Formally Proving Invariant Systemic Properties of Control Programs Using Ghost Code and Integral Quadratic Constraints* Appendix

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 formally verifying invariant properties of control programs both at the model and code levels. The physical process is described by an uncertain discrete-time state-space system, where the dependence of the state-space matrix-valued functions defining the system on the uncertainties can be rational. The proposed approaches make use of pointwise integral quadratic constraints (IQCs) to characterize the uncertainties affecting the behavior of the system. Various uncertainties can be characterized by pointwise IQCs, including static linear time-varying perturbations and sector-bounded nonlinearities. Using the IQC framework, a sound overapproximation of the uncertain system, which is expressible at the code level, is constructed. Tools such as Frama-C, ACSL, WP, and an Alt-Ergo plugin are employed to ensure the validity of the state and output invariant properties across both real and float models. The first proposed approach can be used to formally verify (local) invariant properties of the control code. This capability is demonstrated in a couple of examples involving gain-scheduled path-following controllers designed for an uncrewed aircraft system and an autonomous underwater vehicle. The second approach enables the verification of closed-loop invariant properties, i.e., invariant properties of the controlled system as a whole, in both real and float models, while preserving the integrity of the executable controller code. This is achieved by using ghost code attached to the control code for all elements related to the plant model with uncertainties, as the ghost code does not interfere with the executable code. The effectiveness of this approach is demonstrated in an example on the control of a four-thruster hovercraft.

Keywords: Deductive Verification, Invariant Set, Ghost Code, Uncertain System, IQC Framework, Float Model.

* This material is based upon work supported by the Army Research Office (ARO) under Contract No. W911NF-21-1-0250.

A LFT Framework

Discrete-time uncertain systems are commonly encountered in the field of control theory. They can be derived using system identification techniques or by discretizing continuous-time uncertain systems whose equations of motion are defined by ordinary differential equations. Various discretization methods can be utilized, such as the Euler method and the trapezoidal method [3]. Errors resulting from these discretization methods can be incorporated into the system as additional uncertainties. In this work, we focus on uncertainty operators that admit pointwise IQC characterizations. Henceforth, for simplicity, the uncertainty operators are assumed to be linear memoryless operators.

Consider the uncertain system G_δ defined by the state-space equations

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

where k denotes discrete time and x, y, d denote the state, output, and input signals, respectively. The system G_δ can represent a plant model or a controller, such as a gain-scheduled controller. In the context of feedback controllers, d and y correspond to the feedback signal and the control signal, respectively.

Assuming at most rational dependence on the uncertainties, the system G_δ can be equivalently expressed in an LFT form, i.e., as the interconnection of an LTI nominal model G and a structured uncertainty operator Δ, as depicted in Fig. **??**. This LFT system, (G, Δ), is described by the state-space equations

$$\begin{aligned} x_G(k+1) &= A_G x_G(k) + B_{G_1}\vartheta(k) + B_{G_2}d(k), \\ \varphi(k) &= C_{G_1}x_G(k) + D_{G_{11}}\vartheta(k) + D_{G_{12}}d(k), \\ y(k) &= C_{G_2}x_G(k) + D_{G_{21}}\vartheta(k) + D_{G_{22}}d(k), \\ \vartheta &= \Delta(\varphi), \end{aligned}$$

where $x_G(k) \in \mathbb{R}^{n_G}$, $d(k) \in \mathbb{R}^{n_d}$, $y(k) \in \mathbb{R}^{n_y}$, $\varphi(k) \in \mathbb{R}^{n_\varphi}$, $\vartheta(k) \in \mathbb{R}^{n_\vartheta}$, and $\Delta : \ell_2^{n_\varphi} \to \ell_2^{n_\vartheta}$. In general, the operator Δ can be structured and expressed as $\Delta = \mathrm{blkdiag}(\Delta_1, \Delta_2, \ldots, \Delta_N)$, for some positive integer N, where the diagonal blocks Δ_i are operators corresponding to different types of uncertainties, i.e., $\Delta_i \in \mathbf{\Delta_i}$ for $i = 1, \ldots, N$. The signals φ and ϑ reflect the impact of the uncertainty Δ on the system dynamics. As the uncertainty operators are assumed to be linear and memoryless, the state-space matrices of G_δ and (G, Δ) are related as follows:

