PVS Linear Algebra Libraries for Verification of Control Software Algorithms

Heber Herencia-Zapana, Gilberto Perez, Pablo Ascariz, Sam Owre

National Institute of Aerospace,

SRI International, University of A Coruña

October, 2012

Outline

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- The objective of control theory is to calculate a proper action from the controller that will result in stability for the system
- The software implementation of a control law can be inspected by analysis tools
- However these tools are often challenged by issues for which solutions are already available from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

- Program verification uses proof assistants to ensure the validity of user-provided code annotations.
- These annotations may express the domain-specific properties of the code.
- However, formulating annotations correctly is nontrivial in practice.
- By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

In order to solve these two challenges this work proposes

1. Axiomatization of Lyapunov-based stability as C code annotations,
2. Implementation of linear algebra and control theory results in PVS.

In order to solve these two challenges this work proposes

1. Axiomatization of Lyapunov-based stability as C code annotations,
2. Implementation of linear algebra and control theory results in PVS.

- If there exists a positive definite function V such that $V(\xi(k)) \leq 1$ implies $V(\xi(k+1)) \leq 1$ then this function can be used to establish the stability of the system.
- This Lyapunov function, V, defines the ellipsoid $\{\xi | \, V(\xi) \leq 1\}$, this ellipsoid plays an important role for the stability preservation at the code level.

- If there exists a positive definite function V such that $V(\xi(k)) \leq 1$ implies $V(\xi(k+1)) \leq 1$ then this function can be used to establish the stability of the system.
- This Lyapunov function, V, defines the ellipsoid $\{\xi|\ V(\xi) \leq 1\}$, this ellipsoid plays an important role for the stability preservation at the code level.

Annotated with assertions in the Hoare style we get

- $$\begin{aligned} & \{pre1\} \\ u &= C_c\mathbf{x}_c + D_c y_c \\ & \{post1\} \end{aligned}$$

- $$\begin{aligned} & \{pre2\} \\ \mathbf{x}_c &= A_c\mathbf{x}_c + B_c y_c \\ & \{post2\}. \end{aligned}$$

- To use ellipsoids to formally specify bounded input, bounded state.
- Typically, an instruction S would be annotated in the following way:

$$\{x \in \mathcal{E}_P\}\ y = Ax + b\ \{y - b \in \mathcal{E}_Q\} \tag{1}$$

where the pre- and post- conditions are predicates expressing that the variables belong to some ellipsoid, with $\mathcal{E}_P = \{x : \mathbb{R}^n | x^T P^{-1} x \leq 1\}$ and $Q = APA^T$.

- To use ellipsoids to formally specify bounded input, bounded state.
- Typically, an instruction S would be annotated in the following way:

$$\{x \in \mathcal{E}_P\}\ y = Ax + b\ \{y - b \in \mathcal{E}_Q\} \tag{1}$$

where the pre- and post- conditions are predicates expressing that the variables belong to some ellipsoid, with $\mathcal{E}_P = \{x : \mathbb{R}^n | x^T P^{-1} x \leq 1\}$ and $Q = APA^T$.

An ellipsoid-aware Hoare logic

The mathematical theorem that guarantees the relations is :

Theorem

If M, Q *are invertible matrices, and*
$(x-c)^TQ^{-1}(x-c) \leq 1$ *and*
$y = Mx + b$
then
$(y-b-Mc)^T(MQM^T)^{-1}(y-b-Mc) \leq 1$

We will refer to it as the *ellipsoid theorem*.

- The pre- and post- conditions are expressed as predicates in ACSl and PVS.
- The multiplication of a matrix with a vector is defined with function vect_mult$(\textsf{matrix } A, \textsf{ vector } x)$, which returns a vector.
- Addition and multiplication of 2 matrices, multiplication by a scalar, and inverse of a matrix are declared as matrix types

- The pre- and post- conditions are expressed as predicates in ACSl and PVS.
- The multiplication of a matrix with a vector is defined with function vect_mult$(\textsf{matrix } A, \ \textsf{vector } x)$, which returns a vector.
- Addition and multiplication of 2 matrices, multiplication by a scalar, and inverse of a matrix are declared as matrix types

inverse of a matrix A, mat_inverse(A) is defined using the predicate is_invertible(A) as follows:

