FORMAL VERIFICATION OF CONTROL SOFTWARE

A Thesis
Presented to
The Academic Faculty

by

Romain Jobredeaux

In Partial Fulfillment
of the Requirements for the Degree
Doctor of Philosophy in the
School of Aerospace Engineering

Georgia Institute of Technology
August 2015

Copyright © 2015 by Romain Jobredeaux

FORMAL VERIFICATION OF CONTROL SOFTWARE

Approved by:

Dr. Éric Féron, Advisor
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Pierre-Loïc Garoche
Département de Traitement de
l'Information et Modélisation
ONERA, France

Dr. Alwyn Goodloe
Safety Critical Avionics Systems Branch
NASA Langley Research Center

Dr. Marcus Holzinger
School of Aerospace Engineering
Georgia Institute of Technology

Dr. Eric Johnson
School of Aerospace Engineering
Georgia Institute of Technology

Date Approved: April 3rd, 2015

À Mamé.

ACKNOWLEDGEMENTS

I would like to take a moment to thank all the peope who have made these past 5 years an amazing adventure. First, I would like to thank my thesis commitee members for agreeing to be a part of this journey. Éric Féron, my advisor, who unknowingly set me free from a tyrannic supervisor, and very knowingly set me, naive Master student that I was, on the path to become a doctor. You consistently insisted that I was worth something, and I am so grateful for all your mentoring and advice. Alwyn Goodloe, the first U.S. person I met who not only could place my home town on a map, but even knew about its iconic cathedral! You played no small part in my joining the PhD program, thank you for that, for teaching me a thing or two about history, and for your precious advice. Pierre-Loïc Garoche, who, out of the blue, offered me to spend a summer internship with him France. Your guidance gave me purpose and reassured me at times when I needed it the most. Thank you.

My thanks also go to Heber H., and Sam O., for guiding me through my early stabs at figuring out PVS.

I am very grateful to the National Aeronotics and Space Administration (grant NNX 08AE37A), the National Science Foundation (grants CPS Medium 1135955 (CrAVES) and CPS Synergy 1446758 (SORTIES)), and the Army Research Office (MURI W911NF-11-1-0046) for supporting this work.

Babak, your sometimes overwhelming morning links gave me my daily dose of laughter. Thank you for being always there for a chat, be it about quantum physics or kitchen appliances.

Benjamin M., your door was always open to me. This is something you really come to appreciate when you land in Paris and *forget* your suitcase in the metro. Thank you. Julien A., Romain M., I am so grateful for all the good times, all over the world (or almost).

Aurelie and Delphine, thank you for all the opportunities you gave me to visit Northern Europe! Marion, Bénédicte, Henri, Raphaëlle, thank you for the new years eves, the crazy times'up, the random Facebook conversations. Vivek and Etienne, you abandonned us for a sunnier land, but we definitely walked this PhD journey together :) Thank you! Benoit D., Johann M., thank you for an unforgettable road trip with Destiny, and for letting me abuse your hospitality. Mays, Zac, you took me in and made me part of a community. I will always be grateful for that.

My labmates know it is 6AM as I write those lines and, I hope, will forgive me for the lack of originality. Thank you Aude, Manu, Tim and Flo.

Julie D., thank you for welcoming me in the Shire and making my summer in California such a blast. I also want to thank the PhD students of DTIM at ONERA, and those of ENSEIIHT, for welcoming me to Toulouse as part of their group. In particular, many thanks to Pierre R. and Arnaud D. Alex and Kevin, you made the end of this PhD go by so fast, and feel like a breeze. That's quite a feat! Thank you, and a special thanks to Alex for being my go to Latex expert towards the end.

Mom, Dad, Clémence,Clémentine, Grégory, Didier, Nicolas, céline, my dear nieces, newphew and godson, thank you for your unwavering love and support.

Andrew, I can't imagine how I would have made it without you. Your love and support mean the world to me. Thank you.

Contents

DEDICATION . . . . **iii**

ACKNOWLEDGEMENTS . . . . **iv**

LIST OF FIGURES . . . . **ix**

SUMMARY . . . . **xii**

I INTRODUCTION **5**

1.1 The Credible Autocoding Framework 6

1.2 Scope of the Thesis and Related Work 10

II FINDING INVARIANT ELLIPSOIDS FOR DYNAMICAL SYSTEMS . . **15**

2.1 Ellipsoidal Sets and their Representation 15

2.1.1 The Direct Form, or P-Form 15

2.1.2 The Schur Form, or Q-Form 17

2.1.3 Equivalence Case 17

2.2 Stability of Discrete Dynamical Systems 18

2.2.1 Lyapunov Stability of Discrete Unforced Systems 18

2.2.2 Linear Unforced Systems 19

2.2.3 Bounded-Input, Bounded-State Stability of Linear Systems 19

2.2.4 Non-Linear Systems in the Absolute Stability Framework 21

2.3 Application: Extension of an Abstract Interpreter for Closed-Loop Stability Proofs 27

2.4 Conclusion 29

III A FORMAL FRAMEWORK TO EXPRESS CONTROL SEMANTICS AT THE LEVEL OF THE CODE **30**

3.1 Deductive Methods and Hoare Logic 30

3.2 Using Hoare Logic to Express a Simple Control Property on Code 33

3.3 A Formal Library for the Expression of Control Semantics on Code 35

3.3.1 ACSL Syntax and Semantics 36

3.3.2 Linear Algebra in ACSL . . . 40
3.3.3 Including Proof Elements . . . 44
3.4 Using the Library . . . 46
3.4.1 Key Constructs in the Annotation Process . . . 46
3.4.2 Lyapunov Stability of a Linear System . . . 46
3.4.3 Bounded-Input, Bounded-State Stability of a Linear Controller . . 50
3.4.4 Closed-Loop Stability of a Linear Controller Interacting with a Linear System . . . 51
3.4.5 Closed-Loop Stability of a Non-Linear Controller Interacting with a Linear System under the Absolute Stability Framework . . . 53
3.4.6 Application to Fault Detection and Gain-Scheduled Controllers . . 57
3.5 Conclusion and Perspectives . . . 59
IV A LIBRARY OF MACHINE-PROVEN LINEAR ALGEBRA AND CONTROLS RESULTS . . . 61
4.1 Interactive Theorem Provers . . . 61
4.1.1 The Prototype Verification System: Syntax and Semantics . . . 62
4.1.2 The PVS Proof Environment . . . 64
4.2 Linear Algebra Library in the Prototype Verification System (PVS) . . . 65
4.2.1 Types and Constructors . . . 65
4.2.2 Operators and Subtypes . . . 67
4.2.3 Some Useful Results . . . 67
4.3 Control Semantics in PVS . . . 68
4.3.1 Affine Ellipsoid Combination . . . 69
4.3.2 Ellipsoid Combination through S-Procedure Relaxation . . . 70
4.3.3 Ellipsoid Combinations Using Sector-Bounds . . . 72
4.4 Conclusion and Perspectives . . . 76
V A TOOL FOR THE AUTOMATIC VERIFICATION OF FUNCTIONAL CONTROL PROPERTIES ON CODE . . . 77
5.1 From C code to PVS Theorems . . . 79
5.1.1 Hoare Logic and Weakest Preconditions . . . 79

5.1.2 Converting Verification Conditions to PVS Theorems 80
5.1.3 Theory Interpretation 80
5.2 Generically Discharging the Proofs in PVS 83
5.3 The PVS-Ellipsoid Plugin to Frama-C 84
5.4 Checking Inclusion of the Propagated Ellipsoid, and Floating-Point Soundness 85
5.5 Application and Benchmarking 89
5.6 Conclusion and Perspectives 89

VI CONCLUSION **90**

Appendix A — ACSL LINEAR ALGEBRA LIBRARY **92**

Appendix B — EXEMPLE ANNOTATED PROGRAMS **100**

Appendix C — PVS LIBRARIES **135**

REFERENCES **167**

VITA **172**

List of Figures

1 Traditional V-shaped software development lifecycle. 7

2 The credible autocoding framework: using model-based development and formal methods to bring about shorter certification time and higher levels of assurance. 7

