerical errors and incorporating them into the real model.

Keywords: Deductive Method · Static Analysis · Invariant Set · Linear Parameter-Varying System · Frama-C.

1 Introduction

Ellipsoidal invariant sets constitute an important concept in the field of control theory, specifically in the context of dynamical systems and system stability analysis. These sets are defined by the property that all state trajectories starting from any point within the set remain inside the set for all future times. In other words, if the system's state lies initially inside an ellipsoidal invariant set, then the state evolution is guaranteed to stay within the boundaries of the set. Similarly, invariants in the field of formal methods refer to properties or conditions that hold true throughout the entire or part of the execution of a program, system, or algorithm [26]. The relationship between the two concepts is evident when considering that if a state is inside the ellipsoidal invariant set, then the next states will also be inside of this set. This situation is akin to an invariant property, where the current state being inside the ellipsoidal invariant

* This work was supported by the Office of Naval Research under Award No. N00014-18-1-2627, the Army Research Office under Contract No. W911NF-21-1-0250, and the ANR-17-CE25-0018 FEANICSES project.

set acts as a precondition that implies the next states will remain within the set. There has been a significant amount of research on ellipsoidal invariant sets in the literature. Early work in this area focused on the use of ellipsoidal invariant sets for analyzing the stability of linear systems [20, 5]. Several methods have been developed to construct invariant sets, including methods based on Lyapunov's stability theorem [18], linear matrix inequality (LMI) techniques [6], and sum-of-squares (SOS) programming [30]. Each of these methods has its own benefits and limitations, and the choice of method depends on the specific characteristics of the system being analyzed.

This paper focuses on the formal verification of some invariant properties of the C code describing the dynamics of a discrete-time linear parameter-varying (LPV) system with affine parameter dependence. Specifically, we formally verify that, if the state of the system lies in an ellipsoidal invariant set at the initial time, then it resides in this set at all time instants, and, further, the output of the system resides in another ellipsoid at all time instants as well for all permissible pointwise-bounded inputs and parameter trajectories. These sets are obtained by applying new results developed in [17] for computing state- and output-bounding sets for discrete-time uncertain linear fractional transformation (LFT) systems using pointwise integral quadratic constraints (IQCs) to characterize the uncertainties and the S-procedure. Uncertainties that admit pointwise IQC characterizations include static linear time-invariant and time-varying perturbations, sector-bounded nonlinearities, and uncertain time-varying time-delays. An affine LPV system can be expressed as an LFT on static linear time-varying perturbations, and so the aforementioned results are applicable in our case. The positive definite matrices defining the ellipsoids are obtained by solving semidefinite programs [7]. These solutions of the semidefinite programs, obtained by applying the IQC-based analysis approach, serve as a certificate that proves that the system satisfies the desired properties at the algorithmic level. Moreover, these solutions can be employed to annotate the C code describing the system dynamics with logical expressions, which indicate the set of reachable program states. The annotations are done in ACSL (ANSI/ISO C Specification Language) [3], Frama-C's formal annotation language. Additionally, we utilize WP, a Frama-C plugin based on the weakest precondition calculus and deductive methods, to transform annotations and code into proof objectives. Thus, the software verification in our case focuses on translating the guarantees obtained at the algorithmic level, using the analysis results from [17], and expressing them at the code level. Then, we revalidate the invariant properties at the code level using Alt-Ergo-Poly [28], an extension of the SMT solver Alt-Ergo [8] with a sound Sum-of-Squares solver [27, 22], to discharge positive polynomial constraints. Last, we instrument the contract to account for floating-point errors in the code, ensuring the validity of our contracts despite the noise caused by floating-point inaccuracy.

One of the motivations for this work is analyzing the C code of gain-scheduled controllers, for instance, the robustly stable LPV path-following controller designed in [24] for a small, fixed-wing, unmanned aircraft system (UAS), where

the scheduling parameter is the inverse of the radius of curvature of the path to be traversed. If the output-bounding ellipsoid in this case lies within the actuator saturation limits, then we have a guarantee that the actuators would not saturate for the considered pointwise-bounded measurements.

The paper is structured as follows. In Section 2, we introduce affine LPV systems and explain how to determine state and output invariant ellipsoids. In Section 3, we outline the steps for setting up the necessary Frama-C environment to formally verify the invariant properties at code level. In Section 4, we demonstrate the formal verification of these properties using the real model. In Section 5, we present the verification of these properties in the float model, which involves the use of bounds on numerical errors and their integration into the real model. The paper concludes with Section 6.

2 Affine LPV Systems and Ellipsoidal Invariant Sets

Consider a stable discrete-time LPV system G described by

$$\begin{aligned} x(k+1) &= A(\delta(k))x(k) + B(\delta(k))d(k), \\ y(k) &= C(\delta(k))x(k) + D(\delta(k))d(k), \end{aligned} \tag{1}$$

where $x(k) \in \mathbb{R}^{n_x}$, $y(k) \in \mathbb{R}^{n_y}$, $d(k) \in \mathbb{R}^{n_d}$, and $\delta(k) = (\delta_1(k), \ldots, \delta_{n_\delta}(k)) \in \mathbb{R}^{n_\delta}$ designate the values of the state, output, input, and scheduling parameters at the time instant k, respectively, where k is a nonnegative integer. The state-space matrix-valued functions of G are assumed to have affine dependence on the scheduling parameters; for instance, the state matrix $A(\delta(k))$ can be expressed as

$$A(\delta(k)) = A_0 + \sum_{i=1}^{n_\delta} \delta_i(k) A_i, \tag{2}$$

where the matrices A_i are known and constant for $i = 0, \ldots, n_\delta$, and the scheduling parameters $\delta_i(k) \in [\underline{\delta}_i, \bar{\delta}_i]$ for all integers $k \geq 0$ and $i = 1, \ldots, n_\delta$. The analysis results used in this paper also allow imposing bounds on the parameter increments $d\delta_i(k) = \delta_i(k+1) - \delta_i(k)$ for $i = 1, \ldots, n_\delta$ and all integers $k \geq 0$.

