PVS Linear Algebra Libraries for Verification of Control Software Algorithms in C/ACSL ⋆

Heber Herencia-Zapana[1], Romain Jobredeaux[2], Sam Owre[3], Pierre-Loïc Garoche[4], Eric Feron[2], Gilberto Perez[5], and Pablo Ascariz[5]

[1] National Institute of Aerospace, Hampton, VA
[2] Georgia Institute of Technology, Atlanta, Georgia
[3] SRI International, Menlo Park, California
[4] ONERA, The French Aerospace Lab, Toulouse, France
[5] University of A Coruña, Coruña, Spain

Abstract. The problem of ensuring control software properties hold on their actual implementation is rarely tackled. While stability proofs are widely used on models, they are never carried to the code. Using program verification techniques requires express these properties at the level of the code but also to have theorem provers that can manipulate the proof elements. We propose to address this challenge by following two phases: first we introduce a way to express stability proofs as C code annotations; second, we propose a PVS linear algebra library that is able to manipulate quadratic invariants, i.e., ellipsoids. Our framework achieves the translation of stability properties expressed on the code to the representation of an associated proof obligation (PO) in PVS. Our library allows us to discharge these POs within PVS.

1 Introduction

Critical computing systems, typically driving machinery or vehicles, are those in which failure may result in unacceptable human losses. Examples of critical systems include fly-by-wire controls on an aircraft or on a manned spacecraft, radiation therapy equipment, and nuclear power plant safety systems. Digital computation facilitates the design and the implementation of complex control algorithms. The software implementation of a control law can be inspected by analysis tools [7, 22, 24], however these tools are often challenged by issues for which solutions are already available from control theory.

Control theory is a branch of engineering that focuses on the behavior of dynamical systems. The desired output of a system is called the reference point. When one or more output variables of a system need to follow a certain reference

⋆ This work was supported by the National Aeronautics and Space Administration under NASA Cooperative Agreement NCC-1-02043, the National Science Foundation under Grant CNS - 1135955, the Army Research Office under MURI Award W911NF-11-1-0046, the Air Force Research Laboratory as part of the CertaAMOR program, the Dutton/Ducoffe Professorship at Georgia Tech, FNRAE CAVALE project, and Spanish MEC project TIN2009-14562-C05-04.

over time, a controller manipulates the inputs of the system to obtain the desired effect on its output. The objective of control theory is to calculate a proper action from the controller that will result in stability for the system, that is, the system will hold the reference point and not oscillate around it. Among the different mathematical approaches to prove stability of the controller or the controlled system, Lyapounov based stability relies on ellipsoid characterization and the so-called S-procedure [3, 14, 15]. These works also address the expression of the proof as C code annotation, but do not give means to automate this expression nor to prove it on C code.

Program verification based on deductive methods uses either automatic decision procedures or 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 (i.e., precisely as the domain expert really intends) is nontrivial in practice [2, 12]. By correctly, we mean that the annotations formulate stability properties of an intended mathematical interpretation from control theory.

The challenges of domain-specific code annotation arise along two directions. First, the domain knowledge has its own inherent complexity. When considering control theoretic issues, the annotations need to allow the expression of stability properties using ellipsoids and the S-procedure in the way that was proposed in [3, 14, 15]. Second, the code annotations are meant to be manipulated by automatic theorem provers. But most of the automatic decision procedures are restricted to decidable logics such as Boolean satisfiability or linear arithmetic, which are generally too weak to express the desired user-defined and domain-specific code annotations.

In order to solve these two challenges this paper proposes an axiomatisation of Lyapunov-based stability as C code annotations, and the implementation of linear algebra and control theory results in PVS [28, 26], respectively. The mechanism of theory interpretations [27] enables the translation of POs expressed on the C code as PVS control theory proof obligations. The proof of these obligations can then be discharged using control theory results expressed and proved with our PVS linear algebra library.

Related works To our knowledge, apart from the work of [3, 14, 15], no other research endeavor addresses the issue of proving in the C code the high-level correctness properties of control systems such as stability. Some successful attempts have been made at extracting quadratic invariants from the code, in [1] and [13].