$$\begin{aligned} A(\delta) &= A_G + B_{G_1}\Delta(I - D_{G_{11}}\Delta)^{-1}C_{G_1}, \\ B(\delta) &= B_{G_2} + B_{G_1}\Delta(I - D_{G_{11}}\Delta)^{-1}D_{G_{12}}, \\ C(\delta) &= C_{G_2} + D_{G_{21}}\Delta(I - D_{G_{11}}\Delta)^{-1}C_{G_1}, \\ D(\delta) &= D_{G_{22}} + D_{G_{21}}\Delta(I - D_{G_{11}}\Delta)^{-1}D_{G_{12}}. \end{aligned}$$

Deriving the LFT reformulation (G, Δ) of G_δ can be done manually or using some currently available toolboxes, such as the SMAC toolbox [2].

To illustrate how an uncertain system can be represented in LFT form, consider the following numerical example:

$$x(k+1) = \begin{bmatrix} a_1 & a_2 + a_3\delta(k) + a_4\delta(k)^2 \\ a_5 & a_6 + a_7\delta(k) \end{bmatrix} x(k) + \begin{bmatrix} b_1 \\ b_2 \end{bmatrix} d(k),$$

where a_i, for $i = 1, \ldots, 7$, and b_j, for $j = 1, 2$, are constant scalars, and $\delta(k)$ is an uncertain parameter such that $|\delta(k)| \leq 1$. The LFT form of this system is defined by the following state-space matrices:

$$A_G = \begin{bmatrix} a_1 & a_2 \\ a_5 & a_6 \end{bmatrix}, \qquad B_{G_1} = \begin{bmatrix} a_3 & a_4 \\ a_7 & 0 \end{bmatrix}, \qquad B_{G_2} = \begin{bmatrix} b_1 \\ b_2 \end{bmatrix},$$
$$C_{G_1} = \begin{bmatrix} 0 & 1 \\ 0 & 0 \end{bmatrix}, \qquad D_{G_{11}} = \begin{bmatrix} 0 & 0 \\ 1 & 0 \end{bmatrix}, \qquad D_{G_{12}} = \begin{bmatrix} 0 \\ 0 \end{bmatrix},$$

and $\Delta(k) = \delta(k) I_2$, i.e., $\Delta \in \mathbf{\Delta}_{\text{SLTV}}$ with $\alpha = 1$. The equivalence of these two systems can be easily verified.

B Templates for Verification of Invariant Properties

This appendix presents two templates for verifying the invariant properties of uncertain systems, modeled by the system H defined in (**??**) and coupled with the condition (**??**). First, a pseudocode template outlines the methodology for verifying state and output invariants. Then, an annotated Frama-C script with ACSL annotations is provided, demonstrating the practical application of this approach, complete with detailed formal specifications.

```
Define Structures:
  - state with elements x_1, ..., x_{n_H}.
  - output with elements y_1, ..., y_{n_y}.

Define Predicates:
  - state_ellip(x, λ): true if x^⊤ P x ≤ λ.
  - output_ellip(y, λ): true if y^⊤ Q y ≤ λ.
  - pwIQC_state(r): true if r^⊤ S_x r ≥ 0.
  - pwIQC_output(r): true if r^⊤ S_y r ≥ 0.

Function updateState(x_H(k), ϑ(k), d(k)):
  Update Logic:
    Compute new state values based on (??).
  Preconditions:
    * x_H(k) is valid and has no memory overlap with other
        ↪ structures.
    * d(k) ∈ 𝒟.
    * pwIQC_state(r(k)).
  Contract under real model assumptions:
    * Ensure state_ellip(x_H(k),1) ⇒ state_ellip(x_H(k+1),1).
  Contract under float model assumptions:
```

```
    * Ensure state_ellip(x_H(k),1) ⇒ state_ellip(x_H(k+1),α_{x_H}).

Function updateOutput(x_H(k), y(k), ϑ(k), d(k)):
  Update Logic:
    Compute new output values based on (??).
  Preconditions:
    * x_H(k) and y(k) are valid and have no memory overlap with
        ↪ other structures.
    * d(k) ∈ 𝒟.
    * pwIQC_output(r(k)).
  Contract under real model assumptions:
    * Ensure state_ellip(x_H(k),1) ⇒ output_ellip(y(k),1).
  Contract under float model assumptions:
    * Ensure state_ellip(x_H(k),1) ⇒ output_ellip(y(k),α_y).
```