The analysis results in [17] are used to determine the state-invariant and output-bounding ellipsoids of system G. To apply these results, system G is first expressed as a linear fractional transformation (LFT) on uncertainties, where the uncertainties in this case are the static linear time-varying perturbations δ_i for $i = 1, \ldots, n_\delta$. That is, system G is expressed as an interconnection of a stable nominal linear time-invariant (LTI) system and an uncertainty operator. The set of allowable uncertainty operators is described using the so-called IQC multipliers. Namely, an IQC multiplier is used to define a quadratic constraint that the input and output signals of the uncertainty operator must satisfy. In the work [17], this quadratic constraint must be satisfied at every time instant and is hence referred to as a pointwise IQC. A pointwise IQC is more restrictive than the standard IQC [23], which involves an infinite summation of quadratic terms.

However, the uncertainty set in our problem admits a pointwise IQC characterization. The approach in [17] allows representing the exogenous input d as a pointwise bounded signal, where its value lies in some closed, convex polytope $\mathit{\Gamma}$ for all time instants or an ellipsoid $\mathcal{E}$. While the analysis conditions provided in [17] are generally nonconvex, they can be relaxed into convex conditions by applying the multiconvexity relaxation technique [1], along with gridding. Thus, the positive definite matrices defining the state-invariant and output-bounding ellipsoids can be obtained by solving semidefinite programs.

Let $\mathcal{D}$ be the set of admissible inputs of system G and $X \in \mathbb{S}^n_{++}$, where $\mathbb{S}^n_{++}$ denotes the set of $n \times n$ positive definite matrices. With every $X \in \mathbb{S}^n_{++}$, we associate an ellipsoid $\mathcal{E}_X := \{x \in \mathbb{R}^n \,|\, x^T X x \leq 1\}$, whose shape and orientation are determined by X. Let $\mathcal{E}_P$ and $\mathcal{E}_Q$ be the state-invariant and output-bounding ellipsoids, respectively, obtained by applying the results of [17], where $P \in \mathbb{S}^{n_x}_{++}$ and $Q \in \mathbb{S}^{n_y}_{++}$. This means that, if $x(k) \in \mathcal{E}_P$, then $x(k+1) \in \mathcal{E}_P$ and $y(k) \in \mathcal{E}_Q$ for any integer $k \geq 0$, all $d(k) \in \mathcal{D}$, and all admissible $\delta(k)$. The objective of this paper is to formally verify that the ellipsoids $\mathcal{E}_P$ and $\mathcal{E}_Q$ are state-invariant and output-bounding, respectively, for the affine LPV system G under all admissible inputs $\mathcal{D}$ and all possible scheduling parameters. These properties will be referred to as the state and output invariant properties in the rest of the paper.

3 Frama-C Setup

Frama-C is a suite of tools for the analysis of the source code of software written in C. These tools can be used for tasks such as static analysis, automatic proof generation, testing, and more [9]. In the following, we will use ACSL (ANSI/ISO C Specification Language), Frama-C's formal annotation language, as well as WP, a Frama-C plugin that relies on weakest precondition calculus and deductive methods, to transform annotations and code into proof objectives that are later solved by SMT solvers such as Z3 [12], CVC4 [2], or Alt-Ergo [8]. ACSL is a specification language that can be used to annotate C code and provide precise, machine-readable descriptions of the behavior of C functions and other code elements [4]. These annotations can be used by Frama-C and other tools to perform various kinds of analysis. In Frama-C, ACSL annotations can be used to specify properties of C code, such as preconditions and postconditions for functions, invariants for loops, and more. These annotations can then be checked by the Frama-C tools to ensure that the code adheres to the specified properties. This can be especially useful for developing safety-critical software, where it is important to ensure the code behaves correctly under all possible circumstances.

3.1 C Code of System Dynamics

To express the dynamics of a discrete-time system G in C, we define the function "`updateState`" that updates the state vector of the system and the function "`updateOutput`" that computes the output vector at the current time-step. These functions use the state and output equations in (1).

In the following code, the "`struct state`" defines a new data type that represents the state vector of the system. It has n_x fields: $\mathtt{x}_1$, ..., $\mathtt{x}_{n_x}$, which correspond to the n_x state variables of the system. Similarly, the "`struct output`" defines a new data type that has n_y fields: $\mathtt{y}_1$, ..., $\mathtt{y}_{n_y}$, which correspond to the n_y output variables of the system. The `updateOutput` function takes in the current state of the system `x`, the current input variables $\mathtt{d}_1$, ... , $\mathtt{d}_{n_d}$, and the current values of the scheduling parameters δ_1, ... , δ_{n_δ}. It computes the output vector of the system and stores the result in a "`struct output`" called `y`, following the output equation in (1). The `updateState` function takes in the same inputs as the previous function, stores the values of the current state variables in temporary variables (pre_$\mathtt{x}_1$, ... , pre_$\mathtt{x}_{n_x}$), and computes the next state of the system based on the difference state equation in (1). The state vector at the next time-step is then stored in the "`struct state`" `x`.

C

```
typedef struct { double x_1, ..., x_{n_x}; } state;
typedef struct { double y_1, ..., y_{n_y}; } output;

void updateOutput(state *x, output *y, double d_1, ...,
    double d_{n_d}, double δ_1, ..., double δ_{n_δ}){
    //Compute the output
    y->y_1 = ...;
    y->y_{n_y} = ...;}
void updateState(state *x, double d_1, ..., double d_{n_d}, double
     δ_1, ..., double δ_{n_δ}){
    //Store the current state in temporary variables
    double pre_x_1 = x->x_1, ..., pre_x_{n_x} = x->x_{n_x};
    //Compute the next state
    x->x_1 = ...;
    x->x_{n_x} = ...;}
```

3.2 Invariant Set ACSL Annotation

Let $X \in \mathbb{S}^n_{++}$, then a vector $z \in \mathbb{R}^n \in \mathcal{E}_X$ if and only if

$$z^T X z = \sum_{i=1}^{n} X_{ii} z_i^2 + 2 \sum_{i=1}^{n-1} \sum_{j=i+1}^{n} X_{ij} z_i z_j \leq 1. \tag{3}$$

The invariant properties of the state-invariant ellipsoid $\mathcal{E}_P$ and the output-bounding ellipsoid $\mathcal{E}_Q$ must be annotated in ACSL to enable Frama-C to ensure that the codes adhere to them. This is achieved by defining the predicates `stateinv` and `outputinv` in ACSL as follows:

ACSL

```
//@ predicate stateinv(real x_1, ..., real x_{n_x}, real λ) =
    (P_11 * x_1 * x_1 + 2 * P_12 * x_1 * x_2 + ... + P_{n_x n_x} * x_{n_x} * x_{n_x}
        <= λ);
//@ predicate outputinv(real y_1, ..., real y_{n_y}, real λ) =
```

```
(Q_11 * y_1 * y_1 + 2 * Q_12 * y_1 * y_2 + ...+ Q_{n_y n_y} * y_{n_y} * y_{n_y}
    <= λ);
```

The predicate `stateinv` takes in the elements of the state vector at a given time instant along with a scalar λ. Similarly, `outputinv` takes in the elements of the output vector at a given time instant along with a scalar λ.