ACSL

```
/*@ axiom mat_inv_select_i_eq_j:
@ ∀matrixA, integer i, j;
@ is_invertible(A) && i == j ==>
@ mat_select(mat_mult(A, mat_inverse(A)), i, j) = 1
@
@ axiom mat_inv_select_i_dff_j:
@ ∀matrixA, integer i, j;
@ is_invertible(A) && i != j ==>
@ mat_select(mat_mult(A, mat_inverse(A)), i, j) = 0
@*/
```

Complex constructions or relations can be defined as uninterpreted predicates. The following predicate is meant to express that vector x belongs to $\mathcal{E}_{\mathcal{P}}$:

ACSL

- `//@ predicate` in_ellipsoid(matrix P, vector x);

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ requires P
  //@ ensures R
  Q

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ requires P
  //@ ensures R
  Q

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ requires P
  //@ ensures R
  Q

- The paramount notion in ACSL is the function contract.
- The key word requires is used to introduce the pre-conditions of the triple, and the key word ensures is used to introduce its post-conditions.
- //@ requires P
  //@ ensures R
  Q

pvs
```
in_ellipsoid?(P_0, vect_of_array(xc, 2, floatP_floatM))))))
IMPLIES
in_ellipsoid?(Q, vect_of_array(yc, 2, floatP_floatM0))
```

pvs
```
vect_of_array(yc, 2, floatP_floatM0)'vect =
Ac * vect_of_array(xc, 2, floatP_floatM)'vect
```

For both POs,

- we must first interpret the uninterpreted types and to prove the properties that are defined axiomatically.
- We must then discharge the verification conditions. This is done by using PVS and a linear algebra extension of it.

pvs
```
in_ellipsoid?(P_0, vect_of_array(xc, 2, floatP_floatM))))))
IMPLIES
in_ellipsoid?(Q, vect_of_array(yc, 2, floatP_floatM0))
```

pvs
```
vect_of_array(yc, 2, floatP_floatM0)'vect =
Ac * vect_of_array(xc, 2, floatP_floatM)'vect
```

For both POs,

- we must first interpret the uninterpreted types and to prove the properties that are defined axiomatically.
- We must then discharge the verification conditions. This is done by using PVS and a linear algebra extension of it.

In order to discharge the PO, the following libraries need to be used:

- Linear_algebra:
  - linear_map, matrices, matrix_operator, block_matrices
- Control_theory
  - ellipsoid, s_procedure_def, shur_formula

linear_map

$$T : [n, m, Vector[n] \rightarrow Vector[m]]$$

pvs

```
Mapping:TYPE= [# dom: posnat, codom: posnat, mp:
[Vector[dom]->Vector[codom]] #]
```

$$(f, g) \longmapsto [n, m, f`mp(x) + g`mp(x)]$$

pvs

```
+(f: Maping,(g: (same_dim?(f)))): Maping =
f WITH [`mp:= lambda(x: Vector[f`dom]): f`mp(x) + g`mp(x)]
```

linear_map

$$T : [n, m, Vector[n] \rightarrow Vector[m]]$$

pvs
```
Mapping:TYPE= [# dom: posnat, codom: posnat, mp:
[Vector[dom]->Vector[codom]] #]
```

$$(f, g) \longmapsto [n, m, f`mp(x) + g`mp(x)]$$

pvs
```
+(f: Maping,(g: (same_dim?(f)))): Maping =
f WITH [`mp:= lambda(x: Vector[f`dom]): f`mp(x) + g`mp(x)]
```

linear_map_def

$$\{e(i) \in \mathbb{R}^n | i = 1, \cdots, n\}$$

pvs

```
unit?(n)(e: [below[n] -> Vector[n]]):
bool = FORALL (i: below[n]): e(i)*e(i)=1
ortho?(n)(e: [below[n] -> Vector[n]]):
bool = FORALL (i,j: below[n]): (i /= j IMPLIES e(i)*e(j)=0)
```