3 From traditional autocoding framework to credible autocoding framework: a change of paradigm. 8

4 Example of a full rank ellipsoid in $\mathbb{R}^3$. 16

5 Example of a direct form degenerate ellipsoid in $\mathbb{R}^3$. 16

6 Example of a Schur form degenerate ellipsoid in $\mathbb{R}^3$. 17

7 Luré's absolute stability problem setup. 21

8 Motivating example: a spring-mass system. 24

9 Illustration of the sector bound relationship between y_c (in the grey sector) and $Cx - y_d$. 26

10 Control flow graph for the system with a saturation. 28

11 Valid Hoare triples under the assignment rule. 32

12 Pseudo-code describing a simple linear system. 34

13 Simple linear system annotated with valid loop invariant. 34

14 Fully annotated linear system. 35

15 Writing the Hoare triple of (33) in ACSL. 36

16 Writing the loop invariant of (34) in ACSL 37

17 Using named behaviors to describe the semantics of the saturation function. 38

18 Peano axioms for natural numbers in ACSL. 39

19 Using ghost code to express a bound on the number of times a loop is run. . 39

20 Vector and matrix types in ACSL. 41

21 Accessor functions in ACSL. 41

22 Axiomatization of accessor functions for the constructor `mat_of_1x2_scalar` 42

23 Main matrix operators in ACSL. 42

24 Axiomatization of element accessor for the matrix addition function. 43

25 Axiomatization of dimension accessors for matrix addition function. 43

26 Axiomatizing the unspecified nature of matrix addition when dimensions are incompatible. 44

27 Axiomatization of predicate `in_ellipsoidQ` 45

28 Example of the ACSL extension for proof strategies. 46

29 Annotation asserting that a vector is in an ellipsoid. 46

30 Example of an ACSL annotation expressing an ellipsoidal invariant. 47

31 Example of a local ACSL contract. 47

32 Annotated program describing the linear update of state variables (Part 1). . 48

33 Annotated program describing the linear update of state variables (Part 2). . 49

34 Program step using the S-procedure to obtain a relaxed combination of 2 ellipsoids. 52

35 Template of the update function with added plant semantics in ghost code. . 54

36 Relevant part of annotated code for closed loop stability in the absolute stability problem (Part 1). 57

37 Relevant part of annotated code for closed loop stability in the absolute stability problem (Part 2). 58

38 A typical PVS proof sequent. 65

39 Block matrix definition in PVS. 66

40 Definition of `in_ellipsoid_Q?` in PVS. 69

41 Theorem describing the transformation of an ellipsoid by a linear map in PVS. 69

42 PVS theorem describing how two ellipsoids can be combined with proper multipliers. 71

43 PVS Lemma for bound extraction on linear combination of variables in an ellipsoid. 72

44 PVS lemma expressing the sector-bound on a saturation. 73

45 General view of the automated process. 78

46 Template of the generated loop update function. 78

47 Typical example of an ACSL Hoare Triple. 81

48 Excerpt of the PVS translation of the triple shown in Figure 47. 81

49 ACSL axiom for the row-size of a 2x2 matrix. 82

50 ACSL axiom in Figure 49 translated to PVS. 82

51 The PVS counterpart to ACSL's `mat_add`, in the purpose of carrying out theory interpretation. 83

52 The `ellipsoid_general` theorem in PVS. 84

53 Local ACSL contract before floating-point extension. 86

54 Local ACSL contract after floating-point extension. 86

55 Final contract in the program. 87

SUMMARY

In a context of heightened requirements for safety-critical embedded systems and ever-increasing costs of verification and validation, this research proposes to advance the state of formal analysis for control software. Formal methods are a field of computer science that uses mathematical techniques and formalisms to rigorously analyze the behavior of programs. This research develops a framework and tools to express and prove high level properties of control law implementations. One goal is to bridge the gap between control theory and computer science. An annotation language is extended with symbols and axioms to describe control-related concepts at the code level. Libraries of theorems, along with their proofs, are developed to enable an interactive proof assistant to verify control-related properties. Through integration in a prototype tool, the process of verification is made automatic, and applied to several example systems.

Nomenclature

Symbol	**Description**
$\mathbb{B}$	The set of booleans.
$\vee$	Logical OR.
$\wedge$	Logical AND.
$\mathbb{N}$	The set of natural numbers.
$\mathbb{N}^*$	The set of positive natural numbers.
$\mathbb{F}$	The set of floating-point numbers.
$\mathbb{R}$	The set of real numbers.
$\mathbb{R}^+$	The set of non-negative real numbers.
$\mathbb{R}^{+*}$	The set of positive real numbers.
$\mathbb{R}^n$	The set of real vectors of size n.
$\mathbb{R}^{n\times m}$	The set of real matrices of size $n \times m$.
$\preceq$	Assuming $A, B \in \mathbb{R}^{n\times n}$, $A \succeq B \iff \forall x \in \mathbb{R}^n : x^\mathrm{T}(A - B)x \leq 0$.
$\prec$	Assuming $A, B \in \mathbb{R}^{n\times n}$, $A \succ B \iff \forall x \in \mathbb{R}^n : x \neq 0 \Rightarrow x^\mathrm{T}(A - B)x < 0$.
$\succeq$	Assuming $A, B \in \mathbb{R}^{n\times n}$, $A \succeq B \iff \forall x \in \mathbb{R}^n : x^\mathrm{T}(A - B)x \geq 0$.
$\succ$	Assuming $A, B \in \mathbb{R}^{n\times n}$, $A \succ B \iff \forall x \in \mathbb{R}^n : x \neq 0 \Rightarrow x^\mathrm{T}(A - B)x > 0$.
$\mathcal{E}_P$	Assuming $P \in \mathbb{R}^{n\times n}$ and $P \succeq 0$, the set $\left\{x \in \mathbb{R}^n \,\middle|\, x^\mathrm{T}Px \leq 1\right\}$.
$\mathcal{E}_{P,\lambda}$	Assuming $P \in \mathbb{R}^{n\times n}, P \succeq 0$ and $\lambda > 0$, the set $\left\{x \in \mathbb{R}^n \,\middle|\, x^\mathrm{T}Px \leq \lambda\right\}$.
$\mathcal{G}_Q$	Assuming $Q \in \mathbb{R}^{n\times n}$ and $Q \succeq 0$, the set $\left\{x \in \mathbb{R}^n \,\middle\|\, \begin{bmatrix} 1 & x^\mathrm{T} \\ x & Q \end{bmatrix} \succeq 0\right\}$.

$\mathcal{G}_{Q,\lambda}$ — Assuming $Q \in \mathbb{R}^{n\times n}, Q \geq 0$ and $\lambda > 0$, the set $\left\{ x \in \mathbb{R}^n \middle| \begin{bmatrix} \lambda & x^{\mathrm{T}} \\ x & Q \end{bmatrix} \geq 0 \right\}$.

ACSL

`requires` — Introduces a precondition in a local contract or a function contract.

`ensures` — Introduces a postcondition in a local contract or a function contract.

`behavior` — Introduces a named contract and potential assumptions under which the contract should hold.

`assumes` — Within a behavior, introduces an assumption under which the contract is expressed to hold.

`\result` — Within a function contract, refers to the output of the function.

`axiomatic` — Introduces a set of logic definitions and axioms.

`type` — Introduces a new logic type, within an axiomatic.

`logic` — Introduces a new logic constant, or function, within an axiomatic.

`axiom` — Introduces a property that is assumed always true, within an axiomatic.

`ghost` — Ghost code: has the same syntax as C code, but is not executed. Used to introduce variables in the annotations without affecting the code.

`PROOF_TACTIC` — Within a contract, hints at the proof strategy that can be used to discharge it.

Linear Algebra in ACSL

`vect_of_n_scalar`	A function of n arguments that returns a vector. For example, `vect_of_2_scalar(a,b)` = $\begin{bmatrix} a \\ b \end{bmatrix}$.
`mat_of_nxm_scalar`	A function of $n \times m$ arguments that returns a matrix. For example, `mat_of_2x2_scalar(a,b,c,d)`= $\begin{bmatrix} a & b \\ c & d \end{bmatrix}$.
`mat_select`	Accessor function. Given a matrix $A = (a_{ij})_{1 \leq i,j \leq n}$, `mat_select (A,i,j)`= a_{ij}.
`mat_row`	Returns the row-size of a matrix.
`mat_col`	Returns the column-size of a matrix.
`mat_mult`	Matrix multiplication. If $A \in \mathbb{R}^{n \times m}$ and $B \in \mathbb{R}^{m \times p}$, `mat_mult( A,B)` = AB.
`block_m`	A 2×2 block matrix. Given 4 matrices $A \in \mathbb{R}^{n_1 \times m_1}, B \in \mathbb{R}^{n_1 \times m_2}, C \in \mathbb{R}^{n_2 \times m_1}, D \in \mathbb{R}^{n_2 \times m_2}$, `block_m(A,B,C,D)` = $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$.
`in_ellipsoidQ`	Ellipsoid predicate. Given a matrix $Q \in \mathbb{R}^{n \times n} \succeq 0$ and a vector $x \in \mathbb{R}^n$, `in_ellipsoidQ(Q,x)` $\iff x \in \mathcal{E}_Q$.