For λ equal to 1, the `stateinv` and `outputinv` predicates correspond to (3) with $X = P$ and $X = Q$, respectively. In this case, when true, these predicates imply that the vectors $x = \left[\mathtt{x}_1, \ldots, \mathtt{x}_{n_x}\right]^T$ and $y = \left[\mathtt{y}_1, \ldots, \mathtt{y}_{n_y}\right]^T$ belong to $\mathcal{E}_P$ and $\mathcal{E}_Q$, respectively.

Remark 1. The ACSL language allows predicates to be defined directly on C structs or pointers, but doing so may make it more difficult for automated solvers to prove the generated proof obligations. Based on our observation, it is more effective to define the predicate in a parameterized form, using all of the state/output variables as parameters. This approach may be more amenable to automated proofs and may improve the ability of automated solvers to prove the proof obligations. Note, however, that this observation may change with future versions of the tool or improvements in the solvers.

3.3 Contract-Based Verification

A contract is a set of preconditions, postconditions, and other specifications that describe the expected behavior of a piece of software. Preconditions are conditions that must be met in order for the software to be used correctly, and postconditions are conditions that must be satisfied after the software has been used. Contract-based verification is important for ensuring that the software behaves correctly and that certain properties are maintained under different possible circumstances. In ACSL, preconditions are expressed using the `requires` and `assumes` commands, while postconditions are expressed using the `ensures` command. Consider the simplest contract `// requires P; ensures E;`. It is equivalent to the contract `// requires \true; ensures \old(P)==>E;`, where `\old(P)` denotes the evaluation of predicate `P` before the execution of the function.

As outlined in the ACSL manual [3, §2.3.3], we can rely on named behaviors to structure requirements. For example, the contract sketched in Fig. 1 amounts to requiring property `P` to hold for all cases but only requiring property `E1` to hold when the precondition `A1` is valid. It is syntactic sugar to express `// ensures \old(P)==> ((\old(A1) ==> E1)&& (\old(A2)==> E2));`. This use of named behaviors allows to separate concerns and prevent a non-proven behavior from negatively impacting the analysis of other behaviors.

ACSL

```
/*@ requires P;
  @ behavior b1:
  @   assumes A1;
  @   ensures E1;
  @ behavior b2:
  @   assumes A2;
  @   ensures E2; */
```

Fig. 1. Behaviors in ACSL contracts

Remark 2 (Beware of pointers and use of `\old()`). `\old()` must be used with care since `\old(x)->`$\mathtt{x}_0$ denotes the value of field $\mathtt{x}_0$ for the previous value of the pointer `x`, while `\old(x->`$\mathtt{x}_0$`)` denotes the previous value of the field $\mathtt{x}_0$.

In the upcoming contracts, we assume that the scheduling parameter $\delta_i(k)$ belongs to $[\underline{\delta}_i, \bar{\delta}_i]$, for $i = 1, \ldots, n_\delta$ and a given time-step k. Now, we define the preconditions of our various contracts.

Zero Input Contract: The input set $\mathcal{D} = \{0\}$, i.e., for a given time-step k, if $d(k) \in \mathcal{D}$, then $d_i(k) = 0$ for $i = 1, \ldots, n_d$.

ACSL

```
/*@ behavior zero_input_contract:
      assumes d_1 == 0 && ... && d_{n_d} == 0;
      assumes δ̲_1 <= δ_1 <= δ̄_1 && ... && δ̲_{n_δ} <= δ_{n_δ} <= δ̄_{n_δ}; */
```

Polytope Bounded Input Contract: The input set $\mathcal{D}$ is a polytope. Particularly, if $\mathcal{D}$ is a hyper-rectangle defined such that, given a time-instant k, $d_i(k)$ belongs to $[\underline{d}_i, \bar{d}_i]$ for $i = 1, \ldots, n_d$, we write the following contract:

ACSL

```
/*@ behavior polytope_input_contract:
      assumes d̲_1 <= d_1 <= d̄_1 && ... && d̲_{n_d} <= d_{n_d} <= d̄_{n_d};
      assumes δ̲_1 <= δ_1 <= δ̄_1 && ... && δ̲_{n_δ} <= δ_{n_δ} <= δ̄_{n_δ}; */
```

Ellipsoid Bounded Input Contract: The input set $\mathcal{D}$ is an ellipsoid $\mathcal{E}_M$, where $M \in \mathbb{S}^{n_d}_{++}$. In this case, the predicate `ellipinput` must be defined similarly to the `stateinv` and `outputinv` predicates in Section 3.2, and the contract is expressed as follows:

ACSL

```
//@ predicate ellipinput(real d_1, ..., real d_{n_d}, real λ) =
    (M_11 * d_1 * d_1 + 2 * M_12 * d_1 * d_2 + ... + M_{n_d n_d} * d_{n_d} * d_{n_d}
        <= λ);
/*@ behavior ellipsoid_input_contract:
      assumes ellipinput(d_1, ..., d_{n_d}, 1);
      assumes δ̲_1 <= δ_1 <= δ̄_1 && ... && δ̲_{n_δ} <= δ_{n_δ} <= δ̄_{n_δ}; */
```

4 Validating Contracts: Real Model

In Frama-C, the real model is based on the mathematical model of real numbers. As a result, single and double precision floating-point numbers are mapped to real types in proof objectives. This simplification can make the proof process easier, but it does not take into account the actual computation that is performed using machine-code floating-point numbers. This means that the real model may not accurately reflect the behavior of the system when it is implemented in machine code. Nevertheless, in our setting, using the real model is a reasonable first

step since the system analysis has been performed assuming real computation. The validity of the real model in our setting will be further addressed in the next section by taking into account the potential for numerical errors.