Listing 1.1. Pseudo-Code for Verifying the State and Output Invariants of Uncertain Systems.

In ACSL, annotations are introduced using `//@` for single-line specifications or `/*@ ... */` for block specifications. Here is a brief explanation of key ACSL annotations used in the script:

- `requires`: Specifies preconditions that must hold true before the function executes.
- `ensures`: Specifies postconditions that must hold true after the function executes.
- `\valid`: Checks the validity of pointers, ensuring they point to allocated and accessible memory.
- `assigns`: Lists the variables or memory locations that the function may modify, constraining side effects to specified locations.
- `\separated`: Ensures specified pointers or memory regions do not overlap, preventing unintended side effects.
- `behavior`: Used to define specific behavior contracts for a function under certain conditions.
- `assumes`: Declares assumptions that are considered true within the scope of a behavior (or contract).
- `\old`: Refers to the value of a variable before the function's execution (used in postconditions).

Once the annotated C script is ready, we verify the state and output invariant properties using the WP plugin [1] and the Alt-Ergo-Poly solver [7]. This verification is performed by executing the following command:

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

The `-wp-model real` option directs the WP plugin to perform calculations in the real arithmetic model. Additional command options are available, such as `-wp-timeout` to set a solver timeout [1].

C+ACSL

```
typedef struct { double x1,... , x_nH; } state;

typedef struct { double y1,... , y_ny; } output;

//@ predicate state_ellip(real x1,... , real x_nH, real
    lambda) = P_11 * x1 * x1 + 2 * P_12 * x1 * x2 + ... +
    P_nHnH * x_nH * x_nH <= lambda;

//@ predicate output_ellip(real y1,... , real y_ny, real
    lambda) = Q_11 * y1 * y1 + 2 * Q_12 * y1 * y2 + ... +
    Q_nyny * y_ny * y_ny <= lambda;

//@ predicate pwIQC_state(real r1,... , real r_nr) = Sx_11
    * r1 * r1 + ... + Sx_nrnr * r_nr * r_nr >= 0;

//@ predicate pwIQC_output(real r1,... , real r_nr) = Sy_11
     * r1 * r1 + ... + Sy_nrnr * r_nr * r_nr >= 0;

/*@
requires \valid(x);

requires \separated(&(x->x1),... ,&(x->x_nH));

// Replace r1, r2,..., r_nr with their expressions in terms
     of x, theta, and d based on the state-space equations
    of the augmented system H.
requires pwIQC_state(r1,... , r_nr);

assigns *x;

behavior polytope_input_real_model:
   assumes -d1_bar <= d1 <= d1_bar;
   .
   .
   .
   assumes -d_nd_bar <= d_nd <= d_nd_bar;

   ensures state_ellip(\old(x->x1),... , \old(x->x_nH), 1)
       ==> state_ellip(x->x1,... , x->x_nH, 1);

behavior polytope_input_float_model:
   assumes -d1_bar <= d1 <= d1_bar;
   .
   .
   .
   assumes -d_nd_bar <= d_nd <= d_nd_bar;
```

```
// Replace alpha_x with its expression in Section 3.2
   ensures state_ellip(\old(x->x1),... , \old(x->x_nH), 1)
      ==> state_ellip(x->x1,... , x->x_nH, alpha_x);
*/

void updateState(state *x, double theta1,... , double
    theta_ntheta, double d1,... , double d_nd) {
   double pre_x1 = x->x1,... , pre_x_nH = x->x_nH;

// Update the state vector based on the state-space
    equations of the augmented system H.
   x->x1 = ...;
   .
   .
   .
   x->x_nH = ...;
}

/*@
requires \valid(x);
requires \valid(y);

requires \separated(&(x->x1),... ,&(x->x_nH),&(y->y1),...
    ,&(y->y_ny));

// Replace r1, r2,..., r_nr with their expressions in terms
     of x, theta, and d based on the state-space equations
    of the augmented system H.
requires pwIQC_output(r1,... , r_nr);

assigns *y;

behavior polytope_input_real_model:
   assumes -d1_bar <= d1 <= d1_bar;
   .
   .
   .
   assumes -d_nd_bar <= d_nd <= d_nd_bar;

   ensures state_ellip(\old(x->x1),... , \old(x->x_nH), 1)
      ==> output_ellip(y->y1,... , y->y_ny, 1);

behavior polytope_input_float_model:
   assumes -d1_bar <= d1 <= d1_bar;
   .
   .
   .
   assumes -d_nd_bar <= d_nd <= d_nd_bar;
```

```
// Replace alpha_y with its expression in Section 3.2
   ensures state_ellip(\old(x->x1),... , \old(x->x_nH), 1)
      ==> output_ellip(y->y1,... , y->y_ny, alpha_y);
*/

void updateOutput(state *x, output *y, double theta1,... ,
    double theta_ntheta, double d1,... , double d_nd) {
   double pre_x1 = x->x1,... , pre_x_nH = x->x_nH;

// Compute the output vector based on the state-space
    equations of the augmented system H.
   y->y1 = ...;
   .
   .
   .
   y->y_ny = ...;
}
```