Regarding the prover part of our framework, the developments of tools that support the proof of properties in real arithmetic or real linear algebra is a current concern. However these early development do not cover the entire range of mathematics and are often restricted to specific sub-areas. For example a recent project, Coquelicot, develops real functional analysis , Gaussian elimination and basic properties of matrices and determinants for the Coq proof assistant [18]. Generic design patterns were proposed to define algebraic structures[17]. Formalization and instrumentation of Euclidean spaces also appears to be a new concern for Isabelle/HOL [20]. We should also mention automatic decision procedures for floating point arithmetics, such as Gappa [11]. A PVS formalization of multivariate Bernstein polynomials was presented in [25]. In general however, none of these recent extensions of theorem provers are able to deal with the properties of interest in this paper.

Outline Section 2 reminds the reader of elements of control theory software analysis [3, 14, 15], i.e., it describes the controller stability proof and its expression as C code annotations with Hoare triples. It also discusses issues to be handled on the theorem prover part, mainly the need for two main theorems: one related to ellipsoids and one on the S-procedure. Section 3 introduces our axiomatisation of stability proofs as C code annotations using Hoare triples. Section 4 presents the implementation of linear algebra and theorems related to ellipsoids in PVS. Finally, Section 5 explains how we plan to map POs generated from the Hoare triple stability annotations to PVS, using theory interpretation in PVS, and how to use the ellipsoid library to discharge these POs. Last, Section 6 concludes the paper and discusses future research.

2 Stability and correctness

2.1 Expressing and proving stability of a control system

The basic module for the description of a controller can be presented as

$$\begin{aligned}&\xi(k+1) = f(\xi(k), \nu(k)),\ \xi(0) = \xi_0\\ &\zeta(k) = g(\xi(k), \nu(k))\end{aligned}$$

where $\xi \in \mathbb{R}^n$ is the state of the controller, ν is the input of the controller and ζ is the output of the controller. This system is bounded-input, bounded state stable if for every ε there exists a δ such that $||\nu(k)|| \leq \varepsilon$ implies $||\xi(k)|| \leq \delta$, for every positive integer k. 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; for more details see [8]. 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, for more details see [3, 14, 15].

2.2 Hoare triple and deductive methods

Since their early formalization by Hoare [21] and later by Dijsktra [10], deductive methods from Hoare triple to weakest precondition computation have been widely used on imperative code.

In his initial proposal, Hoare requires a program to be annotated line by line by the invariants that should hold at each program point. He also provides instruction-specific rules that ensure the soundness of the code with respect to the annotation system.

Because it was, in general, not realistic to require a line-based set of annotations, Dijkstra later proposed the weakest precondition computation and the verification conditions, that automatically generate a PO with respect to a Hoare style annotation for a block of instructions. Most software analysis tools which use Hoare Logic are based on this algorithm.

Our current approach does not consist in automating the proof of stability, but rather, given a stability proof, to check the proof automatically. As was suggested in [3, 14, 15], we consider a line-by-line annotation of the code, allowing a Hoare-like reasoning approach to the program.[6]

In general, Hoare proofs are sound, i.e., the proved property indeed holds, if and only if the program terminates. They are complete if the underlying logic – the one used in pre- and post-condition – is complete.

2.3 Application to controller stability: an ellipsoid-aware Hoare logic

We present here the two main patterns used in stability proofs. The main concerns are: to relate quadratic invariants and affine or linear combinations of variables on the one hand – the ellipsoid affine combination theorem; and to extract one quadratic invariant out of implications between several quadratic invariants on the other hand – the S-procedure.

Ellipsoid affine combination theorem The use of ellipsoids to formally specify bounded input, bounded state stability was proposed in [3, 14, 15] following prior work [6]. Stability is then expressed as a predicate stating that the system state remains in a given ellipsoid. 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 $Q = APA^T$.

The mathematical theorem that guarantees the relations in (1) is now stated:

Theorem 1. *If M, Q are invertible matrices, and $(x-c)^T Q^{-1}(x-c) \leq 1$ 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 affine combination theorem.* More details about this result in the context of control theory can be found in [6, 23].

The S-procedure A second common need is to prove the implication between two quadratic invariants. In the initial Hoare proposal [21], the post-condition of a

[6] or equivalently a limited-depth Dijkstra weakest precondition.

$$\{P_1\}\ S_1\ \{Q_1\}$$
$$\{Q_1\}\ \texttt{nil}\ \{P_2\}$$
$$\{P_2\}\ S_2\ \{Q_2\}$$

Fig. 1. Conseq. rule

statement is exactly the pre-condition of its successor. A *consequence rule* allows to transform a post-condition of a statement into another pre-condition for the following statement. This can be understood as the introduction of a `nil` statement that contains this translation of predicates as illustrated in Figure 1. This unavoidable step allows software analyzers to manipulate the annotations along the code. The PO associated to this new `nil` statement is $Q_1 \implies P_2$.