To validate the invariant properties of the system G, we combine the codes in Sections 3.1, 3.2, and 3.3, and we add the missing preconditions and postconditions.

C+ACSL

```
typedef struct { double x_1, ..., x_{n_x}; } state;
typedef struct { double y_1, ..., y_{n_y}; } output;
//@ predicate stateinv(real x_1, ..., real x_{n_x}, real λ) =
    (P_11 * x_1 * x_1 + 2 * P_12 * x_1 * x_2 + ... + P_{n_x n_x} * x_{n_x} * x_{n_x}
        <= λ);
//@ predicate outputinv(real y_1, ..., real y_{n_y}, real λ) =
    (Q_11 * y_1 * y_1 + 2 * Q_12 * y_1 * y_2 + ...+ Q_{n_y n_y} * y_{n_y} * y_{n_y}
        <= λ);

/*@ requires \valid(x) && \valid(y);
    requires \separated(&(x->x_1), ...,&(x->x_{n_x}),&(y->y_1), ...,
         &(y->y_{n_y}));
  assigns *y;
  behavior contract_name:
    assumes ...;
    ensures stateinv(\old(x->x_1),..., \old(x->x_{n_x}),1) ==>
        outputinv(y->y_1, ...,y->y_{n_y},1);
*/
void updateOutput(...) {...}

/*@ requires \valid(x);
    requires \separated(&(x->x_1), ...,&(x->x_{n_x}));
    assigns *x;
    behavior contract_name:
      assumes ...;
      ensures stateinv(\old(x->x_1), ..., \old(x->x_{n_x}),1) ==>
          stateinv(x->x_1, ...,x->x_{n_x},1);
*/
void updateState(...) {...}
```

In this code, the `\valid`, `\separated`, and `assigns` annotations are used for expressing constraints on the memory layout of the program and specifying which variables or memory locations may be modified by the code. Precisely, the `\valid` annotation is used to specify that a certain pointer or array refers to a valid, allocated region of memory, the `\separated` annotation is used to specify that certain variables or memory locations must be separated from each other in order for the code to be executed, and the `assigns` annotation is used to specify which variables or memory locations may be modified by the code. As shown in the above script, it is generally recommended to place an annotation before the code it is associated with.

To formally verify the invariant properties, we use the polynomial inequalities plugin of Alt-Ergo in the WP framework of Frama-C. This plugin, unlike other solvers, can deal with the type of predicates considered. The following command runs the formal verification process and returns the verification results:

```
frama-c -wp -wp-model real -wp-prover Alt-Ergo-Poly source_file.c
```

Many options and arguments can be used with the `frama-c -wp` command to customize and control the analysis process. For example, the `-wp-timeout` option is used to set a time limit for the analysis, which can be helpful in cases where the analysis is expected to take a long time. In our experiment, Alt-Ergo-Poly (the SOS plugin) was the only solver able to discharge any of our proof objectives. For instance, running the above command returned the following result:

```
[wp] 51 goals scheduled
[wp] Proved goals:   51 / 51
  Qed:              18  (2ms-7ms-14ms)
  Alt-Ergo-Poly :   33  (5ms-150ms-843ms) (3290)
```

The goals associated with the memory-related annotation were validated using the simpler internal solver Qed, while all the ellipsoid-related goals required the use of Alt-Ergo-Poly.

5 Validating Contracts: Float Model

In C, floating-point numbers are represented using a binary floating-point format, which is a method for representing real numbers with a fixed number of bits allocated to the mantissa (the fractional part of the number) and the exponent (the power of 2 by which the mantissa must be multiplied). The floating-point model in Frama-C adheres to the IEEE 754 standard for floating-point representation. This standard defines various floating-point formats for representing real numbers, including single-precision (32-bit) and double-precision (64-bit) formats.

5.1 Issues with Deductive Methods and the Floating-Point Model

While the float model is a more accurate representation of computation, it can produce proof goals that are more difficult to solve. This can be illustrated with the following simple example:

C+ACSL
```
/*@ requires x > 0 && x <= 10;
  @ ensures \result > 0; */
double f(double x) { return x + 0.25; }
```

This contract is easily solved using the real model:

```
% frama-c -wp -wp-model real -wp-prover z3,cvc4,alt-ergo simple.c
[wp] Proved goals:    1 / 1
------------------------------------------------------------
Prover Z3 4.11.2 returns Valid (Qed:0.81ms) (30ms) (21112)
Prover CVC4 1.8 returns Valid (Qed:0.81ms) (40ms) (5926)
Prover Alt-Ergo  returns Valid (Qed:0.81ms) (8ms) (8)
------------------------------------------------------------
```

However, if we analyze the same program using a more accurate encoding of floats, i.e., by omitting the option `-wp-model real`, we get

```
Prove: .0 < of_f64(add_f64(x, to_f64((1.0/4)))).
Prover Z3 4.11.2 returns Failed Unknown error
Prover CVC4 1.8 returns Timeout (Qed:2ms) (10s)
Prover Alt-Ergo  returns Timeout (Qed:2ms) (10s)
```

Given the limitations of automated provers in handling simple verification conditions involving floats, there are two primary alternatives to consider. One approach involves using proof assistants, like Gappa [11], which require more manual intervention but offer precise axiomatization of floating-point computations. Alternatively, static analysis tools such as FPTaylor [29], Fluctuat [13], and Rosa [10], which employ techniques like Taylor expansion or affine arithmetic, provide more systematic error bounding solutions.