C Templates for Verification with Ghost Code

This appendix provides a pseudocode template and a corresponding Frama-C script with ACSL annotations for verifying systemic properties of control programs using ghost code. The template illustrates how ghost variables and functions model the dynamics of the plant, enabling verification without modifying the executable code.

```
Define Structures:
  - Controller State with elements x_1,...,x_{n_c}.
  - Control Input with elements u_1,...,u_{n_u}.
  - Output Measurements with elements y_1,...,y_{n_y}.

Define Predicates:
  - state_ellip(x, λ): true if x^⊤ P x ≤ λ.
  - pwIQC_state(r): true if r^⊤ S_x r ≥ 0.

Define ghost code and variables:
  - ghost x_H.
  - ghost ϑ.
  - ghost d.
  - ghost Function plantDynamics(x_H(k),ϑ(k),d(k),u(k)) that
      ↪ computes x_H(k+1) based on the dynamics of H.

Function controller(x_c(k), u(k), y(k)):
  Update Logic:
    Compute new controller state and input values based on
       ↪ (??)
  Preconditions:
```

```
    * x_c(k) and u(k) are valid and have no memory overlap with
        ↪ other structures.
    * d(k) ∈ 𝒟.
    * pwIQC_state(r(k)).
    * y(k) satisfies the output equation of H
  Contract under real model assumptions:
    * Ensure state_ellip((x_H(k), x_c(k)),1) ⇒
             state_ellip((x_H(k+1), x_c(k+1)),1),
             where x_H(k+1) ← plantDynamics(x_H(k),ϑ(k),d(k),u(k))
  Contract under float model assumptions:
    * Ensure ∀ i ∈ {1,...,n_u}, −1 ≤ l_i ≤ 1,
             state_ellip((x_H(k), x_c(k)),1) ⇒
             state_ellip((x_H(k+1), x_c(k+1)),α),
             where x_H(k+1) ← plantDynamics(x_H(k),ϑ(k),d(k),
                 ↪ u(k) + l · e_u), with l = [l_1,...,l_{n_u}]^⊤.
```

Listing 1.2. Pseudo-code Template for Verifying Systemic Properties of a Control Program Using Ghost Code Annotations

In addition to the ACSL annotations defined in Appendix B, the annotated script in this appendix uses the following annotations:

- `ghost`: Declares ghost variables or functions that do not affect the actual execution of the program but are used for verification purposes.
- `\let`: Defines temporary variables that hold specific values for use in an expression.
- `logic`: Defines a logic function, which is a function in ACSL used for calculations within annotations.
- `\at`: Allows referencing the value of a variable at a specific program point.