$$x = \sum_{i=1}^{n} (xe(i))e(i)$$

pvs

```
vec_expan?(n)(e: [below[n] -> Vector[n]]):
bool = FORALL (x: Vector[n]): x = SigmaV(0,n-1,x*e)
```

linear_map_def

$$\{e(i) \in \mathbb{R}^n | i = 1, \cdots, n\}$$

pvs

```
unit?(n)(e: [below[n] -> Vector[n]]):
bool = FORALL (i: below[n]): e(i)*e(i)=1
ortho?(n)(e: [below[n] -> Vector[n]]):
bool = FORALL (i,j: below[n]): (i /= j IMPLIES e(i)*e(j)=0)
```

$$x = \sum_{i=1}^{n} (xe(i))e(i)$$

pvs

```
vec_expan?(n)(e: [below[n] -> Vector[n]]):
bool = FORALL (x: Vector[n]): x = SigmaV(0,n-1,x*e)
```

linear_map_def

$$\{e(i) \in \mathbb{R}^n | i = 1, \cdots, n\}$$

pvs

```
base?(n)(e: [below[n] -> Vector[n]]): bool = unit?(n)(e) and
ortho?(n)(e) and vec_expan?(n)(e)
```

exercise

```
e(n): [below[n]->Vector[n]] = LAMBDA(j: below[n]): LAMBDA(i:
below[n]): IF (i=j) THEN 1 ELSE 0 ENDIF
```

exercise

```
cano_base:LEMMA base?(n)(e(n))
```

linear_map_def

$$\{e(i) \in \mathbb{R}^n | i = 1, \cdots, n\}$$

pvs

```
base?(n)(e: [below[n] -> Vector[n]]): bool = unit?(n)(e) and
ortho?(n)(e) and vec_expan?(n)(e)
```

exercise

```
e(n): [below[n]->Vector[n]] = LAMBDA(j: below[n]): LAMBDA(i:
below[n]): IF (i=j) THEN 1 ELSE 0 ENDIF
```

exercise

```
cano_base:LEMMA base?(n)(e(n))
```

linear_map_def

$$h\Big[\sum_{i=0}^{l-1} x_i F_i\Big] = \sum_{i=0}^{l-1} h\big[x_i F_i\big]$$

pvs

```
linear_map_e?(h,l,n,m): bool = h`dom = n and h`codom = m and
FORALL (x: Vector[l],F: [below[l]->Vector[n]]):
h`mp(SigmaV[below[l],n](0,l-1,x*F)) = SigmaV[below[l],m](0,l -
1,x*(h`mp o F));
```

$$\begin{aligned} h[x=y] &= h[x]+h[y] \\ h[ax] &= ah[x] \end{aligned}$$

pvs

```
additive?(f): bool = FORALL (x,y: Vector[f`dom]):
f`mp(x + y) = f`mp(x) + f`mp(y)
homogeneous?(f): bool = FORALL (a: real, x: Vector[f`dom]):
f`mp(a*x) = a*f`mp(x)
linear_map?(f): bool = additive?(f) AND homogeneous?(f)
```

linear_map_def

exercise

```
linear_map_characterization: LEMMA FORALL (f: Map(n,n)):
linear_map?(f) IFF linear_map_e?(n,n)(f)
```

matrices

$$A : [n, m, (i, j) \longmapsto A_{i,j}]$$

pvs

```
Matrix: TYPE = [# rows: posnat, cols: posnat,
matrix: [below(rows), below(cols) -> real] #]
```

$$+ : (M, N) \longmapsto [n, m, (i, j) \longmapsto M_{i,j} + N_{i,j}]$$

pvs

```
+(M, (N: (same_dim?(M)))): Matrix = M WITH [ ‘matrix :=
LAMBDA (i: below(M‘rows), j: below(M‘cols)):
M‘matrix(i, j) + N‘matrix(i, j) ];
```

matrices

$$A : [n, m, (i, j) \longmapsto A_{i,j}]$$

pvs

```
Matrix: TYPE = [# rows: posnat, cols: posnat,
matrix: [below(rows), below(cols) -> real] #]
```