In the following, instead of using the float model, we rely on such a static analysis to bound the numerical imprecision of the computation. For instance, in this example, using interval arithmetic, which will be discussed in Section 5.2, we can bound the values of `\result` by $[0.25, 10.25] + \pm 2.275958\text{E}{-15}$, where the first interval denotes the interval of double $[0.25, 10.25]$ and the term $\pm 2.275958\text{E}{-15}$ denotes the over-approximation of accumulated errors. The contract can then be instrumented, and the floating-point "noise" can be included in the `\ensures` statement as follows:

ACSL

```
/*@ requires x > 0 && x <= 10;
  @ ensures \forall real λ, -1 <= λ <= 1 ==> \result + λ *
      2.275958E-15 > 0; */
```

With this approach, we can use the real model for analysis to formally verify the postcondition in the float model:

```
% frama-c -wp -wp-model real -wp-prover z3,cvc4,alt-ergo simple2.c
[wp] Proved goals:    1 / 1
  Qed:             0  (2ms)
  Alt-Ergo :       1  (9ms) (12)
------------------------------------------------------------
Prover Alt-Ergo  returns Valid (Qed:2ms) (9ms) (12)
------------------------------------------------------------
```

$$([x_1,x_2],e)+([y_1,y_2],f)=\left(\begin{array}{l}[\mathrm{fl}\left(x_1+y_1\right),\mathrm{fl}\left(x_2+y_2\right)],e_{new}\\ \text{with } e_{new}=\max\left(\begin{array}{l}-\mathrm{fl}_{-\infty}\left(-e-f-e^{+}(x_1,y_1)\right),\\ \mathrm{fl}_{+\infty}\left(e+f+e^{+}(x_2,y_2)\right)\end{array}\right)\end{array}\right),$$

$$([x_1,x_2],e)*([y_1,y_2],f)=\left(\left[\begin{array}{l}\min(\mathrm{fl}\left(x_1y_1\right),\mathrm{fl}\left(x_1y_2\right),\mathrm{fl}\left(x_2y_1\right),\mathrm{fl}\left(y_1y_2\right)),\\ \max(\mathrm{fl}\left(x_1y_1\right),\mathrm{fl}\left(x_1y_2\right),\mathrm{fl}\left(x_2y_1\right),\mathrm{fl}\left(y_1y_2\right))\end{array}\right],e_{new}\right),$$

where $e^{+}(a,b)$ is defined as $(|a|+|b|)\texttt{eps}$ and $e^{*}(a,b)$ as $|a*b|\ \texttt{eps}+\texttt{eta}$ and with

$$e_{new}=\max\left(\begin{array}{l}-\mathrm{fl}_{-\infty}\left(\begin{array}{l}\min(\mathrm{fl}_{-\infty}\left(-ey_1\right),\mathrm{fl}_{-\infty}\left(-ey_2\right),\mathrm{fl}_{-\infty}\left(ey_1\right),\mathrm{fl}_{-\infty}\left(ey_2\right))\\ +\min(\mathrm{fl}_{-\infty}\left(-x_1f\right),\mathrm{fl}_{-\infty}\left(x_1f\right),\mathrm{fl}_{-\infty}\left(-x_2f\right),\mathrm{fl}_{-\infty}\left(x_2f\right))\\ -\min(e^{*}(x_1,y_1),e^{*}(x_1,y_2),e^{*}(x_2,y_1),e^{*}(x_2,y_2))\end{array}\right),\\ \mathrm{fl}_{+\infty}\left(\begin{array}{l}\max(\mathrm{fl}_{+\infty}\left(-ey_1\right),\mathrm{fl}_{+\infty}\left(ey_2\right),\mathrm{fl}_{+\infty}\left(ey_1\right),\mathrm{fl}_{+\infty}\left(ey_2\right))\\ +\min(\mathrm{fl}_{+\infty}\left(-x_1f\right),\mathrm{fl}_{+\infty}\left(x_1f\right),\mathrm{fl}_{+\infty}\left(-x_2f\right),\mathrm{fl}_{+\infty}\left(x_2f\right))\\ +\min(e^{*}(x_1,y_1),e^{*}(x_1,y_2),e^{*}(x_2,y_1),e^{*}(x_2,y_2))\end{array}\right)\end{array}\right).$$

Fig. 2. Addition and multiplication on intervals with floating-point errors.

5.2 Bounding Numerical Errors

We refer the reader to [21, 16, 15] for more details on means to bound floating-point accumulated rounding errors. We recall the characterization of floating-point values for addition and multiplication of floating-point numbers:

$$(u+e^u)+(v+e^v)=(u+v)+(e^u+e^v+e^{+}(u,v)), \tag{4}$$
$$(u+e^u)*(v+e^v)=(u*v)+(e^u*v+e^v*u+e^{*}(u,v)), \tag{5}$$

with $|e^{+}(u,v)|\leq|u+v|\ \texttt{eps}$ and $|e^{*}(u,v)|\leq|u*v|\ \texttt{eps}+\texttt{eta}$.

In the following discussions, $\mathrm{fl}\left(e\right)$ denotes the floating-point approximation of value e using a "round to the nearest" mode. Rounding towards $-\infty$ and $+\infty$ are denoted by $\mathrm{fl}_{-\infty}\left(\cdot\right)$ and $\mathrm{fl}_{+\infty}\left(\cdot\right)$, respectively. The constants `eps` and `eta` denote the precision of the floating-point format and its precision in case of underflows, respectively. For single precision floating-point numbers, $\texttt{eps}=2^{-22}\approx10^{-7}$ and $\texttt{eta}=2^{-149}\approx10^{-45}$, while for double precision, $\texttt{eps}=2^{-52}\approx10^{-16}$ and $\texttt{eta}=2^{-1074}\approx10^{-324}$.

Equations (4) and (5) can be adapted to intervals, as detailed in Figure 2. The interval $[a,b]$ with additional error $\pm e$ is denoted by $([a,b],e)$. This method allows to characterize both the actual values, obtained by floating-point computation in the value part, and a safe error term. In case of a deterministic loopless code computing an expression exp, one would obtain the abstract value $[x,x]\pm e$ where the singleton interval for the value part denotes exactly the value x that would have been obtained when computing $\mathrm{fl}\left(\mathrm{exp}\right)$. Thanks to the handling of floating-point errors, the computation of exp with reals is guaranteed to lie within $[\mathrm{fl}_{-\infty}\left(x-e\right),\mathrm{fl}_{+\infty}\left(x+e\right)]$.

5.3 Error Hyper-rectangle Approach

For the analysis that follows, we will assume that the initial state of the system G is represented by a floating-point number that belongs to the state-invariant ellipsoid. The vector $z \in \mathbb{R}^n$ serves as a placeholder for both the updated state and output vectors of the system. The floating-point representation $\mathrm{fl}(z)$ of the exact vector z satisfies the following inequalities:

$$z - e \preceq \mathrm{fl}(z) \preceq z + e, \tag{6}$$

where $\preceq$ denotes the componentwise inequality and $e = \left[e_1, \ldots, e_n\right]^T$ is the "error" vector whose i^{th} element is an over-approximation of the accumulated error associated with the computation of the i^{th} component of z using float model arithmetic. Consequently, it is clear that $\mathrm{fl}(z)$ belongs to a hyper-rectangle Γ that is symmetric about the exact vector z and that has 2^n vertices $\hat{z}_i$, where $i = 1, \ldots, 2^n$. Assume that $z \in \mathcal{E}_X$ is formally verified in the real model. Then, to prove that $\mathrm{fl}(z) \in \mathcal{E}_X$, it is sufficient to verify that $\Gamma \subset \mathcal{E}_X$. This sufficient condition can be established using either of the following two methods.