PVS

`THEORY`	The basic unit of specification in PVS. Contains definitions, axioms, lemmas and theorems.
`LEMMA` or `THEOREM`	A property within a theory. Must be proven.
`AXIOM`	A property within a theory, that is assumed to be always true.

Acronyms

ACSL	ANSI/C Specification Language
CPS	Cyber-Physical System
PVS	Prototype Verification System

TCC	Type-Correctness Conditions
V&V	Verification and Validation

Chapter I

INTRODUCTION

Cyber-Physical Systems (CPS) are ubiquitous. They are systems that interact physically with their environment, and are controlled, in part, by computers. They are called safety-critical when their failure entails catastrophic consequences in terms of loss of life or significant money amounts. The safety requirements for the computing part of such systems are very high, and the increasing complexity of CPS has led to a considerable increase in the costs of their Verification and Validation (V&V). The matter is made all the more urgent that, as the Federal Aviation Administration (FAA) and other regulation authorities work on defining the proper frame to open the airspace to Unmanned Aerial Systems (UAS), a major market is about to bloom and will benefit from automated and simple tools to facilitate product certification. In the medical world as well, examples abound of software failures causing catastrophic damage [1, 2], and regulations are bound to become tighter [3, 4]. This research proposes to advance the state of formal analysis techniques for these systems and, in particular, for control systems.

The current certification process mostly involves a significant amount of simulations, testing and reports, both to verify the input to output behavior of all implemented functions (unit tests) and to validate the high level design choices made early in the development process (integration tests). Formal methods offer to use rigorous mathematical techniques to prove properties of code. They are well-suited for safety-critical systems since they provide very high levels of assurance. They are a powerful tool to automatically prove the correctness of code [5, 6, 7].

However, the computational part of CPS can ideally have a complex numerical core and logic structure, making it difficult, if not impossible, for off-the-shelf formal analysis tools to

extract useful information beyond divide-by-zero errors and other low-level, non-functional properties. One key idea this research is based upon, is that the domain-specific knowledge that was used to design a CPS can prove very useful in the anlysis of its code.

This work proposes to develop libraries and tools to leverage domain-specific knowledge in the verification of domain-specific code. In particular, focus is given to proving stability and performance properties of control systems, at the level of the code.

1.1 The Credible Autocoding Framework

The credible autocoding framework is proposed as an alternative way of developing safety-critical software. The current model for the development of such certifiable software is process-based, that is, it relies on heavy documentation describing the various steps that were taken to ensure compliance with the requirements. It is based on the traditional V-cycle shown in Figure 1. High-level requirements are established, and validation activities are described. They will need to be carried out once the final product has reached the other end of the cycle. The high level requirements are then refined into a lower level description of the software product, which includes specifications for each and every function required in the final product. At this stage, unit tests are developed, to be carried out after the development phase to verify the individual functionalities of each code function. The "bottom" of the V-cycle is reached when the development phase is carried out. They are followed by the V&V activities documented during design. One major drawback of this approach is the important time separation between design and V&V. When an issue is exposed by a certain validation simulation, the whole process must be worked through again: new high-level requirements leading to new low-level specifications and new code. In addition, as systems become more and more complex, and requirements from the certification authorities are more and more stringent, the costs involved in the right part of the V-cycle become increasingly prohibitive.

The credible autocoding framework is presented in Figure 2. It proposes to use a combination of model-based development and formal methods, in order to automatically generate

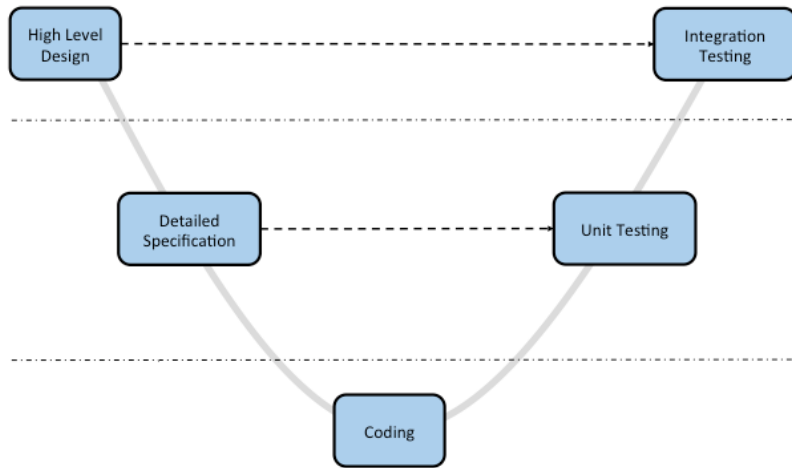

Figure 1: Traditional V-shaped software development lifecycle.

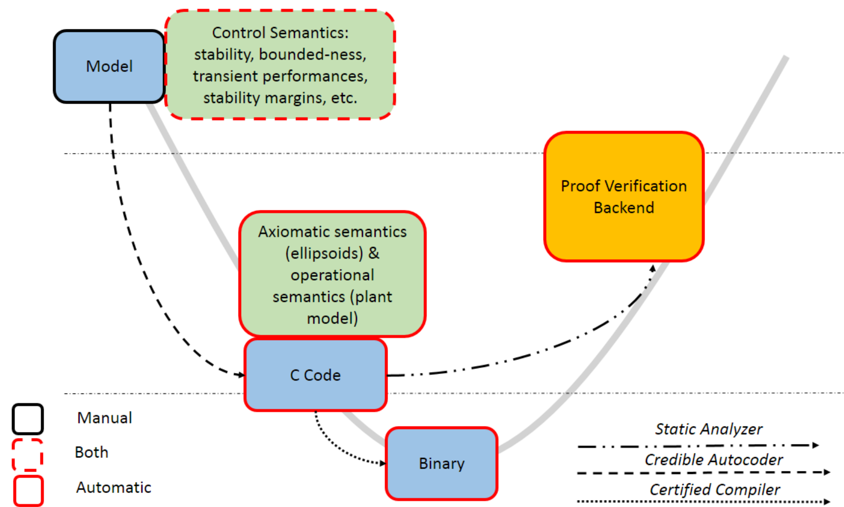

Figure 2: The credible autocoding framework: using model-based development and formal methods to bring about faster certification time and higher levels of assurance.

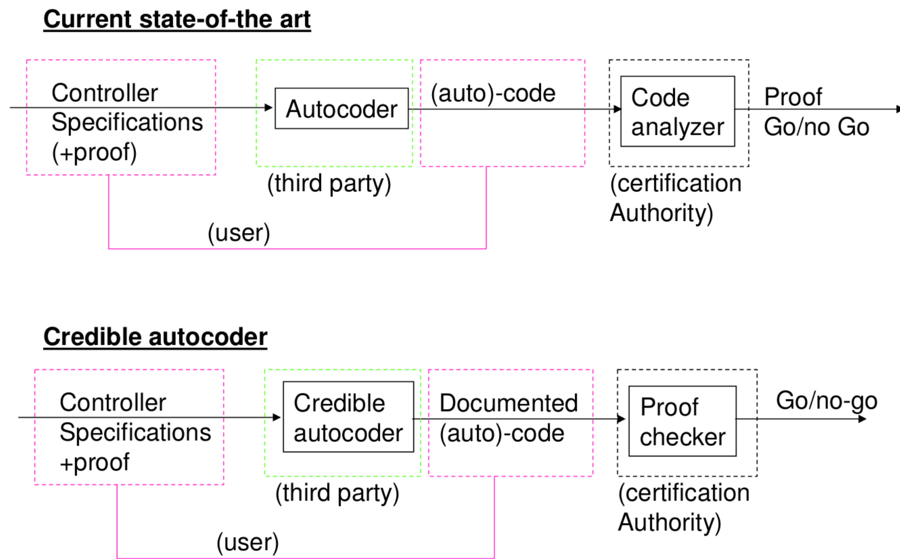

Figure 3: From traditional autocoding framework to credible autocoding framework: a change of paradigm.

the code implementing a safety-critical system, as well as a formal documentation describing its behavior. The paradigm shift introduced by this model is summarized in Figure 3: because the software now comes with a formal description of its safety features, part of the V&V process can be made automatic: a proof checker can be developed to provide a certificate of correctness for the annotations. The work presented in this manuscript is centered around the development of such a proof checker. Credible autocoding leverages domain-specific knowledge in order to provide automation wherever possible for the development and certification of safety-critical software. It promises to save time and money while offering higher levels of assurance.