A frequent proof pattern when using ellipsoid-based stability proofs is to show that the inequality $x^T Px \leq 1$ implies $y^T Qy \leq 1$. Such implications are usually difficult to prove. We need to give conditions under which, given symmetric matrices A_0 and A_1, statement 2. implies statement 1. in the following:

$$\begin{aligned} &1. \quad \forall x \in \mathbb{R}^n : x^T A_1 x \geq 0 \implies x^T A_0 x \geq 0 \\ &2. \quad \exists a \in \mathbb{R} : \forall x \in \mathbb{R}^n x^T (A_0 - aA_1)x > 0 \end{aligned}$$

From [3, 14, 15], a typical property for the composition of Hoare triples is to prove that the implication

$$\{\mathbf{x}_c \in \mathcal{E}_P,\ y_c^2 \leq 1\} \text{ implies } \{A_c\mathbf{x}_c + B_c y_c \in \mathcal{E}_P,\ y_c^2 \leq 1\}$$

is a consequence of the inequality

$$(A_c\mathbf{x}_c + B_c y_c)^T P (A_c\mathbf{x}_c + B_c y_c) - \mu\mathbf{x}^T P\mathbf{x} - (1-\mu)y^2 \leq 0.$$

This type of property may be proved using the following theorem, which the *S-procedure* [6, 23] is a by-product of.

Theorem 2. *Let the real valued functionals* $\sigma_k : \mathbb{R}^n \to \mathbb{R}$ *where* $k = 0, 1, 2, \ldots, N$ *and consider the following two conditions:*

1. S_1*:* $\forall y \in \mathbb{R}^n : (\forall k = 1, 2, \ldots, N : \sigma_k(y) > 0) \implies \sigma_0(y) \geq 0$
2. S_2*: There exists* $\tau_k \geq 0$*,* $k = 1, 2, \ldots, N$ *such that*

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

Then $S_2 \implies S_1$*.*

The *S-procedure* is the method of verifying S_1 using S_2.

3 Defining quadratic invariants as code annotations

Now that we know the annotations that we want to generate on the code, we have to find a concrete way to express them on actual C code. The ANSI/ISO C Specification Language (ACSL) [5] allows its user to specify the properties of a C program within comments, in order to be able to formally verify that the implementation respects these properties. This language was proposed as part of the Frama-C platform [9], which provides a set of tools to reason on both C programs and their ACSL annotations. ACSL offers the means to extend its internal logic with user-defined theory, i.e., types, constructors, functions, predicates and axioms.

We outline the axiomatisation in ACSL to fit our needs, which consist of expressing ellipsoid-based Hoare triples over C code. We first present the axiomatisation of linear algebra elements in ACSL. Then we present the Hoare triple annotations in ACSL and the POs generated by them.

3.1 Linear algebra in ACSL predicates

The following abstract types are declared: matrix, vector, integer, and real. With these abstract types, basic matrix operations and properties are introduced : a component of the matrix is a real number accessed using the function mat_select (matrix A, integer i, integer j), total number of rows and columns are integers accessed with mat_row(matrix A), and mat_col(matrix A), respectively. The multiplication of a matrix with a vector is defined with function vect_mult(matrix A, vector x), which returns a vector. The concatenation of vectors x and y, itself a vector, is accessed through Vconcat(vector x, vector y). Addition and multiplication of 2 matrices, multiplication by a scalar, and inverse of a matrix are declared as matrix type as follows:

$$\mathsf{mat\_add}(\mathsf{matrix}\ A, \mathsf{matrix}\ B), \quad \mathsf{mat\_mult}(\mathsf{matrix}\ A, \mathsf{matrix}\ B)$$
$$\mathsf{mat\_mult\_scal}(\mathsf{matrix}\ A, \mathsf{real}\ a), \text{ and } \mathsf{mat\_inverse}(A).$$