Method 1: Checking All Points in the Hyper-rectangle. The first method to verify that $\Gamma \subset \mathcal{E}_X$ is to formally verify that all the points in Γ belong to $\mathcal{E}_X$, i.e., for all $z_e \in \Gamma$, $z_e \in \mathcal{E}_X$. To express this condition in ACSL, we first need to know how to express all the vectors that belong to Γ. We notice that the i^{th} component of any vector z_e belonging to Γ can be expressed as $z_{e,i} = z + l_i e_i$, where $l_i \in [-1, 1]$ for $i = 1, \ldots, n$. This formulation of z_e can be expressed in ACSL using the universal quantifier $\forall$ (`\forall`) and n bound variables ($\mathtt{l}_1$, ..., $\mathtt{l}_n$), each belonging to $[-1, 1]$. For instance, in the case of formally verifying the state invariant property, the postcondition is the following:

ACSL
```
//State Invariant Postcondition
ensures \forall real l1; ...;\forall real lnx; -1 <= l1 <= 1
    ==> ... ==> -1 <= lnx <= 1 ==> stateinv(\old(x->x1), ...,
    \old(x->xnx),1) ==> stateinv(x->x1+l1*e1, ..., x->xnx+lnx*
    enx,1);
```

In this code, n_x bound variables are used with the universal quantifier `\forall` to represent all the vectors x_e belonging to Γ. Similarly, when formally verifying the output invariant property, the following postcondition is used:

ACSL
```
//Output Invariant Postcondition
ensures \forall real l1; ...;\forall real lny; -1 <= l1 <= 1
    ==> ... ==> -1 <= lny <= 1 ==> stateinv(\old(x->x1), ...,
    \old(x->xnx),1) ==> outputinv(y->y1+l1*e1, ..., y->yny+lny*
    eny,1);
```

Method 2: Checking Each Vertex. The second method to verify that $\Gamma \subset \mathcal{E}_X$ benefits from the convexity of the quadratic function $z^T X z$. Precisely, by leveraging the convexity of the quadratic function $z^T X z$, the following holds: $\Gamma \subset \mathcal{E}_X$ if and only if all the vertices $\hat{z}_i$, for $i = 1, \ldots, 2^n$, of Γ belong to $\mathcal{E}_X$. Therefore, to formally verify that $\mathrm{fl}(z) \in \mathcal{E}_X$, we must formally verify that the vertices of Γ belong to $\mathcal{E}_X$. The vertices $\hat{z}_i$, for $i = 1, \ldots, 2^n$, of Γ can be expressed in ACSL using the universal quantifier $\forall$ (`\forall`) and n bound variables ($\mathtt{l}_1, \ldots, \mathtt{l}_n$), each belonging to $\{-1, 1\}$. The postconditions for verifying the state and output invariant properties of the system using this method are similar to the ones used in the first method, with the exception that the bound variables' inequalities (`-1<=`$\mathtt{l}_i$`<=1`) are replaced by (`l`$_i$`==-1 || l`$_i$`==1`).

Assessment of Both Methods. While it is possible to formally verify that $\Gamma \subset \mathcal{E}_X$ using the methods described before, the use of quantifiers may lead to a proliferation of variables or constraints, which can make it difficult for the automated prover to discharge the proof: this may either lead to an extended time to prove the goals or to a solver failure. For instance, in our experiments, it was possible to verify the invariant properties of an LTI system with 16 state variables, 10 inputs, and 4 outputs in the float model using Method 1, but it was not possible to do so for any of the considered affine LPV systems with 4 state variables, 2 inputs, 2 outputs, and up to 2 scheduling parameters. On the other hand, using Method 2, it was possible to verify the invariant properties of these affine LPV systems and corresponding LTI systems in the float model, but it was not possible to do so for the large LTI system verified using Method 1. To address this issue, we present a different approach in the next section for formally verifying the invariant properties in the float model without the use of quantifiers.

5.4 Error Ball Approach

Consider the "error ball" $\mathcal{B}_e$ centered around the exact vector z with a radius r such that $\mathcal{B}_e$ covers the hyper-rectangle Γ. The ball $\mathcal{B}_e$ is defined as $\mathcal{B}_e = \{z_e \in \mathbb{R}^n | \, z_e = z + ru, \, \|u\|_2 \leq 1\}$, where $\|.\|_2$ is the standard Euclidean norm. Since $\mathrm{fl}(z) \in \Gamma$, it follows that $\mathrm{fl}(z) \in \mathcal{B}_e$ as well. Therefore, to verify that $\mathrm{fl}(z) \in \mathcal{E}_X$, it is sufficient to show that $\mathcal{B}_e \subset \mathcal{E}_X$. Clearly, $\mathcal{B}_e \subset \mathcal{E}_X$ if and only if all the points belonging to $\mathcal{B}_e$ also belong to $\mathcal{E}_X$. In other words, $\mathcal{B}_e \subset \mathcal{E}_X$ if and only if, for all $u \in \mathbb{R}^n$ such that $\|u\|_2 \leq 1$,

$$z_e^T X z_e = (z + ru)^T X (z + ru) = z^T X z + 2ru^T X z + r^2 u^T X u \leq 1.$$

It is not difficult to prove that the following inequality holds [7]:

$$z_e^T X z_e \leq z^T X z + 2r\|X\|_2 \|z\|_2 + r^2 \|X\|_2, \tag{7}$$

where $\|X\|_2$ is the matrix 2-norm induced by the vector Euclidean norm, i.e., $\|X\|_2 = \lambda_{\max}(X)$, where $\lambda_{\max}(X)$ is the maximum eigenvalue of X. We recall

that, by assumption, it is formally verified in the real model that the exact vector z belongs to $\mathcal{E}_X$. Based on this assumption, we can find the maximum 2-norm of z such that $z \in \mathcal{E}_X$ by solving the following nonconvex optimization problem:

$$\begin{aligned} &\text{maximize} \quad z^T z \\ &\text{subject to} \quad z^T X z \leq 1. \end{aligned} \tag{8}$$