While control theorists are familiar with the notion of open and closed-loop stability, and have developed various means to study it – e.g. the Routh-Hurwitz criterion, root locus analyses, or the Nyquist stability criterion –, its evaluation or formal verification at code or system level remains an open question.

At the computer science level, these control properties are rarely known and hard to

express or evaluate in the later stages of system development. In other words, these meaningful requirements of the system tend to disappear when defining the software requirements. This absence precludes a precise analysis of the interaction between the physical arithmetic equations characterizing the dynamic of the plant and the actual implementation of the controller in a computer, with all its associated limitations: bounded memory, real time issues, floating point computations, etc.

Addressing these questions, i.e., evaluating control level properties at code level, would allow for a clearer understanding of the behavior of the final system and could avoid detecting issues too late in the development process.

An interesting outcome of this kind of analyses is the possibility to exhaustively evaluate control-level properties, as usually evaluated through simulation on hybrid systems.

Research Questions The questions this research will endeavor to answer are as follows:

- How can domain-specific knowledge in control theory improve the state of V&V techniques for safety-critical CPS?
- How can domain-specific, control theoretic properties be proven at the level of the code?
- How can the gap between the high-level proof of these properties and their code-level counterpart, introduced by implementation artefacts, be handled?

Contributions We argue that proper domain-knowledge, combined with formal methods, make it possible to automatically verify control properties on source code, in a sound manner. Specifically, we demonstrate the feasability of a verification backend for the credible autocoding framework, in the case of stability properties for quasi-linear systems. We draw from the fields of computer science and control theory, and show how their collaboration can advance the state of verification and validation technology. The main contributions are as follows.

1. This work provides heuristics for the first extension of an abstract interpretation tool to support closed-loop stability proofs for linear systems interacting with saturated linear controllers (Section 2.3).

2. The first annotation language extension is developed that supports the expression of:

   - linear algebra constructs;
   - control properties;
   - proof information;

   at the level of C code (Chapter 3).

3. This work initiated the construction, and significantly contributed to the first library of machine-proven linear algebra and control theory results (Chapter 4). Specifically, fundamental results on matrices, quadratic forms defined by them, and ellipsoidal sets, are established within a theorem prover.

4. The first tool that supports the automatic verification of stability properties from annotated code was developed as part of this work (Chapter 5).

5. An approach to handling discrepancies beween model and code, introduced by floating-point computations, is presented. (Chapter 5).

1.2 Scope of the Thesis and Related Work

This thesis does not seek to create new mathematical results in the field of control theory. Neither does it claim to develop new static analysis methods in computer science. Rather, the combination of results from these two fields which, in the past, have seen little communication or collaboration, forms a meaningful contribution to the state-of-the-art in verification and validation technologies. This work seeks to obtain *sound* measures of safety on software with the help of control theory: rather than focus on complex classes of systems or controllers, we instead make sure the properties that we prove are assuredly valid, robust

to implementation artefacts, and come with a trace, a documented proof which constitutes a safety case.

Although, to the author's knowledge, the techniques and tools presented in this dissertation are novel and rather unique. This research belongs to the field of formal methods and cyber-physical systems verification, whose related literature is now reviewed.

Daniel Jackson [8] led early efforts to expose the need for trustworthy certificates on safety-critical software. Initial attempts at formalizing control semantics in Simulink can be attributed to Ursula Martin in [9]. Although limited to frequency responses of Single-Input, Single-Output (SISO) systems in Simulink, this is the first suggestion of combining formal tools from computer science with known results from control theory.

There are many modern techniques to analyze software. Model checking is one that endeavors to automatically prove safety properties of finite-state systems [10]. It is widely used in industry as recent developments have made boolean satisfiability problem solvers (SAT solvers) and satisfiability modulo theories problem solvers (SMT solvers) much more efficient and scalable [11, 12]. Unfortunately, control software remains subject to an explosion of the state-space, making the use of these techniques difficult for this research. In the process of verifying quadratic properties with SMT solvers, the work in [13] shows promise, although it only solves the satisfiability problem on reals up to an error δ.

Abstract interpretation has proven to be a powerful, scalable technique to prove low-level properties of code. It was successfully applied on the Airbus A380 code to prove the absence of runtime errors caused by buffer overflow or index out-of-bound failures [14]. The choice of a proper abstract domain and good widening/narrowing heuristics remains a difficult one. In particular, there is no good lattice structure on the domain of ellipsoids, crucial to many results of control theory. Finally, some control systems require highly non-linear Lyapunov functions in their proof of stability, involving transcendental functions that no current domain encompasses, to our knowledge. Feret's work [15, 16] is a practical approach to the problem of extracting quadratic invariants in an abstract interpretation framework.

Its goal is to address the need by Astrée [17] to handle the linear filters present in Airbus real time software. Work by Monniaux [18] addressed the same class of systems but not on actual code. Both of these efforts address a strict subset of the systems we consider in this work.

On a more abstract side, one can find work that targets similar properties. Roozbehani, Feron and Megretski in [19], and Cousot in [20], introduced a Lagrangian relaxation approach applied to program termination analysis. One can also cite the works of Adjé, Gaubert and Goubault [21], and Gawlitza and Seidl [22] on policy iterations and non-linear forms. The latter two aim at replacing a Kleene based fixpoint computation by a symbolic reasoning based on semi-definite programming. They are more inspired by abstract results leading to the analysis. The work in [21, 23] even cites the existence of Lyapunov based invariant as a prerequisite for the method. These works address the analysis of a wider class of systems than this thesis. However, they do not provide an automatic framework to carry out the safety analysis: they assume to be given templates. One of the objectives of this work is precisely to provide automation through domain-specific knowledge. Finally, these works do not address floating point issues.

Our work should be considered as an in-between solution in terms of abstraction. It takes ideas from control theory results but targets the analysis of specific, operational systems. Furthermore, it addresses floating point errors as well as the validity analysis of the obtained invariants.

To our knowledge, apart from the work in [24], which was the starting point for this dissertation, no other research endeavor addresses the issue of proving in the C code the high-level correctness properties of control systems such as closed loop stability.

Regarding the prover aspect of our framework, there already exists tools that support the proof of properties in real arithmetic or real linear algebra. 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 [25]. Generic design patterns were proposed to define algebraic structures [26]. Formalization and instrumentation of Euclidean spaces also exist for Isabelle/HOL [27]. Automatic decision procedures exist for floating point arithmetics, such as Gappa [28]. A formalization of multivariate Bernstein polynomials for the Prototype Verification System (PVS) was presented in [29]. None of these recent extensions of theorem provers are able to deal with the properties of interest of this research. In addition, there was no equivalent effort in the theorem prover used in this work, PVS.

Finally, there is a a very large body of work focusing on hybrid systems. It is difficult to summarize these analyses in a few words. It can however be said that they usually (1) address systems of a somewhat different nature with a central continuous behavior described by differential equations and few discrete events (for instance a bouncing ball or an overflowing water tank) whereas controllers perform discrete transitions on a periodical basis, and (2) focus on bounded time properties rather than invariant generation. A lot of the early work on hybrid systems involves classes of continuous dynamics which are not expressive enough for our purposes, like linear hybrid automata in HyTech [30]. More recent tools like SpaceEx [31] handle systems that are closer to those under consideration in this research, however they remain focused on proving finite-time properties rather than invariants (indeed, discovering invariants on hybrid systems is, more often than not, undecidable). Such reachability analysis tools thus do not achieve the objectives of this work. Guaranteed simulation strategies also enter in the category of powerful formal tools, but are not suitable to the generation of invariant properties. Controller synthesis from temporal logic formulas[32] can provide guarantees of controller behavior when it succeeds. KeYmaera is a powerful software tool for model level verification of properties for hybrid systems [33]. It combines automated strategies and heuristics to prove invariants on CPS. However, these last two research directions focus on the model level and are not necessarily sound. This research is focused on sound verification at code level, in order to provide

guarantees that hold in spite of implementation artifacts.

Chapter II

FINDING INVARIANT ELLIPSOIDS FOR DYNAMICAL SYSTEMS

The credible autocoding process relies on the assumption that a control engineer designing a control law can provide mathematical evidence of its stability. This chapter introduces some of the fundamental control theory results used for the analysis of the type of systems under consideration in this disseration. Ellipsoidal sets are first introduced, as they form the basis of the subsequent proof work. Then, the concept of Lyapunov stability and bounded-input, bounded-state stability is introduced for a variety of systems. Stability analysis problems are formulated as matrix inequalities, whose solutions either provide a Lyapunov function or an invariant ellipsoidal set for the state variables of a model. Linear Matrix Inequalities (LMI), in which unknowns only appear linearly, can be solved efficiently using traditional SDP solvers [34]. For stability analysis problems that cannot be formulated as a LMI, heuristics are described that offer a possible approach, for lack of a systematic solution.