The matrix operations are defined axiomatically, for example the 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
@*/
```

In the same axiomatic way, the main matrix operations are declared. Complex constructions or relations can be defined as uninterpreted predicates, i.e., with no associated axiom. The semantics of those predicates are introduced in PVS, as discussed in section 5. The following predicate is meant to express that vector x belongs to $\mathcal{E}_{\mathcal{P}}$:

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

And last, a set of typing functions, associated to a set of axioms, such as mat_of_array or vect_of_array, is used to associate an ACSL matrix type to a C array.

ACSL
```
//@ logic matrix mat_of_array{L}(float *A, integer row, integer col);
```

3.2 Linear Algebra Code Annotations

The paramount notion in ACSL is the function contract, [7]. It can be understood as a Hoare triple for a whole function. 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. Dealing with a low-level language has its disadvantages: we need to deal with memory issues. In general, we want all functions to be called with valid pointers as arguments, i.e., valid array and therefore valid matrices. This is what the built-in ACSL predicate valid does. The followings snippet shows how the contract can be written using mat_select and mat_of_array,

ACSL
```
/*@ requires (valid(a + (0..3)));
@ ensures ∀integer i, j; 0 ≤ i < 2 && 0 ≤ j < 2
@ ==> mat_select(mat_of_array(a, 2, 2), i, j) == 0;
@ */
void zeros_2x2(float* a)
{ a[0]=0; a[1]=0; a[2]=0; a[3]=0; }
```

In the following example the uninterpreted predicate in_ellipsoid is used:

ACSL
```
/*@ requires
@ (valid(xc + (0..1))) && (valid(yc)) && (valid(u)) &&
@ in_ellipsoid(Q_mu, Vconcat(vect_of_array(xc, 2), vect_of_array(yc, 1)));
@ ensures
@ in_ellipsoid(U_bound, vect_of_array(u, 1)) &&
@ in_ellipsoid(Q_mu, Vconcat(vect_of_array(xc, 2), vect_of_array(yc, 1)));
@ */
void inst2(float* xc, float* yc, float* u)
{ u[0] = 564.48*xc[0] - 1280*yc[0]; }
```

where Q_{mu}, $C = [564.48 \quad 0 \quad -1280]$ are matrices and

$$U_{\mathsf{bound}} = \mathsf{mat\_inv}(\mathsf{mat\_mult}(\mathsf{mat\_mult}(C, \mathsf{mat\_inv}(Q_{\mathsf{mu}})), \mathsf{transpose}(C)))$$

One important assumption which will be made throughout the rest of this article is that all computations in the program yield their exact, real result. Errors due to floating point approximations are thus not taken into account. The Frama-C toolset offers the possibility of making this assumption by including the pragma 'JessieFloatModel(Math)'. Verification conditions are then generated with no concern for floating point computations.

3.3 Generating Proof Obligations

Frama-C tools do not require an annotation at each line as proposed by Hoare. They rather rely on Dijkstra-style weakest precondition calculus to compute the backward semantics of the function code S to the post-condition Q and generate the weakest pre-condition $wp(S, Q)$ that guarantees to obtain Q after executing S. The generated PO is then $P \implies wp(S, Q)$ where P is the pre-condition.

Focusing on single line contract, i.e., the Hoare annotations as described in [21], these tools will generate the following two kinds of POs when used with ellipsoid-based annotations.

First, we have the POs associated with the use of the ellipsoid affine combination theorem, see Equation 1:

```
ACSL
in_ellipsoid(matrix $P$, vector $x$)
IMPLIES
in_ellipsoid(matrix $Q$, vector (vect_mult(matrix $A$, vector $x$))
```

One can remark that both axiom-based and uninterpreted predicates are expressed in the same way. The only difference is that axiom-based predicate definitions appear in the other generated files of the proof obligation generation phase.

Second, we have the POs associated with the use of the S-procedure, cf. Theorem 2:

```
ACSL
in_ellipsoid($A_1, x$) IMPLIES in_ellipsoid($A_0, x$)
IF AND ONLY IF
in_ellipsoid(mat_add($A_0$, mat_mult_scal($A_1, a$)), $x$)
```

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 using the appropriate theorem. This is done by using PVS and a linear algebra extention of it, presented below.

4 Linear algebra in PVS

First, we define matrices, vectors, etc. in PVS in a way that can be used to interpret in_ellipsoid and S-procedure. Second, we provide the main theorems and basic principles of linear algebra in PVS that are needed to support this interpretation. General linear algebra references include [19, 4, 16].

4.1 Bases for linear algebra in PVS

We first define maps as follows:

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

This is the set of functions that take a vector and return a vector. A linear map is defined as a map $h \in$ `Mapping` with the linear property $h(\Sigma_{i=0}^{N}(a(i)x(i))) = \Sigma_{i=0}^{N}(a(i)h(x(i)))$. this property in PVS is expressed as follows:

```
PVS
linear_map_e?(h,l,n,m): bool = h`dom=n and  h`codom=m and
  ∀(x: Vector[l], F: [below[l]->Vector[n]]):
    h`mp($\Sigma_{i=0}^{l-1}$x(i)*F(i)) = $\Sigma_{i=0}^{l-1}$(x(i)*(h`mp(F(i))))
linear_map_e?(n,m)(h): bool = ∀(l): linear_map_e?(h,l,n,m)
Map_linear(n,m): TYPE = {h: Map(n,m) | linear_map_e?(n,m)(h)}
```

The algebra of matrices is the set of matrices together with the operations addition, multiplication and multiplication by scalar, and these operations satisfy the associative and commutative properties. The algebra of linear maps is the set of linear maps with the operations of composition and multiplication and preserving the associative and commutative properties [4, 16]. We define the operator `L(n,m)` from the algebra of linear maps `Map_linear(n,m)` to the algebra of matrices `Mat(m,n)` as follows:

PVS
```
L(n,m)(f) = (# rows:=m, cols:=n, matrix:=λ(j,i): f`mp(e(n)(i))(j) #)
```

where `f`$\in$ `Map_linear(n,m)`. We define the operator `T(n,m)` from `Mat(m,n)` to `Map_linear(n,m)` as follows:

PVS
```
T(n,m)(A) = (# dom:=n, codom:=m,
               mp:=λ(x,j):
                     Σ_{i=0}^{A`cols−1}(λ(i): A`matrix(j,i)*x(i)
               #))
```

With these two operators connecting linear maps and matrices, the following PVS lemmas prove the isomorphism between them:

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

Because of the isomorphism between these two operators, the following lemma holds:

PVS
```
L_inverse: LEMMA inverse(L(n,m))=T(n,m)
```

More practical lemmas for proving properties in PVS are:

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

An important consequence of the isomorphism is the relation between the operations of the isomorphic spaces. For example, the composition of two linear maps is equivalent to the multiplication of their corresponding matrices:

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

and the addition of two linear maps is equivalent to the addition of their corresponding matrices:

PVS
```
iso_add: LEMMA ∀(f, g:Map_linear(n,m):
                    L(n,m)(f + g) = L(n,m)(f) + L(n,m)(g)
```

The main reason for the isomorphism is to define the inverse of a matrix; one condition for the existence of the inverse in linear maps is that the linear map needs to be bijective. The space of matrices having inverses is defined in PVS as follows:

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

where `Square` is the type of matrices having the same number of rows and columns, `squareMat?(n)(A)` is the type of matrices having the same number of rows and column and equal to `n`, and the predicate `bijective?(n)(T(n,n)(A))` expresses that the linear map, `T(n,n)(A)`, associated to a matrix `A` is bijective.

The inverse operator, `inv(n)`, maps `Matrix_inv(n)` to `Matrix_inv(n)` and is defined as follows:

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

It is important to note that the operators `L`, `T` and the isomorphism play an important role in this definition. The main lemmas for the matrix inverse are proved in PVS, such as: the multiplication of the matrix and its inverse is equal to the identity matrix, `I(n)`, the inverse of a transpose matrix is equal to the transpose of its inverse or the distributive property of the inverse over matrix multiplication.

The PVS libraries also have basic lemmas from the matrices theory such as the solution to a matrix equation, the transpose of matrix multiplication, and the multiplication of matrix transpose and vectors.

One important point of this development is that the conditions under which the inverse of a matrix exists is that the linear map associated to the matrix is a bijective map. A common test for the existence of the inverse of a matrix is that the determinant of the matrix be not equal to zero, The equivalence between these two conditions needs to be implemented in PVS for which more mathematical theories such as multi-linear forms, convex spaces, and so forth are currently under development. The two conditions of S-procedure, i.e., Theorem 2, were implemented as follows:

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

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

We are still working on the proof of the equivalence of these two conditions, one result that is needed for the proof is the Hyperplane theorem, which is a theorem from real analysis currently under development in PVS.

4.2 Ellipsoid affine combination theorem in PVS

The implication associated to Equation 1 can be proved using the following theorem implemented in PVS.