This optimization problem is a special case of a nonconvex problem discussed in [7, Chapter 5.2.4], for which strong duality holds [25, 7], i.e., the optimal value, $\|z^*\|_2^2$, of the primal problem is equal to the optimal value of the following dual problem:

$$\begin{aligned} &\text{minimize} \quad \alpha \\ &\text{subject to} \quad X^{-1} \preceq \alpha I. \end{aligned} \tag{9}$$

The optimal value of the dual problem is $\alpha^* = \lambda_{\max}(X^{-1}) = \frac{1}{\lambda_{\min}(X)}$, where X^{-1} is the inverse of X and $\lambda_{\min}(X)$ is the minimum eigenvalue of X. Accordingly, the optimal value of the primal nonconvex problem is $\|z^*\|_2^2 = \alpha^* = (\lambda_{\min}(X))^{-1}$. Then, for all $z \in \mathcal{E}_X$, the following inequalities hold:

$$\begin{aligned} z_e^T X z_e &\leq z^T X z + 2r\|X\|_2\|z\|_2 + r^2\|X\|_2 \\ &\leq z^T X z + r\lambda_{\max}(X)\left(2\left(\lambda_{\min}(X)\right)^{-\frac{1}{2}} + r\right). \end{aligned} \tag{10}$$

Therefore, it is sufficient to formally verify that

$$z^T X z \leq 1 - r\lambda_{\max}(X)\left(2\left(\lambda_{\min}(X)\right)^{-\frac{1}{2}} + r\right) \tag{11}$$

to conclude that $z_e^T X z_e \leq 1$ for all $z_e \in \mathcal{B}_e$, and that $\mathcal{B}_e \subset \mathcal{E}_X$. In other words, if (11) is formally verified, then $\mathrm{fl}(z) \in \mathcal{E}_X$ and the ellipsoidal invariant property is verified in the float model. To formally verify (11), we need to compute the radius r of $\mathcal{B}_e$ such that $\Gamma \subset \mathcal{B}_e$, as well as the maximum and minimum eigenvalues of X. Since Γ is a symmetric hyper-rectangle about z, the smallest radius of $\mathcal{B}_e$ such that $\mathcal{B}_e$ covers Γ is $r = \|e\|_2$ [7], where e is the error vector satisfying (6). Then, for any $r \geq \|e\|_2$, $\mathcal{B}_e$ covers Γ. One acceptable choice of r is $r = n\|e\|_\infty$, where $\|e\|_\infty = \max_{i=1,\ldots,n} |e_i|$ is the ∞-norm of e. This choice is valid because $\|e\|_2 \leq \sqrt{n}\,\|e\|_\infty$. For our analysis, it is a better choice to set $r = n\|e\|_\infty$, as this computation only requires one operation compared to the $2n$ operations required for computing $\|e\|_2$, which minimizes the accumulated floating-point error during the computation of r. The error vector e is computed outside of Frama-C and injected in the contract. As for the computation of the maximum and minimum eigenvalues of X, there are several algorithms that can be used to compute the eigenvalues of a matrix, such as the diagonalization, power iteration, and QR algorithms, and singular value decomposition (SVD) methods [14, 31]. These algorithms are generally reliable and can be expected to produce accurate results in most cases. For instance, iterative methods like the power iteration algorithm can be employed to compute the eigenvalues of a matrix, starting with

a random initial vector [19]. This approach allows for an over-approximation of the converged value by estimating it from above. The over-approximation is then fed back into the algorithm as input for subsequent iterations, which refines the approximation and helps ensure its validity.

Hence, to formally verify the state and output invariant properties of the system G in the float model, we add `float_model` contracts to the code in Section 4 as follows:

C+ACSL

```
/*@ behavior contract_name_float_model:
      assumes ...;
      ensures stateinv(\old(x->x_1), ..., \old(x->x_{n_x}),1) ==>
          outputinv(y->y_1, ...,y->y_{n_y},1 - 2 * r_y * norm_Q *
          norm_y_max - r_y * r_y * norm_Q);*/
void updateOutput(...) {...}
/*@ behavior contract_name_float_model:
      assumes ...;
      ensures stateinv(\old(x->x_1), ..., \old(x->x_{n_x}),1) ==>
          stateinv(x->x_1, ...,x->x_{n_x},1 - 2 * r_x * norm_P *
          norm_x_max - r_x * r_x * norm_P));*/
void updateState(...) {...}
```

In this code, the implied expressions in the postconditions correspond to inequality (11). The terms `norm_P`, `norm_Q`, `r_x`, `r_y`, `norm_x_max`, and `norm_y_max` correspond to $\|P\|_2$, $\|Q\|_2$, the radii of the error balls centered around the updated state and output vectors x and y, $(\lambda_{\min}(P))^{-\frac{1}{2}}$, and $(\lambda_{\min}(Q))^{-\frac{1}{2}}$, respectively. This approach allows for the formal verification of the invariant properties of all considered affine LPV and LTI systems in the float model.

6 Conclusion

This paper demonstrates a process for formally verifying the invariant properties of a C code describing the dynamics of a discrete-time LPV system with affine parameter dependence. The ACSL language and the WP plugin in Frama-C are used to express the invariant properties and generate proof obligations, and the polynomial inequalities plugin in Alt-Ergo is used discharge these proof obligations. The invariant properties were successfully verified in both the real and float models, with the latter requiring the use of bounds on numerical errors and their incorporation into the real model. This process can be applied to other systems with similar properties. The installation instructions of the tools used in this work along with the experiments are available at `https://github.com/ploc/verif-iqc`. Additionally, a dockerfile is also available at `https://hub.docker.com/r/ekhalife/verif-iqc`, and the instructions for using the dockerfile can be found in the same GitHub repository. In future work, we plan to extend this approach to more general classes of uncertain systems.