The same command as in Appendix B is executed to verify the systemic properties of the control program.

C+ACSL

```
/* First, we define
   - the state_ellip predicate describing the state
       invariant ellipsoid
   - one logic function for each plant state that
       corresponds to the computation of that state.
    (The logic functions are not mandatory but will
       facilitate writing contracts.)
*/

//@ predicate state_ellip(real x1,..., real x_ncl, real
    lambda) = ...;

//@ predicate pwIQC_state(real r1,... , real r_nr) = Sx_11
    * r1 * r1 + ... + Sx_nrnr * r_nr * r_nr >= 0;
```

```
// Logic Functions to compute the augmented states based on
     x(k+1) = A_H*x(k) + B_H1*theta(k) + B_H2*d(k) + B_H3*u
    (k)
// Computing x1
/*@
  logic real update_x1 (real pre_x1, ..., real pre_x_nH,
      real d1, ..., real d_nd, real theta1, ..., real
      theta_ntheta, real u1, ..., real u_nu) =
  (Equation)
 */

// Computing x_nH
/*@
  logic real update_x_nH (real pre_x1, ..., real pre_x_nH,
      real d1, ..., real d_nd, real theta1, ..., real
      theta_ntheta, real u1, ..., real u_nu) =
  (Equation)
 */

typedef struct { double x1, ..., x_nc; } state;

typedef struct {double u1, ..., u_nu;} control_input;

/*@
    requires \valid(xc) && \valid(u);
    requires -d1_bar <= d1 <= d1_bar;
    .
    .
    .
    requires -d_nd_bar <= d_nd <= d_nd_bar;
    requires \separated(&(xc->x1),...,&(xc->x_nc),&(u->u1),
        ...,&(u->u_nu));
    requires state_ellip(x1, ..., x_nH, xc->x1, ...,
        xc->x_nc, 1);

    // Replace r1, r2,..., r_nr with their expressions in
        terms of x_cl, theta, d, and u based on the
        state-space equations of the augmented system H.
    requires pwIQC_state(r1,... , r_nr);

    // Require that the output vector satisfies the output
        equations
    requires y1 == ...;
    .
    .
    .
    requires y_ny == ...;
    assigns *xc, *u;
```

```
    // Real model:
    ensures \let nx1 = update_x1 (\at(x1, Pre), ..., \at(
        x_nH, Pre), d1, ..., d_nd, theta1, ...,
        theta_ntheta, u->u1, ..., u->u_nu);
      .
      .
      .
        \let nx_nH = update_x_nH (\at(x1, Pre), ..., \at(
            x_nH, Pre), d1, ..., d_nd, theta1, ...,
            theta_ntheta, u->u1, ..., u->u_nu);
        state_ellip(nx1, ..., nx_nH, xc->x1, ..., xc->x_nc,
             1);

  // Float model (replace alpha with its expression in
      Section 4.2):
    ensures \forall real l_1; ...; \forall real l_nu;
        \let nx1 = update_x1 (\at(x1, Pre), ..., \at(x_nH,
            Pre), d1, ..., d_nd, theta1, ..., theta_ntheta,
             u->u1 + l_1 * e_u, ..., u->u_nu + l_nu * e_u);
      .
      .
      .
        \let nx_nH = update_x_nH (\at(x1, Pre), ..., \at(
            x_nH, Pre), d1, ..., d_nd, theta1, ...,
            theta_ntheta, u->u1 + l_1 * e_u, ..., u->u_nu +
             l_nu * e_u);
        -1 <= l_1 <= 1 ==> ... ==> -1 <= l_nu <= 1 ==>
        state_ellip(nx1, ..., nx_nH, xc->x1, ..., xc->x_nc,
             alpha);
*/
void controller(state *xc, control_input *u, double y1, ...
    , double y_ny) /*@ ghost (double x1, ..., double x_nH,
    double theta1, ..., double theta_ntheta, double d1, ...
    , double d_nd) */ {
    double pre_xc1 = xc->x1, ..., pre_xc_nc = xc->x_nc;

    // Compute control inputs
    u->u1 = ...;
    .
    .
    .
    u->u_nu = ...;

    // Update controller states
    xc->x1 = ...;
    .
    .
```

```
    .
    xc->x_nc = ...;
}
```