```
                                                                          PVS
ellipsoid_affine_comb: 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
```

This lemma was proved in PVS, the main part of the proof was to show that replacing y by $M * x + b$ in $(y - b - M * c) * (inv(n)(M * (Q * transpose(M))) * (y - b - M * c))$, we obtain $(x - c) * (inv(n)(Q) * (x - c))$. In order to manipulate $(y - b - M * c) * (inv(n)(M * (Q * transpose(M))) * (y - b - M * c))$ the following PVS lemmas *trans_mat_scal*, *prod_inv_oper*, *tran_inv_oper*, *transpose_product* and basic properties of *SigmaV* were used.

5 Mapping ACSL predicates to PVS linear algebra concepts

On the one hand, using ACSL and the Frama-C framework, we were able to generate POs about the ellipsoid predicate. Frama-C tools even make it possible to express the PO in PVS, along with a complete axiomatisation in PVS of C programs semantics. On the other hand, we have developed a PVS library that is able to reason about these properties.

We now must link these two worlds: ACSL ellipsoids predicate proof obligation in PVS must be connected with with our linear algebra PVS library. We first propose to relate ASCL constructs to PVS Linear Algebra library elements and achieve a proof on the latter. A current ongoing approach, presented at the end of the section, is to automate this mapping using theory interpretations in PVS.

5.1 Mapping ACSL predicates to PVS linear algebra

Frama-C tools automatically generate the proof obligations (POs) associated with a function contract, in our case, a Hoare triple. Depending on the back-end used, the PO can be expressed either to target an automatic decision procedure such as an SMT-solver, or to target a proof assistant, like Coq or PVS.

Using the PVS back-end, both the PO and all the axiomatisation of C semantics and all ACSL defined theories and predicates are expressed in PVS files. We now map PVS-encoded version of ACSL predicates into their PVS linear algebra library equivalent. A few examples of how such a mapping is performed are given in the rest of this section.

The ACSL logic function mat_of_array(ptr, n, m), when put through the PVS back-end, appears with an additional argument, mat_of_array(ptr, n, m, mem), which describes the memory state at the point where the function is used. The mapping for this function and the accessor mat_select are as follows:

```
                                                                          PVS
mat_of_array(ptr, n, m, mem) = A where
 A ∈ Matrix,  A‘rows = n,  A‘cols = m
 FORALL (i: below(A‘rows), j: below(A‘cols)):
  A‘matrix(i,j) = select[real, floatP](mem, shift[floatP](ptr, i*n+j))

mat_select(A, i, j) = A‘matrix(i,j) where A ∈ Matrix
```

The `select` and `shift` functions are part of the axiomatisation of C semantics pertaining to memory access.
Function mat_inverse and predicate is_invertible are interpreted as follows:

```
                                                                          PVS
mat_inverse(matrix A) := inv(n)(A)
is_invertible(matrix A)
      := square?(A) AND squareMat?(n)(A) AND bijective?(n)(T(n, n)(A))
```

And the following axiomatic definition of inverse

```
                                                                         ACSL
/* @axiom mat_inv_select_eq: ∀ matrix A, integer: i, j; i=j
@ is_invertible(M)  ⟹  mat_select(mat_mult(A, mat_inverse(A)),i,j) = 1
@*/
```

is mapped to the following lemma:

```
                                                                          PVS
LEMMA squareMat?(n)(M) and bijective?(n)(T(n,n)(M)) and
             i=j and i≤n
                IMPLIES
                (M*inv(n)(M))‘(i,j) = 1
```

which was also proved, using the concepts introduced in the linear algebra library and basic properties in PVS.

In the same way we develop the interpretation for the basic matrix operators such as addition, transposition, multiplication by scalars, multiplication by vectors, and so forth.

5.2 Discharging Proof Obligations

We now sketch the typical use of our framework to prove a specific Hoare triple. We consider the following single line function annoted with ellipsoid-based pre- and post-condition. This function corresponds to the definition of the linear combination of matrices as presented in Equation (1).