$$+ : (M, N) \longmapsto [n, m, (i, j) \longmapsto M_{i,j} + N_{i,j}]$$

pvs

```
+(M, (N: (same_dim?(M)))): Matrix = M WITH [ `matrix :=
LAMBDA (i: below(M`rows), j: below(M`cols)):
M`matrix(i, j) + N`matrix(i, j) ];
```

matrices

$$M * M^{-1} = M^{-1} * M = 1$$

pvs

```
inverse?(M: Square)(N: Square | N‘rows = M‘rows): bool =
M * N = I(M‘rows) and N * M = I(M‘rows)
```

pvs

```
invertible?(M: Square): bool = EXISTS (N: (inverse?(M))):
inverse?(M)(N)
```

exercise

```
inverse_unique: lemma FORALL (M: (invertible?), N, P:
(inverse?(M))): N = P
```

matrices

$$M * M^{-1} = M^{-1} * M = 1$$

pvs

```
inverse?(M: Square)(N: Square | N‘rows = M‘rows): bool =
M * N = I(M‘rows) and N * M = I(M‘rows)
```

pvs

```
invertible?(M: Square): bool = EXISTS (N: (inverse?(M))):
inverse?(M)(N)
```

exercise

```
inverse_unique: lemma FORALL (M: (invertible?), N, P:
(inverse?(M))): N = P
```

matrix_operator

$$L(n,m) : f \longmapsto [m, n, (i,j) \longmapsto f(e(n)(i))(j) = A_{ij}]$$

pvs

```
L(n,m): [Map_linear(n,m) -> Mat(m,n)] =
(lambda(f: Map_linear(n,m)):
(# rows:= m,cols:= n,matrix:=
lambda(j: below[m],i: below[n]): f`mp(e(n)(i))(j)#))
```

$$T(n,m) : A \longmapsto [n, m, x \longmapsto A * x]$$

pvs

```
T(n,m): [Mat(m,n) -> Map_linear(n,m)]=
(lambda(A: Mat(m,n)): (#dom:= n,codom:= m,mp:=
lambda(x: Vector[n]): lambda(j: below[m]):
sigma(0,A`cols-1,lambda(i: below[A`cols]):
A`matrix(j,i)*x(i))#))
```

matrix_operator

$$L(n,m) : f \longmapsto [m, n, (i,j) \longmapsto f(e(n)(i))(j) = A_{ij}]$$

pvs

```
L(n,m): [Map_linear(n,m) -> Mat(m,n)] =
(lambda(f: Map_linear(n,m)):
(# rows:= m,cols:= n,matrix:=
lambda(j: below[m],i: below[n]): f`mp(e(n)(i))(j)#))
```

$$T(n,m) : A \longmapsto [n, m, x \longmapsto A * x]$$

pvs

```
T(n,m): [Mat(m,n) -> Map_linear(n,m)]=
(lambda(A: Mat(m,n)): (#dom:= n,codom:= m,mp:=
lambda(x: Vector[n]): lambda(j: below[m]):
sigma(0,A`cols-1,lambda(i: below[A`cols]):
A`matrix(j,i)*x(i))#))
```

matrix_operator

exercise

```
Iso: LEMMA bijective?(L(n,m))
```

exercise

```
map_matrix_bij: LEMMA FORALL (A: Mat(m,n)):
L(n,m)(T(n,m)(A)) = A
iso_map: LEMMA FORALL (f: Map_linear(n,m)):
T(n,m)(L(n,m)(f)) = f
```

matrix_operator

exercise

```
comp_mult: LEMMA FORALL (g: Map_linear(n,m),f: Map_linear(m,p)):
L(n,p)(f o g) = L(m,p)(f)*L(n,m)(g)
```

pvs

```
Matrix_inv(n):type= {A: Square | squareMat?(n)(A) and
bijective?(n)(T(n,n)(A))}
```

$$\begin{array}{rccc} inv(n): & Matrix\_inv(n) & \longrightarrow & Matrix\_inv(n) \\ & A & \longmapsto & L_{n,n}((T_{n,n}(A))^{-1}) \end{array}$$