2.1 Ellipsoidal Sets and their Representation

Throughout this work, ellipsoidal sets, or ellipsoids, will be used to characterize various properties in control system models and implementations. Two different representations are introduced, their differences and case of equivalence discussed and proven in this section.

2.1.1 The Direct Form, or P-Form

An ellipsoid is typically described using a positive-definite or positive-semidefinite quadratic form. Given a positive-semidefinite matrix $P \in \mathbb{R}^{n\times n}$, the ellipsoid $\mathcal{E}_P$ is given by the set of all vectors $x \in \mathbb{R}^n$ such that $x^{\mathrm{T}}Px \leq 1$. Equivalently, the notation $\mathcal{E}_{P,\lambda} = \{x \in \mathbb{R}^n : x^{\mathrm{T}}Px \leq \lambda\}$, where $\lambda > 0$, will be used at times. This representation will be referred to as *direct form* or

P-form. An example of ellipsoid in $\mathbb{R}^3$ is given in Figure 4, where P is full rank.

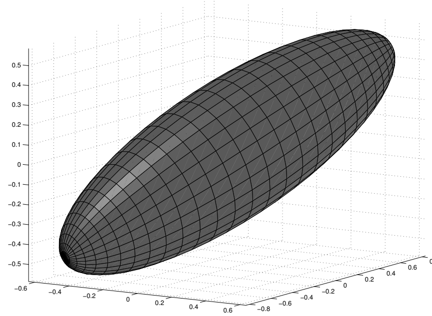

Figure 4: Example of a full rank ellipsoid in $\mathbb{R}^3$.

If P is not full rank, the ellipsoid is said to be degenerate. Indeed, any vector in the kernel of P belongs to the ellipsoid, which means that in some directions, it extends to infinity: this type of degenerate ellipsoid is actually a lower dimension ellipsoid extended cylindrically in the directions of the kernel of P. A three-dimensional example is shown in Figure 5.

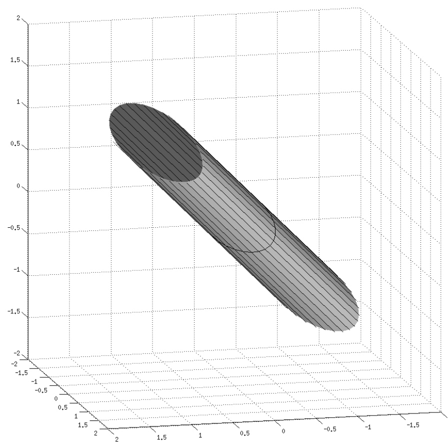

Figure 5: Example of a direct form degenerate ellipsoid in $\mathbb{R}^3$.

2.1.2 The Schur Form, or Q-Form

Another possible representation of ellipsoids, given a positive-semidefinite matrix $Q \in \mathbb{R}^{n \times n}$, is denoted as $\mathcal{G}_Q$. $\mathcal{G}_Q$ is the set of all vectors $x \in \mathbb{R}^n$ such that the matrix $\begin{bmatrix} 1 & x^{\mathrm{T}} \\ x & Q \end{bmatrix}$ is positive-semidefinite. This description will be referred to as the *Schur form*, or *Q-form* of an ellipsoid. When Q is full rank, it will be shown that this definition is no more, and no less expressive than the full rank direct form.

Degenerate ellipsoids under this representation, however, differ from the P-form. Indeed, if Q is not full rank, $\mathcal{G}_Q$ represents an ellipsoid of lower dimension, in a given vector subspace of $\mathbb{R}^n$. Figure 6 shows an example of the kind of ellipsoid a rank two Q in $\mathbb{R}^3$ yields. Effectively, if Q has rank r, $x \in \mathcal{G}_Q$ encodes two pieces of information: that some projection of x on a subspace of dimension r belongs to a full-rank ellipsoid, and that there exists $n - r$ linear relationships between the elements of x.

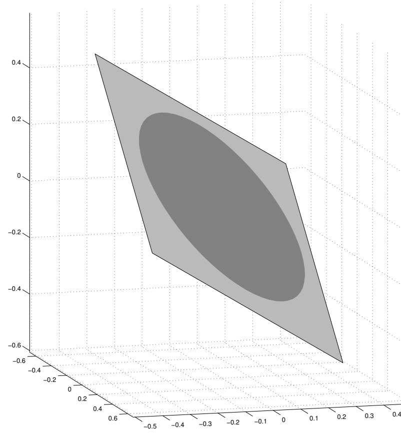

Figure 6: Example of a Schur form degenerate ellipsoid in $\mathbb{R}^3$.

2.1.3 Equivalence Case

Whenever $P > 0$, the following holds:

$$\mathcal{E}_P = \mathcal{G}_{P^{-1}}. \tag{1}$$

The proof of this statement relies on a well-known result based on Schur complement, given here for the sake of completeness, and for which a proof can be found in [35]:

Lemma 1 *Let* $X = \begin{bmatrix} A & B^{\mathrm{T}} \\ B & C \end{bmatrix}$ *be a* $(n+m)\times(n+m)$ *symmetric matrix. Assume* $C > 0$. *Then:*

$$X \geq 0 \iff A - B^{\mathrm{T}}C^{-1}B \geq 0. \tag{2}$$

The proof of the equivalence between the two forms follows.

Proof 1 $\Longrightarrow$ *Assume* $x \in \mathcal{E}_P$. *Apply Lemma 1 (in the* $\Longleftarrow$ *direction) to* $A = 1$, $B = x$, *and* $C = P^{-1}$ *to conclude.*

$\Longleftarrow$ *Assume* $x \in \mathcal{G}_{P^{-1}}$. *Apply Lemma 1 (in the* $\Longrightarrow$ *direction) to* $A = 1$, $B = x$, *and* $C = P^{-1}$ *to conclude.* □

2.2 *Stability of Discrete Dynamical Systems*

2.2.1 Lyapunov Stability of Discrete Unforced Systems

A dynamical system defined by $x_{k+1} = f(x_k), x(0) = x_0$, where $f : \mathbb{R}^n \to \mathbb{R}^n$ is continuous and $f(0) = 0$, is Lyapunov stable if and only if

$$\forall \epsilon > 0, \exists \delta > 0, \|x_0\| \leq \delta \implies \forall k \in \mathbb{N}, \|x_k\| \leq \epsilon.$$

This property is equivalent to the existence of a function $V : \mathbb{R}^n \to \mathbb{R}$, such that:

- $V(0) = 0$;
- $\forall x \in \mathbb{R}^n : x \neq 0 \implies V(x) > 0$;
- $\forall x \in \mathbb{R}^n : V(f(x)) \leq V(x)$.

This fundamental result is the discrete equivalent of Lyapunov's seminal result on continuous differential equations [36]. A thorough treatment of the topic can be found in [37]. Within this dissertation, we will focus our attention to *quadratic* Lyapunov functions, that is, functions of the form $V(x) = x^{\mathrm{T}}Px$, where $P \in \mathbb{R}^{n\times n}$ is a positive definite matrix. Note that these functions verify the first two items of the above list by construction.

2.2.2 Linear Unforced Systems

Unforced linear systems are the most well-studied and well-understood class of dynamical systems. Locally, they can model a great number of engineering systems dynamics very acurately. The dynamic equation for such systems is simply given by

$$x_{k+1} = Ax_k, x(0) = x_0. \tag{3}$$

where $A \in \mathbb{R}^{n \times n}$, with n a positive integer characterizing the dimension of the state of the system. When studying their stability using a Lyapunov-based approach, one looks for a matrix $P \succ 0$ such that

$$\forall x \in \mathbb{R}^n, (Ax)^{\mathrm{T}} P (Ax) \leq x^{\mathrm{T}} P x,$$

or, equivalently

$$A^{\mathrm{T}} P A - P \preceq 0.$$

This last equation is called the Lyapunov equation and is a special case of a Linear Matrix Inequality (LMI) [34]. Efficient solvers for such inequalities exist, like SeDuMi [38]. If a solution exists, then one can say that:

$$\forall x \in \mathbb{R}^n, x^{\mathrm{T}} P x \leq x_0^{\mathrm{T}} P x_0,$$

which is equivalent to $\forall x \in \mathbb{R}^n, x \in \mathcal{E}_{P'}$, where $P' = P/(x_0^{\mathrm{T}} P x_0)$. Thus, solving the LMI naturally provides an invariant ellipsoidal set for the system.