Bibliography

[1] Apkarian, P., Tuan, H.D.: Parameterized LMIs in control theory. SIAM Journal on Control and Optimization **38**(4), 1241–1264 (2000)

[2] Barrett, C.W., Conway, C.L., Deters, M., Hadarean, L., Jovanovic, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification - 23rd International Conference, CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proceedings. Lecture Notes in Computer Science, vol. 6806, pp. 171–177. Springer (2011). https://doi.org/10.1007/978-3-642-22110-1_14

[3] Baudin, P., Cuoq, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language ACSL version 1.18, implementation in FRAMA-C 26.0

[4] Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI C specification language. CEA-LIST, Saclay, France, Tech. Rep. v1 **2** (2008)

[5] Blanchini, F.: Set invariance in control. Automatica **35**(11), 1747–1767 (1999)

[6] Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear matrix inequalities in system and control theory. SIAM (1994)

[7] Boyd, S., Vandenberghe, L.: Convex optimization. Cambridge university press (2004), `http://www.stanford.edu/%7Eboyd/cvxbook/`

[8] Conchon, S., Iguernelala, M., Mebsout, A.: A collaborative framework for non-linear integer arithmetic reasoning in alt-ergo. In: Bjørner, N.S., Negru, V., Ida, T., Jebelean, T., Petcu, D., Watt, S.M., Zaharie, D. (eds.) 15th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2013, Timisoara, Romania, September 23-26, 2013. pp. 161–168. IEEE Computer Society (2013). https://doi.org/10.1109/SYNASC.2013.29

[9] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., Yakobowski, B.: Frama-C. In: Eleftherakis, G., Hinchey, M., Holcombe, M. (eds.) Software Engineering and Formal Methods. pp. 233–247. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)

[10] Darulova, E., Kuncak, V.: Sound compilation of reals. SIGPLAN Not. **49**(1), 235–248 (jan 2014). https://doi.org/10.1145/2578855.2535874

[11] Daumas, M., Melquiond, G.: Certification of bounds on expressions involving rounded operators. ACM Trans. Math. Softw. **37**(1) (jan 2010). https://doi.org/10.1145/1644001.1644003

[12] De Moura, L., Bjørner, N.: Z3: An efficient SMT solver. In: Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems. p. 337–340. TACAS'08/ETAPS'08, Springer-Verlag, Berlin, Heidelberg (2008)

[13] Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., Védrine, F.: Towards an industrial use of FLUCTUAT on safety-critical avionics

software. In: Alpuente, M., Cook, B., Joubert, C. (eds.) Formal Methods for Industrial Critical Systems, 14th International Workshop, FMICS 2009, Eindhoven, The Netherlands, November 2-3, 2009. Proceedings. Lecture Notes in Computer Science, vol. 5825, pp. 53–69. Springer (2009). https://doi.org/10.1007/978-3-642-04570-7_6

[14] Golub, G.H., Van Loan, C.F.: Matrix computations. JHU press (2013)

[15] Goubault, E.: Static analyses of the precision of floating-point operations. In: Proceedings of the 8th International Symposium on Static Analysis. pp. 234–259. SAS '01, Springer-Verlag, London, UK, UK (2001)

[16] Goubault, E., Putot, S.: Static analysis of finite precision computations. In: VMCAI. pp. 232–247 (2011)

[17] Khalife, E., Abou Jaoude, D., Farhood, M., Garoche, P.L.: Computation of invariant sets for discrete-time uncertain systems. Submitted (2022)

[18] Khalil, H.K.: Nonlinear systems; 3rd ed. Prentice-Hall, Upper Saddle River, NJ (2002), `https://cds.cern.ch/record/1173048`

[19] Kuczyński, J., Woźniakowski, H.: Estimating the largest eigenvalue by the power and Lanczos algorithms with a random start. SIAM Journal on Matrix Analysis and Applications **13**(4), 1094–1122 (Oct 1992). https://doi.org/10.1137/0613066

[20] Kurzhanski, A.B., Varaiya, P.: Ellipsoidal techniques for reachability analysis. In: International workshop on hybrid systems: Computation and control. pp. 202–214. Springer (2000)

[21] Martel, M.: An overview of semantics for the validation of numerical programs. In: Verification, Model Checking, and Abstract Interpretation, 6th International Conference, VMCAI 2005, Paris, France, January 17-19, 2005, Proceedings. pp. 59–77 (2005). https://doi.org/10.1007/978-3-540-30579-8_4

[22] Martin-Dorel, É., Roux, P.: A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In: Bertot, Y., Vafeiadis, V. (eds.) Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017. pp. 90–99. ACM (2017). https://doi.org/10.1145/3018610.3018622

[23] Megretski, A., Rantzer, A.: System analysis via integral quadratic constraints. IEEE Transactions on Automatic Control **42**(6), 819–830 (1997)

[24] Muniraj, D., Palframan, M.C., Guthrie, K.T., Farhood, M.: Path-following control of small fixed-wing unmanned aircraft systems with $\mathcal{H}_\infty$ type performance. Control Engineering Practice **67**, 76–91 (2017)

[25] Nocedal, J., Wright, S.J.: Numerical optimization. Springer (1999)

[26] Peled, D.A.: Software reliability methods. Springer Science & Business Media (2013)

[27] Roux, P.: Formal proofs of rounding error bounds - with application to an automatic positive definiteness check. J. Autom. Reason. **57**(2), 135–156 (2016). https://doi.org/10.1007/s10817-015-9339-z

[28] Roux, P., Iguernlala, M., Conchon, S.: A non-linear arithmetic procedure for control-command software verification. In: Beyer, D., Huisman, M.

(eds.) Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II. Lecture Notes in Computer Science, vol. 10806, pp. 132–151. Springer (2018). https://doi.org/10.1007/978-3-319-89963-3_8

[29] Solovyev, A., Baranowski, M., Briggs, I., Jacobsen, C., Rakamaric, Z., Gopalakrishnan, G.: Rigorous estimation of floating-point round-off errors with symbolic taylor expansions. ACM Transactions on Programming Languages and Systems (TOPLAS) **41**(1), 2:1–2:39 (dec 2018). https://doi.org/10.1145/3230733

[30] Topcu, U., Packard, A., Seiler, P.: Local stability analysis using simulations and sum-of-squares programming. Automatica **44**(10), 2669–2675 (2008). https://doi.org/https://doi.org/10.1016/j.automatica.2008.03.010

[31] Trefethen, L.N., Bau III, D.: Numerical linear algebra, vol. 50. SIAM (1997)