pvs

```
inv(n)(A) = L(n,n)(inverse(n)(T(n,n)(A)))
```

matrix_operator

exercise

```
comp_mult: LEMMA FORALL (g: Map_linear(n,m),f: Map_linear(m,p)):
L(n,p)(f o g) = L(m,p)(f)*L(n,m)(g)
```

pvs

```
Matrix_inv(n):type= {A: Square | squareMat?(n)(A) and
bijective?(n)(T(n,n)(A))}
```

$$\begin{array}{rccc} inv(n): & Matrix\_inv(n) & \longrightarrow & Matrix\_inv(n) \\ & A & \longmapsto & L_{n,n}((T_{n,n}(A))^{-1}) \end{array}$$

pvs

```
inv(n)(A) = L(n,n)(inverse(n)(T(n,n)(A)))
```

How is the $inverse?(M)$?

exercise

```
inv: LEMMA squareMat?(n)(M) AND bijective?(n)(T(n,n)(M))
IMPLIES inverse?(M)(inv(n)(M))
```

matrix_operator

exercise

```
prod_inv_oper:LEMMA square?(A) and squareMat?(n)(A) and
bijective?(n)(T(n,n)(A)) AND
square?(B) and squareMat?(n)(B) and bijective?(n)(T(n,n)(B))
IMPLIES
inv(n)(A*B)=inv(n)(B)*inv(n)(A)
```

$$M = \begin{pmatrix} M_1 & M_3 \\ M_2 & M_4 \end{pmatrix}$$

$$M : [row1, row2, cols1, cols2, (i,j) \longmapsto M_{i,j}] \tag{2}$$

pvs

```
Block_Matrix: TYPE = [# rows1: posnat,rows2: posnat,
cols1: posnat,cols2: posnat,
matrix: [below(rows1 + rows2), below(cols1 + cols2) -> real] #]
```

$$Block2M1 : M \longmapsto [rows1, cols1, (i,j) \longmapsto M_{i,j}]$$

pvs

```
Block2M1(M): Matrix = (# rows := M`rows1, cols := M`cols1,
matrix := LAMBDA (i: below(M`rows1), j: below(M`cols1)):
M`matrix(i,j) #)
```

$$M = \begin{pmatrix} M_1 & M_3 \\ M_2 & M_4 \end{pmatrix}$$

$$M : [row1, row2, cols1, cols2, (i, j) \longmapsto M_{i,j}] \tag{2}$$

pvs

```
Block_Matrix: TYPE = [# rows1: posnat,rows2: posnat,
cols1: posnat,cols2: posnat,
matrix: [below(rows1 + rows2), below(cols1 + cols2) -> real] #]
```

$$Block2M1 : M \longmapsto [rows1, cols1, (i, j) \longmapsto M_{i,j}]$$

pvs

```
Block2M1(M): Matrix = (# rows := M`rows1, cols := M`cols1,
matrix := LAMBDA (i: below(M`rows1), j: below(M`cols1)):
M`matrix(i,j) #)
```

schur_formula

$$M = \begin{pmatrix} M_1 & M_3 \\ M_2 & M_4 \end{pmatrix} > 0 \quad (3)$$

$$M_4 > 0 \text{ and } (M_1 - M_3(M_4)^{-1}M_2) > 0 \quad (4)$$

pvs

```
strict_schur_formula: LEMMA (Block2M2(M) =
transpose(Block2M3(M)) AND invertible?(Block2M4(M)) AND
symmetric?(Block2M1(M)) AND symmetric?(Block2M4(M)))
IMPLIES
(def_pos?(M)
IFF
def_pos?(Block2M4(M)) AND def_pos?(Block2M1(M) -
Block2M3(M)*inverse(Block2M4(M))*Block2M2(M)))
```

S-procedure

Thus the implication that must be proved is as follows:

$$\{\mathbf{x}^T P\mathbf{x} \leq 1, \text{ and } y^2 \leq 1\}\text{implies } (A\mathbf{x} + By)^T P(A\mathbf{x} + By) \leq 1. \quad (5)$$

Applying the S-procedure the implication 5 is equivalent to: Exist a $\mu \in \mathbb{R}$

$$(A\mathbf{x} + By)^T P(A\mathbf{x} + By) - \mu\mathbf{x}^T P\mathbf{x} - (1 - \mu)y^2 \leq 0.$$