2.2.3 Bounded-Input, Bounded-State Stability of Linear Systems

Input-output linear systems are modeled with the following set of equations:

$$x_{k+1} = Ax_k + Bu_k, x(0) = x_0, \tag{4}$$

$$y_k = Cx_k + Du_k, \tag{5}$$

where $u_k \in \mathbb{R}^m$ represents external *inputs*, $y_k \in \mathbb{R}^l$ contains the *outputs*, and A, B, C, D, are matrices of commensurate dimensions characterizing the dynamics of the system.

One property of interest is whether the state variables of such a system remain bounded under the assumption that its input is. If this result can be established, it naturally follows from (5) that the system is Bounded Input, Bounded Output (BIBO)-stable. Control theory dictates that, regardless of the actual value of a global bound on u_k, the dynamical system in (4), (5) is bounded-input, bounded-state stable if and only if there exists a matrix $P \succ 0$ such that

$$A^{\mathrm{T}} P A - P \prec 0.$$

This LMI can be solved efficiently. However, in the developments that follow, we will look for concrete, stable ellipsoids for all variables, given an actual bound on the input u_k, $u_k^{\mathrm{T}} u_k \leq U$. To achieve this, one can look for a matrix $P \succ 0$ such that, for all $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$:

$$x^{\mathrm{T}} P x \leq 1 \wedge u^{\mathrm{T}} u \leq U \implies (Ax + Bu)^{\mathrm{T}} P (Ax + Bu) \leq 1. \tag{6}$$

Equation (6) can be rewritten as:

$$\begin{bmatrix} x \\ u \\ 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} P & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ u \\ 1 \end{bmatrix} \leq 0 \wedge \begin{bmatrix} x \\ u \\ 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} 0 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & -U \end{bmatrix} \begin{bmatrix} x \\ u \\ 1 \end{bmatrix} \leq 0 \implies \begin{bmatrix} x \\ u \\ 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} A^{\mathrm{T}} P A & A^{\mathrm{T}} P B & 0 \\ B^{\mathrm{T}} P A & B^{\mathrm{T}} P B & 0 \\ 0 & 0 & -1 \end{bmatrix} \begin{bmatrix} x \\ u \\ 1 \end{bmatrix} \leq 0. \tag{7}$$

The relaxation technique known as S-Procedure [39] provides a means to express (7) as a single inequality. Indeed, Equation (7) is implied by the existence of nonnegative real numbers λ and μ such that, for all $x \in \mathbb{R}^n$, $u \in \mathbb{R}^m$:

$$\begin{bmatrix} A^{\mathrm{T}} P A & A^{\mathrm{T}} P B & 0 \\ B^{\mathrm{T}} P A & B^{\mathrm{T}} P B & 0 \\ 0 & 0 & -1 \end{bmatrix} - \lambda \begin{bmatrix} P & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} - \mu \begin{bmatrix} 0 & 0 & 0 \\ 0 & I_m & 0 \\ 0 & 0 & -U \end{bmatrix} \leq 0,$$

which is equivalent to:

$$\begin{bmatrix} A^{\mathrm{T}} P A - \lambda P & A^{\mathrm{T}} P B & 0 \\ B^{\mathrm{T}} P A & B^{\mathrm{T}} P B - \mu I_m & 0 \\ 0 & 0 & -1 + \lambda + \mu U \end{bmatrix} \leq 0. \tag{8}$$

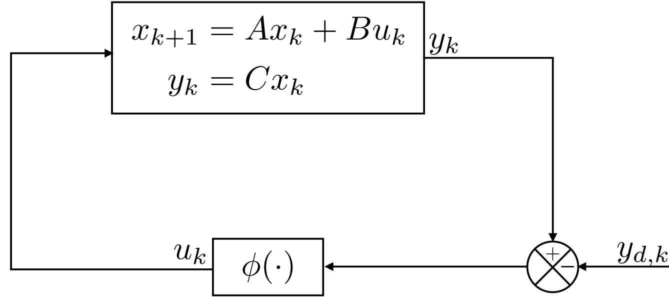

Figure 7: Luré's absolute stability problem setup.

Note that, because λ and P are both unknown, this matrix inequality is not linear. However, the bottom right corner of the matrix in (8) indicates that $0 \leq \lambda \leq 1$. A possible heuristic consists in iterating on potential values of λ, proceeding, for example, with a dichotomy in the $[0, 1]$ segment, and attempting to solve the resulting LMI.

2.2.4 Non-Linear Systems in the Absolute Stability Framework

The most complex dynamical systems considered in this work are the discrete-time version of those described in the absolute stability framework [40], as shown in Figure 7. They are defined by the following set of equations:

$$x_{k+1} = Ax_k + B\phi(Cx_k - y_{d,k}), x_0 = 0, \tag{9}$$

$$y_k = Cx_k. \tag{10}$$

where the vectors $x_k \in \mathbb{R}^n$, $y_{d,k} \in \mathbb{R}^l$ and $y_k \in \mathbb{R}^l$ are respectively the *state* of the system, its *inputs*, and its *outputs*. $\phi : \mathbb{R}^l \mapsto \mathbb{R}^m$ encapsulates the non-linearities of the system. A, B, C, and D are matrices of commensurate dimmensions. In general, ϕ can be time-dependent, however we will not consider this case here. The function ϕ is assumed to be sector-bounded, that is, there exist matrices $M_1, M_2 \in \mathbb{R}^{m\times l}$ such that:

$$\forall y \in \mathbb{R}^l : (\phi(y) - M_1 y)^{\mathrm{T}}(\phi(y) - M_2 y) \leq 0, \tag{11}$$

or, equivalently:

$$\forall y \in \mathbb{R}^l : \begin{bmatrix} y \\ \phi(y) \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} M_1^{\mathrm{T}} M_2 & -\frac{1}{2}(M_1^{\mathrm{T}} + M_2^{\mathrm{T}}) \\ -\frac{1}{2}(M_1 + M_2) & I_m \end{bmatrix} \begin{bmatrix} y \\ \phi(y) \end{bmatrix} \leq 0. \tag{12}$$

2.2.4.1 *Stability analysis with $y_{d,k} = 0$*

A LMI formulation of the stability analysis problem for the system in (9), (10), in the case $y_{d,k} = 0$, is provided here. Looking for a quadratic Lyapunov function $V(x) = x^{\mathrm{T}} P x$, the stability condition is

$$\begin{aligned} V(Ax_k + B\phi(Cx_k)) \leq V(x) \iff & x_k^{\mathrm{T}} A^{\mathrm{T}} P A x_k + \phi(Cx_k)^{\mathrm{T}} B^{\mathrm{T}} P B \phi(Cx_k) \\ & + 2x_k^{\mathrm{T}} A^{\mathrm{T}} P B \phi(Cx_k) - x_k^{\mathrm{T}} P x_k \leq 0. \end{aligned} \tag{13}$$

$$\iff \begin{bmatrix} x_k \\ \phi(Cx_k) \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} A^{\mathrm{T}} P A - P & A^{\mathrm{T}} P B \\ B^{\mathrm{T}} P A & B^{\mathrm{T}} P B \end{bmatrix} \begin{bmatrix} x_k \\ \phi(Cx_k) \end{bmatrix} \leq 0 \tag{14}$$

This condition must hold whenever, according to (12):

$$\begin{bmatrix} x_k \\ \phi(Cx_k) \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} C^{\mathrm{T}} M_1^{\mathrm{T}} M_2 C & -\frac{1}{2} C^{\mathrm{T}} (M_1^{\mathrm{T}} + M_2^{\mathrm{T}}) \\ -\frac{1}{2}(M_1 + M_2) C & I_m \end{bmatrix} \begin{bmatrix} x_k \\ \phi(Cx_k) \end{bmatrix} \leq 0. \tag{15}$$

Once again, the S-Procedure is used to relax this implication as follows. The system is Lyapunov stable if there exist a matrix $P > 0$ and a nonnegative coefficient μ such that:

$$\begin{bmatrix} A^{\mathrm{T}} P A - P & A^{\mathrm{T}} P B \\ B^{\mathrm{T}} P A & B^{\mathrm{T}} P B \end{bmatrix} - \mu \begin{bmatrix} C^{\mathrm{T}} M_1^{\mathrm{T}} M_2 C & -\frac{1}{2} C^{\mathrm{T}} (M_1^{\mathrm{T}} + M_2^{\mathrm{T}}) \\ -\frac{1}{2}(M_1 + M_2) C & I_m \end{bmatrix} \geq 0,$$

which is a LMI in P and the scalar variable μ.