D Verification of Invariant Systemic Properties of Affine LPV Controllers

Consider an output feedback discrete-time affine LPV controller $\tilde{K}$ as defined in (**??**), where the state-space matrices $A_c(\delta(k))$, $B_c(\delta(k))$, $C_c(\delta(k))$, and $D_c(\delta(k))$ have an affine dependence on the scheduling parameters $\delta(k) = (\delta_1(k), \ldots, \delta_{n_\delta})$. While it is feasible to apply the verification approach used for the LTI controller in Section **??** directly to an affine LPV controller, difficulties may arise in successfully discharging proofs of systemic properties, particularly when the state-space equations of the controller involve multiple terms dependent on scheduling parameters. To address this, our approach involves modeling the affine LPV controller as an augmented system $\tilde{H}$. We then formally verify that $\tilde{H}$ is a sound representation of $\tilde{K}$ and that the systemic properties for the closed-loop system comprising $\tilde{H}$ and the plant model H hold. Since $\tilde{H}$ is shown to be a sound model of $\tilde{K}$, we can consequently infer that the systemic properties verified for the augmented system also hold for the executable affine LPV controller code.

Affine LPV Controller Model Verification Assuming one scheduling parameter for simplicity, i.e., $n_\delta = 1$, the affine LPV controller $\tilde{K}$ can be equivalently expressed in LFT form with uncertainty set $\mathbf{\Delta}_{\text{SLTV}}$, which satisfies the pointwise IQC defined by $(\Psi_{\text{SLTV}}, S_{\text{SLTV}})$, as shown in Section **??**. To distinguish the φ, ϑ, and r variables of the controller from those of the plant model, we denote the controller variables with a tilde. Accordingly, we can construct an augmented system $\tilde{H}$, which, along with the pointwise IQC condition $\tilde{r}(k)^\top S_{\text{SLTV}} \tilde{r}(k) \geq 0$, models the affine LPV controller. However, we need to formally verify that $\tilde{H}$ is a sound model of the executable controller code. Specifically, this verification requires showing that for any given current state $x_c(k)$, output measurement $y(k)$, and scheduling parameter $\delta(k)$, it is possible to construct $\tilde{\varphi}(k)$ and $\tilde{\vartheta}(k)$ such that the updated state and control input of $\tilde{H}$ match those of the executable controller code, while also satisfying the IQC condition.

To this end, we verify at the code level that, at any time instant k,

$$\forall x_c(k) \in \mathbb{R}^{n_c},\, y(k) \in \mathbb{R}^{n_y},\, |\delta(k)| \leq \alpha,\, \exists\, \tilde{\varphi}(k) \in \mathbb{R}^{n_{\tilde{\varphi}}} \text{ and } \tilde{\vartheta}(k) = \delta(k)\tilde{\varphi}(k)$$
$$\text{such that } x_{\tilde{H}}(k+1) = x_c(k+1),\, u_{\tilde{H}}(k) = u(k), \text{ and } \tilde{r}(k)^\top S_{\text{SLTV}} \tilde{r}(k) \geq 0.$$