S-procedure

Control_theory

Let the linear functionals $\sigma_k : R^n -> R$ and consider the following two conditions

1. S_1: For all $k = 1, 2, \ldots, N$, $\sigma_k > 0$ implies $\sigma_0 \geq 0$

pvs
```
s1_condition?(m)(beta: fun_constraint(m),f: Map(n,1)): bool
= FORALL (x: Vector[n]): pos_constraint_point?(m)(beta,x)
IMPLIES f`mp(x)(0) >= 0
```

2. S_2: There exists $\tau_k \geq 0$, $k = 1, 2, \ldots, N$ such that

$$\sigma_0(y) - \Sigma_{k=1}^{N} \tau_k \sigma_k(y) \geq 0, \ \forall y \in R^n$$

pvs
```
s2_condition?(m)(beta: fun_constraint(m),f: Map(n,1)): bool
= EXISTS (r: pos_scalar_family(m)): (FORALL (x: Vector[n]):
f`mp(x)(0) - sigma[below[m]](0,m - 1, LAMBDA(i: below[m]):
r(i)*beta(i)`mp(x)(0)) >= 0)
```

Control_theory

pvs

```
ellipsoid: LEMMA ∀ (n:posnat, Q, M: SquareMat(n), x, y, b, c:
Vector[n]):
bijective?(n)(T(n,n)(Q)) AND bijective?(n)(T(n,n)(M))
AND (x-c)*(inv(n)(Q)*(x-c))≤ 1
AND y=M*x + b
IMPLIES
(y-b-M*c)*(inv(n)(M*(Q*transpose(M)))*(y-b-M*c))≤ 1
```

Control_theory_verification

$$\{x \in \mathcal{E}_P\}\ y = Mx + b\ \{y - b \in \mathcal{E}_Q\} \tag{6}$$

pvs

```
in_ellipsoid?(P, X) and Y=MX+b
IMPLIES
in_ellipsoid?(MQM^T,Y-b)
```

pvs

```
bijections :LEMMA
bijective?(2)(T(2,2)(Q) AND  bijective?(2)(T(2,2)(M))
```

Control_theory_verification

$$\{x \in \mathcal{E}_P\}\ y = Mx + b\ \{y - b \in \mathcal{E}_Q\} \tag{6}$$

- pvs
  ```
  in_ellipsoid?(P, X) and Y=MX+b
  IMPLIES
  in_ellipsoid?(MQM^T,Y-b)
  ```
- pvs
  ```
  bijections :LEMMA
  bijective?(2)(T(2,2)(Q) AND  bijective?(2)(T(2,2)(M))
  ```

Conclusions

- We have outlined a global approach to validate stability properties of code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- Linear algebra PVS libraries can be used for the formal specification of control theory properties
- We have defined a PVS library able to manipulate predicates over the code.

Conclusions

- We have outlined a global approach to validate stability properties of code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- Linear algebra PVS libraries can be used for the formal specification of control theory properties
- We have defined a PVS library able to manipulate predicates over the code.

Conclusions

- We have outlined a global approach to validate stability properties of code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- Linear algebra PVS libraries can be used for the formal specification of control theory properties
- We have defined a PVS library able to manipulate predicates over the code.

Conclusions

- We have outlined a global approach to validate stability properties of code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- Linear algebra PVS libraries can be used for the formal specification of control theory properties
- We have defined a PVS library able to manipulate predicates over the code.

Conclusions

- We have outlined a global approach to validate stability properties of code implementing controllers.
- Our approach requires the code to be annoted by Hoare triples,
- Linear algebra PVS libraries can be used for the formal specification of control theory properties
- We have defined a PVS library able to manipulate predicates over the code.

- Linear Algebra:
  - sigma_lemmas,linear_map, sigma_vector, linear_map_def,vect_of_vect
  - matrices, matrix_operator, matrix_lemmas, block_matrices
- Control Theory:
  - ellipsoid, convex_def, s_procedure_def
  - schur_prelim, schur_formula