2.2.4.2 *Searching for an invariant ellipsoid with bounded $y_{d,k}$*

A bounded, non-zero input $y_{d,k}$ is now assumed:

$$y_{d,k}^{\mathrm{T}} y_{d,k} \leq U. \tag{16}$$

In addition, tu support further developments in this work, we search for an invariant ellipsoid, that is, $P \succ 0$ such that, assuming

$$x_k^{\mathrm{T}} P x_k \leq 1, \tag{17}$$

then $x_{k+1}^{\mathrm{T}} P x_{k+1} \leq 1$, in other words:

$$\begin{bmatrix} x_k \\ \phi(Cx_k - y_{d,k}) \end{bmatrix}^{\mathrm{T}} \underbrace{\begin{bmatrix} A^{\mathrm{T}}PA - P & A^{\mathrm{T}}PB \\ B^{\mathrm{T}}PA & B^{\mathrm{T}}PB \end{bmatrix}}_{\mathcal{T}} \begin{bmatrix} x_k \\ \phi(Cx_k - y_{d,k}) \end{bmatrix} \leq 1. \tag{18}$$

Equation (15) can be rewritten, taking $y_{d,k}$ into account, as:

$$\begin{bmatrix} x_k \\ \phi(Cx_k - y_{d,k}) \\ y_{d,k} \end{bmatrix}^{\mathrm{T}} \underbrace{\begin{bmatrix} C^{\mathrm{T}}M_1^{\mathrm{T}}M_2C & -\frac{1}{2}C^{\mathrm{T}}(M_1^{\mathrm{T}} + M_2^{\mathrm{T}}) & C^{\mathrm{T}}M_1^{\mathrm{T}}M_2 \\ -\frac{1}{2}(M_1 + M_2)C & I_m & \frac{1}{2}(M_1 + M_2) \\ M_2^{\mathrm{T}}M_1C & \frac{1}{2}(M_1^{\mathrm{T}} + M_2^{\mathrm{T}}) & M_1^{\mathrm{T}}M_2 \end{bmatrix}}_{\mathcal{S}} \begin{bmatrix} x_k \\ \phi(Cx_k - y_{d,k}) \\ y_{d,k} \end{bmatrix} \leq 0. \tag{19}$$

A sufficient condition for the invariance of the ellipsoid defined by P is then (19) $\wedge$ (17) $\wedge$ (16) $\implies$ (18), which can once more be recast in a single matrix inequality: if one can find λ, μ, and ν, positive coefficients such that

$$\begin{bmatrix} \mathcal{T} & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & -1 \end{bmatrix} - \lambda \begin{bmatrix} P & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & -1 \end{bmatrix} - \mu \begin{bmatrix} \mathcal{S} & 0 \\ 0 & 0 \end{bmatrix} - \nu \begin{bmatrix} 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & I_l & 0 \\ 0 & 0 & 0 & -U \end{bmatrix} \leq 0, \tag{20}$$

then the ellipsoid defined by P is invariant. Once again, this inequality is not linear in the unknowns, and a heuristic similar as that described in Section 2.2.3 can be used.

2.2.4.3 A Heuristic in the Saturation Case

When the system is single-input, single-output (SISO), and the non-linearity is a saturation, without additional information, the best sector-bound choice is the $[0, 1]$ sector. In this section, we develop a heuristic on the example of a controlled spring-mass system, still in

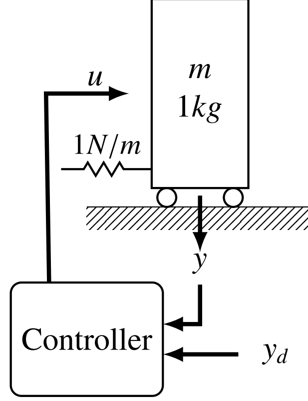

Figure 8: Motivating example: a spring-mass system.

the purpose of finding an invariant ellipsoid. Indeed, on the chosen example, the heuristic described in Section 2.2.4.2 fails.

System Model The spring-mass system in Figure 8 can be modeled with the continuous differential equation $\ddot{x} = -x + u$, where x is the displacement of the spring from its resting position, in meters, and u is the external force applied to the mass, in Newtons. A sensor observes the position of the mass, so that $y = x$. A 100 Hz first-order Euler discretized version of these dynamics is given, in state-space form, by:

$$x_{p,k+1} = A_p x_{p,k} + B_p u_k, \tag{21}$$

$$y_k = C_p x_k, \tag{22}$$

with

$$A_p = \begin{bmatrix} 1.00 & 0.01 \\ -0.01 & 1.00 \end{bmatrix}, B_p = \begin{bmatrix} 5 \cdot 10^{-5} \\ 0.01 \end{bmatrix}, \text{ and } C_p = \begin{bmatrix} 1 & 0 \end{bmatrix}. \tag{23}$$

Given a desired position y_d, with $|y_d| \leq 0.5$, one possible lead-lag controller is given here. It reacts to the saturated error $y_c = \mathbf{SAT}(y - y_d)$, where **SAT** is the unit saturation operator: $\mathbf{SAT}(x) = \max(\min(x, 1), -1)$. Its transfer function is given by:

$$u(s) = -128 \frac{s+1}{s+0.1s} \frac{s/5+1}{s/50+1} y_c(s).$$

A digital filter implementing a discretization of this controller, running at 100 Hz, can be obtained with the following state-space equations:

$$x_{c,k+1} = A_c x_k + B_c y_{c,k}, x_{c,0} = 0, \tag{24}$$

$$u_k = C_c x_k + D_c y_{c,k}, \tag{25}$$

where

$$A_c = \begin{bmatrix} .4990 & -0.05 \\ 0.01 & 1.0 \end{bmatrix}, B_c = \begin{bmatrix} 1 \\ 0 \end{bmatrix}, C_c = \begin{bmatrix} 564.48 & 0 \end{bmatrix}, \text{ and } D_c = -1280. \tag{26}$$

The interaction of the spring-mass system and this digital controller fit the systems description in Figure 7 with

$$\phi = \mathbf{SAT}, x_k = \begin{bmatrix} , x_{c,k} \\ x_{p,k} \end{bmatrix} A = \begin{bmatrix} A_c & 0 \\ B_p C_c & A_p \end{bmatrix}, B = \begin{bmatrix} B_c \\ B_p D_c \end{bmatrix}, \text{ and } C = \begin{bmatrix} 0 & C_p \end{bmatrix}. \tag{27}$$

Finding an Invariant Ellipsoid Assuming $x_k^{\mathrm{T}} P x_k \leq 1$, a bound on $|Cx_k|$ is given by $\gamma := \sqrt{CP^{-1}C^{\mathrm{T}}}$. Since $|y_{d,k}| \leq 0.5$, the constant $\tilde{\gamma} := \gamma + 0.5$ is an upper bound on $|Cx_k - y_{d,k}|$. Letting $y_{c,k} := \mathbf{SAT}(Cx_k - y_{d,k})$, the following sector bound holds:

$$\left(y_{c,k} - \frac{1}{\tilde{\gamma}}(Cx_k - y_{d,k}) \right) (y_{c,k} - (Cx_k - y_{d,k})) \leq 0. \tag{28}$$

Figure 9 illustrates the reason for this inequality. $y_{c,k} = \mathbf{SAT}(Cx_k - y_{d,k})$ is represented as a thick line. With the added bound $\tilde{\gamma}$ on $|Cx_k - y_{d,k}|$, one sees that $y_{c,k}$ necessarily lies between $Cx_k - y_{d,k}$ and $\frac{1}{\tilde{\gamma}}(Cx_k - y_{d,k})$. Then $y_{c,k} - \frac{1}{\tilde{\gamma}}(Cx_k - y_{d,k})$ and $y_{c,k} - (Cx_k - y_{d,k})$ must be of opposite signs, hence the inequality.

Thus, one looks for a matrix P such that

$$\sqrt{CP^{-1}C^{\mathrm{T}}} \leq \gamma, \tag{29}$$

and

$$\left(x_k^{\mathrm{T}} P x_k \leq 1 \wedge y_{d,k}^2 \leq 0.5^2 \wedge (28) \right) \implies x_{k+1}^{\mathrm{T}} P x_{k+1} \leq 1. \tag{30}$$

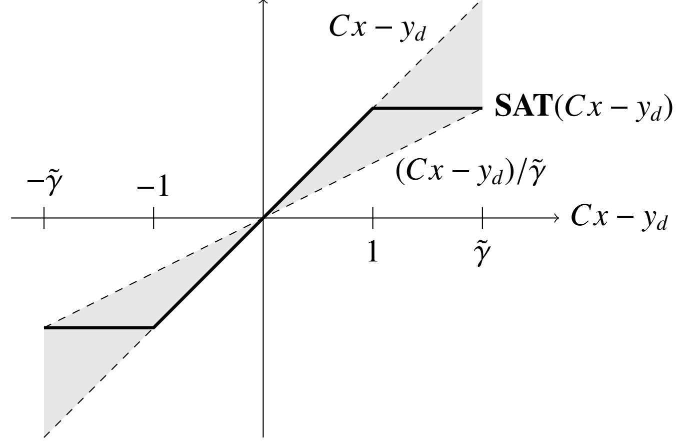

Figure 9: Illustration of the sector bound relationship between y_c (in the grey sector) and $Cx - y_d$.