```
                                                                         ACSL
/*@ requires (valid(xc + (0..1))) && (valid(yc)) && (valid(u)) &&
@ in_ellipsoid(Q_mu, Vconcat(vect_of_array(xc,2), vect_of_array(yc,1)));
@ ensures in_ellipsoid(U_bound, vect_of_array(u,1)) &&
@ in_ellipsoid(Q_mu,Vconcat(vect_of_array(xc,2), vect_of_array(yc,1)));
@ */
void inst2(float* xc, float* yc, float* u) {
u[0] = 564.48*xc[0] - 1280*yc[0];
}
```

Using Frama-C and its PVS back-end on these annotations generate the following PVS PO:

```
PVS
FORALL ...  in_ellipsoid(Q_mu,
             Vconcat(vect_of_array(xc, 2, floatP_floatM),
                     vect_of_array(yc, 1, floatP_floatM))) IMPLIES
 FORALL (result: real) :
 result = select[real, floatP](floatP_floatM, shift[floatP](xc, 0))
 IMPLIES
   FORALL (result0: real) :
   result0 = select[real, floatP](floatP_floatM, shift[floatP](yc, 0))
   IMPLIES
     FORALL (floatP_floatM0: memory[floatP, real]) :
     floatP_floatM0 = store[floatP, real]
        (floatP_floatM, u, 564.48 * result - 1280.0 * result0)
     IMPLIES in_ellipsoid(U_bound, vect_of_array(u, 1, floatP_floatM0))
     AND in_ellipsoid(Q_mu,
                      Vconcat(vect_of_array(xc, 2, floatP_floatM0),
                              vect_of_array(yc, 1, floatP_floatM0)))
```

In order to discharge this PO, we first give a meaning to the predicate `in_ellipsoid`.

```
PVS
in_ellipsoid(matrix P, vector x)=x*(inv(n)(P)*x)≤ 1
```

Then, after skolemisation, we can split the conjunction in the consequence and prove the two implications using the ellipsoid affine combination theorem in PVS, presented in paragraph 4.2.

5.3 Theory interpretations

Theory interpretation is a logical technique for relating one axiomatic theory to another. This technique makes it possible to show that one collection of theories is correctly interpreted by another collection of theories under a user-specified interpretation for the uninterpreted types and constants. PVS supports theory interpretations [27]. A theory instance is generated and imported, while the axiom instances become POs to ensure that the interpretation is valid. Interpretations can be used to show that an implementation is a correct refinement of a specification, or that an axiomatically defined specification is consistent.

We outline here a possible use of theory interpretation to automate this mapping between the two theories. This will be developed as future work.

Jessie to PVS The Jessie plugin translates obligations to uninterpreted types and constants of PVS. When generating the PVS file associated to an annotated C file, all ACSL definitions and the generated POs are declared under a new theory `acsl_theory`. This theory contains new types, for example, the uninterpreted type matrix. To provide an interpretation for matrix, we first import the interpreting theory `matrices`, then we import the uninterpreted theory `acsl_theory` with mappings for matrix, matselect, mat_mult, etc., as shown below.