In essence, this verification confirms that the reachable states and control inputs of $\tilde{K}$ are contained within those of $\tilde{H}$, and that the constructed variables $\tilde{\varphi}(k)$ and $\tilde{\vartheta}(k)$ satisfy the pointwise IQC condition. In the case of LPV systems, as shown in Section **??**, the pointwise IQC condition takes the form $\tilde{r}(k)^\top S_{\text{SLTV}} \tilde{r}(k) = (\alpha^2 - \delta(k)^2)\tilde{\varphi}(k)^\top X \tilde{\varphi}(k) \geq 0$ for a given matrix $X \succeq 0$. Consequently, the verification of this property can be implied by verifying the more

general property $(\alpha^2 - \delta(k)^2) z^\top X z \geq 0$ for all $z \in \mathbb{R}^{n_{\tilde{\varphi}}}$ and any $X \succeq 0$ since the considered values of z would include those of the constructed $\tilde{\varphi}(k)$.

Remark 1. Verifying that an $n \times n$ symmetric matrix X is positive semidefinite can be done in several ways. For instance, it can be formally proven by verifying that $z^\top X z \geq 0$ holds for all vectors $z \in \mathbb{R}^n$ using a tool like KeYmaera X [6]. Alternatively, the verification can be empirically carried out by computing the eigenvalues of X and ensuring that, within an acceptable margin, these eigenvalues are nonnegative. Once verified, this positive semidefiniteness property can then be passed as an axiom to tools like Frama-C, allowing them to assume it as a given fact for further analysis.

Verification of Systemic Properties in the Float Model Upon closing the loop between H and $\tilde{H}$, the closed-loop system's state $x_{\text{cl}} = (x_H, x_{\tilde{H}})$ is formed. Applying the results from Section **??**, we determine the state-invariant and output-bounding ellipsoids, $\mathcal{E}_P$ and $\mathcal{E}_Q$, respectively. We then follow similar steps to those presented in Section **??**, with the key difference being that H is modeled using ghost code while $\tilde{H}$ is implemented as executable code. Since we have formally established that $\tilde{H}$ is a sound model for the executable controller code in the real model, verifying the systemic property for this closed-loop system implies that it holds for the controller's executable code as well. Now, we extend this verification to the float model, as the executable code operates with finite precision in practice.

To carry out this task, we compute the floating-point errors, $e_{x_{\tilde{H}}}$ and $e_{u_{\tilde{H}}}$, associated with the states and control input of $\tilde{H}$, respectively, using bounds obtainable from projecting $\mathcal{E}_P$ and $\mathcal{E}_Q$ onto appropriate planes. Similarly, we use these projected ellipsoid bounds to calculate the floating-point errors for the states and control input of the executable controller code, denoted as $e_{x_{\text{exe}}}$ and $e_{u_{\text{exe}}}$. To ensure that the augmented system $\tilde{H}$ captures the floating-point behavior of the executable controller code, we define the effective error bounds as $e_{x_c} = \max(e_{x_{\tilde{H}}}, e_{x_{\text{exe}}})$ and $e_u = \max(e_{u_{\tilde{H}}}, e_{u_{\text{exe}}})$. Finally, we modify the post-condition used in Section **??** by incorporating these combined error bounds, resulting in the adjusted ellipsoid $\mathcal{E}_{\tilde{P}}$ as defined in Section **??**. Here, e_{x_c} and e_u are defined as maximum error bounds to ensure that the verification accounts for the largest possible floating-point deviations across both the executable code and the augmented system. Verifying the modified systemic property in the real model thus ensures the validity of the original system property for the executable code in the float model as well.

Example: Two-Mass Rotational System

We consider the two-mass rotational system from [5] consisting of two bodies with moments of inertia $J_1 = 1$ and $J_2 = 0.1$, a spring, and a damper. The system comprises $n_G = 4$ state variables, which are the angular displacements and velocities of the two bodies, θ_1, θ_2, $\dot{\theta}_1$, and $\dot{\theta}_2$. The system has $n_u = 1$ control