```
                                                                    PVS
importing matrices
importing acsl_theory{{ matrix := Matrix,
               mat_select := λ M, i, j: M‘matrix(i, j),
               mat_mult := *,
               ... }}
```

This action generates POs corresponding to the axioms of the theory `acsl_theory`. In a similar fashion, all uninterpreted predicates may be given interpretations, and any axiom instances become POs. In the early stages of development, predicates such as in_ellipsoid may not have axioms provided in Jessie, in which case there is no guarantee of soundness. However, the system still generates POs corresponding to type correctness conditions (TCCs).

6 Conclusion and Future Work

We have described a global approach to validate stability properties of C code implementing controllers. Our approach requires the code to be annoted by Hoare triples, following [3, 14, 15], proving the stability of the control code using ellipsoid affine combinations and S-procedure.

We have defined an ACSL extension to describe predicates over the code, as well as a PVS library able to manipulate these predicates. This library contains matrices, linear maps, ellipsoid affine combination theorem, isomorphism between matrices and linear maps and theirs basic properties. The PVS libraries can be found at http://shemesh.larc.nasa.gov/fm/ftp/larc/PVS-library/pvslib.html.

We have also outlined an approach based on theory interpretation that maps proof obligations generated from the code to their equivalent in this new PVS library. This mapping allows to discharge POs using the ellipsoid affine combination and S-procedure theorems implemented in PVS.

Currently we are working on the automatic translation, using theory interpretations, of POs in ACSL about matrices properties into POs in PVS and discharging these POs using our linear algebra libraries. We are also working on the proof of the S-procedure in PVS, which involves more mathematical results such as hyperplane theorem, multilinear forms etc. As future research we are going to develop PVS strategies for automatically discharging proof obligation generated from the ACSL annotations of the control code and also to prove the equivalence between $Det(A) \neq 0$ and the inversibility of matrix A.

Acknowledgments

The authors would like to thank Dr. A. Goodloe for his suggestion of the use of the Frama-C toolset and his help in axiomatising of linear algebra in ACSL.

References

1. Adjé, A., Gaubert, S., Goubault, E.: Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In: ESOP. pp. 23–42 (2010)

2. Ahn, K.Y., Denney, E.: Testing first-order logic axioms in program verification
3. Alegre, F., Feron, e., Pande, S.: Using ellipsoidal domains to analyze control systems software. CoRR abs/0909.1977 (2009)
4. Axler, S.: Linear Algebra Done Right. Springer, second edn. (1997)
5. Baudin, P., Filliâtre, J.C., Marché, C., Monate, B., Moy, Y., Prevosto, V.: ACSL: ANSI/ISO C specification language. preliminary design (version 1.5)
6. Boyd, S., El Ghaoui, L., Feron, E., Balakrishnan, V.: Linear Matrix Inequalities in System and Control Theory, Studies in Applied Mathematics, vol. 15. SIAM (Jun 1994)
7. Burghardt, J., Gerlach, J., Hartig, K.: ACSL by example towards a verified C standard library version 4.2.0 for Frama-C beryllium 2 (2010)
8. Chen, C.T.: Linear System Theory and Design. Oxford University Press, USA, third edn. (1998)
9. Correnson, L., Cuoq, P., Puccetti, A., Signoles, J.: Frama-C user manual
10. Dijkstra, E.: A Discipline of Programming. Prentice-Hall (1976)
11. de Dinechin, F., Quirin Lauter, C., Melquiond, G.: Certifying the floating-point implementation of an elementary function using Gappa. IEEE Trans. Computers 60(2), 242–253 (2011)
12. Eriksson, J., Back, R.J.: Applying PVS background theories and proof strategies in invariant based programming. In: ICFEM'10. pp. 24–39. Springer (2010)
13. Feret, J.: Static analysis of digital filters. In: ESOP. pp. 33–48 (2004)
14. Feron, E.: From control systems to control software. Control Systems, IEEE 30(6) (2010)
15. Feron, E., Alegre, F.: Control software analysis, part I open-loop properties. CoRR abs/0809.4812 (2008)
16. Friedberg, S., Insel, A., Spence, L.: Linear Algebra. Prentice-Hall, third edn. (1997)
17. Garillot, F., Gonthier, G., Mahboubi, A., Rideau, L.: Packaging Mathematical Structures. In: Theorem Proving in Higher Order Logics. LNCS, vol. 5674. Springer (2009), http://hal.inria.fr/inria-00368403/en/
18. Gonthier, G.: Point-free, set-free concrete linear algebra. In: ITP. pp. 103–118 (2011)
19. Halmos, P.: Finite-Dimensional Vector Spaces. Springer (1974)
20. Harrison, J.: The HOL light formalization of euclidean space. In: AMS Special Session on Formal Mathematics for Mathematicians (2011)
21. Hoare, C.A.R.: An axiomatic basis for computer programming. Comm. ACM 12, 576–580 (October 1969)
22. Izerrouken, N., Thirioux, X., Pantel, M., Strecker, M.: Certifying an Automated Code Generator Using Formal Tools : Preliminary Experiments in the GeneAuto Project. In: ERTS (2008)
23. Jonsson, U.T.: A lecture on the S-procedure (2001)
24. Moy, Y.: Union and cast in deductive verification
25. Muñoz, c., Narkawicz, A.: Formalization of an efficient representation of Bernstein polynomials and applications to global optimization. J. of Automated Reasoning (2011)
26. Owre, S., Rushby, J.M., , Shankar, N.: PVS: A prototype verification system. In: CADE. LNAI, vol. 607, pp. 748–752. Springer (Jun 1992)
27. Owre, S., Shankar, N.: Theory interpretations in PVS. Tech. Rep. SRI-CSL-01-01, Computer Science Laboratory, SRI International (April 2001)
28. Owre, S., Shankar, N., Rushby, J.M., J.Stringer-Calvert, D.W.: PVS Language Reference. Computer Science Laboratory, SRI International (Sep 1999)