input and $n_d = 1$ exogenous input d, each corresponding to a torque applied to body 1. The spring's stiffness coefficient is uncertain and time-varying but measurable at each moment in time, specifically, $k_s(\delta(k)) = 0.075 + 0.025\delta(k)$, where $|\delta(k)| \leq 1$ for all $k \geq 0$, and the damping coefficient is $b = 0.004$. The equations of motion, given in [5], are discretized via zero-order hold sampling, with a sampling time of $T = 0.1\,\mathrm{s}$. The resulting discrete-time system is an LPV system with affine dependence on the scheduling parameter δ. Choosing the performance and measurement outputs to be $z(k) = (\theta_2(k), u(k))$ and $y(k) = \theta_2(k)$, respectively, we design an affine LPV controller K for the system following the approach developed in [4], where the matrix Γ, specifying the uncertain initial state, is chosen as $\Gamma = \begin{bmatrix} 0.05 & 0 & 0.05 & 0 \end{bmatrix}^\top$. The affine LPV controller obtained has $n_c = 4$ state variables. The controller is expressed in LFT form, and following model reduction, the number of scheduling parameter copies is $n_{\tilde{\varphi}} = n_{\tilde{\vartheta}} = 5$. Uncertainty operators in both the plant and controller LFT systems are defined similarly to those in the UAS controller. The augmented systems H and $\tilde{H}$ are constructed, with $n_\vartheta = 1$ and $n_{\tilde{\vartheta}} = 5$. Next, we verify that $\tilde{H}$ is a sound model of the executable code of the affine LPV controller K by following the steps discussed earlier in this appendix. Running the verification command, we obtain the following results:

```
[wp] 8 goals scheduled
[wp] Proved goals:    8 / 8
  Qed:             5  (2ms-8ms-27ms)
  Alt-Ergo :       1  (74ms) (313)
  Z3 4.8.7:        2  (20ms-60ms) (186217)
```

In these results, one of the proved goals, discharged by Z3, corresponds to the state and control input reachability condition, while the goal discharged by Alt-Ergo pertains to the general condition that implies the pointwise IQC condition. In the verification annotations, an axiom asserting that a 5×5 matrix X is positive semidefinite, supporting the verification of the pointwise IQC condition, is used. Since these goals have been successfully verified, we affirm that $\tilde{H}$ is a sound model of the executable controller code. We then close the loop between H and $\tilde{H}$ and compute the state-invariant ellipsoid $\mathcal{E}_P$ along with S_x, using results from Section **??**. Following the steps in Section **??** and using the template in Appendix C, we write the annotated C script where H is expressed using ghost code to model plant dynamics, while $\tilde{H}$, as a sound model of K, is implemented as executable code. Running the verification command yields the following results:

```
[wp] 7 goals scheduled
[wp] Proved goals:    7 / 7
  Qed:               5  (3ms-9ms-26ms)
  Alt-Ergo-Poly :    2  (559ms) (1303)
```

These results confirm that the systemic property of this control program is verified in real and float models, using Alt-Ergo-Poly. Hence, we conclude that the systemic property holds for the executable code of the affine LPV controller K.

Bibliography

[1] Baudin, P., Bobot, F., Correnson, L., Dargaye, Z., Blanchard, A.: WP plug-in manual. Frama-c. com (2020)

[2] Biannic, J.M., Roos, C.: Generalized state space: a new matlab class to model uncertain and nonlinear systems as linear fractional representations (2012-2021). URL http://w3. onera. fr/smac/gss (2021)

[3] Butcher, J.C.: Numerical methods for ordinary differential equations. John Wiley & Sons (2016)

[4] Farhood, M.: Control of polytopic lpv systems with uncertain initial conditions. In: 2023 62nd IEEE Conference on Decision and Control (CDC). pp. 3712–3719 (2023)

[5] Farhood, M., Dullerud, G.E.: Control of systems with uncertain initial conditions. IEEE Transactions on Automatic Control **53**(11), 2646–2651 (2008)

[6] Fulton, N., Mitsch, S., Quesel, J.D., Völp, M., Platzer, A.: Keymaera x: An axiomatic tactical theorem prover for hybrid systems. In: Automated Deduction-CADE-25: 25th International Conference on Automated Deduction, Berlin, Germany, August 1-7, 2015, Proceedings 25. pp. 527–538. Springer (2015)

[7] 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)