Defining an extended state vector $\epsilon_k := [x_k\ y_{c,k}\ y_{d,k}\ 1]^T$ and the matrices

$$\mathcal{U} := \begin{bmatrix} A^{\mathrm{T}}PA & A^{\mathrm{T}}PB & 0_{4\times1} & 0_{4\times1} \\ B^{\mathrm{T}}PA & B^{\mathrm{T}}PB & 0 & 0 \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 0 & -1 \end{bmatrix}, \mathcal{V} := \begin{bmatrix} P & 0_{4\times1} & 0_{4\times1} & 0_{4\times1} \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 0 & -1 \end{bmatrix},$$

$$\mathcal{W} := \begin{bmatrix} \frac{2}{\tilde{\gamma}}C^{\mathrm{T}}C & -\left(1+\frac{1}{\tilde{\gamma}}\right)C^{\mathrm{T}} & -\frac{2}{\tilde{\gamma}}C^{\mathrm{T}} & 0_{4\times1} \\ -\left(1+\frac{1}{\tilde{\gamma}}\right)C & 2 & 1+\frac{1}{\tilde{\gamma}} & 0 \\ -\frac{2}{\tilde{\gamma}}C & 1+\frac{1}{\tilde{\gamma}} & \frac{2}{\tilde{\gamma}} & 0 \\ 0_{1\times4} & 0 & 0 & 0 \end{bmatrix}, \mathcal{Y} := \begin{bmatrix} 0_{4\times4} & 0_{4\times1} & 0_{4\times1} & 0_{4\times1} \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 1 & 0 \\ 0_{1\times4} & 0 & 0 & -0.5^2 \end{bmatrix}.$$

Equation (30) is rewritten as

$$\left(\epsilon_k^{\mathrm{T}}\mathcal{V}\epsilon_k \leq 0 \wedge \epsilon_k^{\mathrm{T}}\mathcal{Y}\epsilon_k \leq 0 \wedge \epsilon_k^{\mathrm{T}}\mathcal{W}\epsilon_k \leq 0\right) \implies \epsilon_k^{\mathrm{T}}\mathcal{U}\epsilon_k \leq 0.$$

Equation (30) can then be relaxed by S-procedure: it will hold if there exists nonnegative coefficients λ, μ, and ν, such that

$$\mathcal{U} - \lambda\mathcal{V} - \mu\mathcal{W} - \nu\mathcal{Y} \leq 0. \tag{31}$$

Equation (29) can be rewritten using Schur complement:

$$\begin{bmatrix} \gamma^2 & C \\ C^{\mathrm{T}} & P \end{bmatrix} \leq 0. \tag{32}$$

Note that for fixed λ and γ, Equations (31) and (32) form a Linear Matrix Inequality (LMI) in P, μ and ν, which means it can be solved thanks to a semidefinite programming solver [34]. $\tilde{\gamma} = \gamma + 0.5$ is expected to be larger than 1 (otherwise the saturation would never be activated, and one could revert to simpler analysis methods). Moreover, since the saturation should somewhat "bound" this value, it is expected not to span over multiple orders of magnitude. Also, the bottom right coefficient of the LMI in (31) indicates that $\lambda \in [0, 1]$. One possible strategy is then to iterate on potential values of λ and γ, and solving the corresponding LMI at each iteration. If a solution exists, it will provide the invariant $x^T P x \leq 1$ for the system with saturation. For our running example, we generated a suitable template in 279s on an Intel Core2 @ 2.4GHz. Values for λ are chosen by exploring $(0, 1)$ with numbers of the form $\frac{k}{2^i}$ for increasing values of $i \geq 1$, and $k < 2^i$. For each choice of λ, the LMI is solved with values of $\tilde{\gamma}$ ranging from 1 to 5 by increments of .1. The solution is found for $\lambda = \frac{63}{64}$ and $\tilde{\gamma} = 3.1$, which amounts to 2605 calls to the LMI solver. The solution is given by:

$$P = 10^3 \begin{bmatrix} 0.0494 & -0.0010 & 0.0086 & -0.5781 \\ -0.0010 & 0.0003 & -0.0006 & 0.0068 \\ 0.0086 & -0.0006 & 0.0066 & -0.0733 \\ -0.5781 & 0.0068 & -0.0733 & 7.0279 \end{bmatrix}, \lambda = 0.9844, \mu = 0.0235, \nu = 0.0601.$$

Variations of this example will be used throughout this disseration to illustrate different points and methods.

2.3 *Application: Extension of an Abstract Interpreter for Closed-Loop Stability Proofs*

The methods and heuristics presented in Section 2.2 are just a few examples of the array of analysis techniques available to the control engineer for the analysis of a given design. They will be used in the rest of this dissertation. They were also provided as input to a parallel work on a tool attempting to automatically extract ellipsoidal invariants on control code, which is briefly introduced in this section. This section is not intended to be a thorough

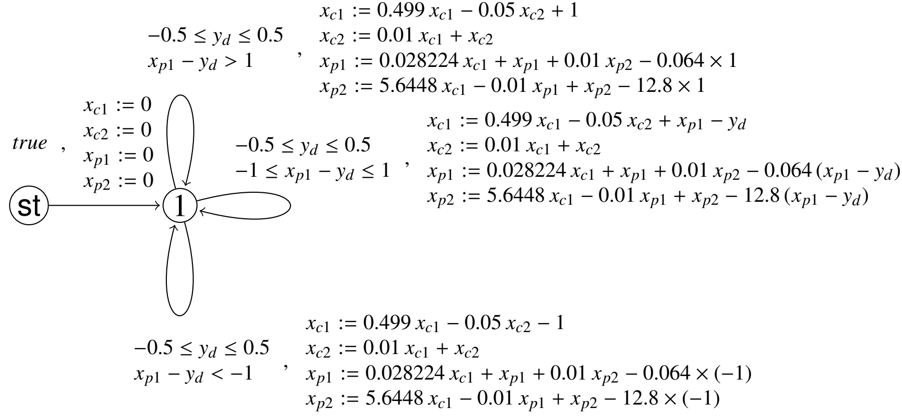

Figure 10: Control flow graph for the system with a saturation.

presentation of the tool, but rather an overview that emphasizes how this dissertation can contribute to bridge the gap between control theorists and computer scientists.

The tool, called smt-ai [41], works on objects called *control flow graphs*, which are used to represent the switching structure of a program, based on loops and branches. The control-flow graph for a program implementing the example closed-loop system in Section 2.2.4.3, is given in Figure 10.

Smt-ai uses the static analysis technique known as abstract interpretation [42], which has been very successful in automatically extracting invariants from programs. However, traditional static analyzers are scarcely concerned with the physical system interacting with the code they analyze, while smt-ai is. In addition, until recently, linear controllers, and particularly high-order ones, were out of reach for such analysis techniques, which often search for "box" invariants, the kind of which do not exist for linear systems. Policy iteration [43] offers to search for more exotic invariants. It works best, however, with an external input in order to have a sense of promising "shapes", or templates, of invariants. The heurisitics developed in the present chapter, based on domain-specific knowledge in control theory, were used to provide input to smt-ai. The focus was on proving the closed loop stability of a linear controller with a saturated input, interacting with a linear plant. The tool is able to successfully extract an invariant ellipsoid for our running example. In

addition, the guarantees offered are sound with respect to floating-point computation, an issue which is discussed furter in subsequent developments.

2.4 Conclusion

This chapter introduced some of the traditional concepts of control theory that are relevant to this research. The notion of ellipsoid, and different notations and results for these sets, were presented. Some heuristics were developed in order to obtain invariant ellipsoids on a variety of system models. These ellipsoids will be used in the following developments to enable an automatic code-level analysis of the implementation of a control law. They were also used as an input to enhance smt-ai, an existing static analyzer that, as a result, became, concurrently with the tool presented in this work, one of the first analyzers able to extract closed-loop properties of control systems.

Chapter III

A FORMAL FRAMEWORK TO EXPRESS CONTROL SEMANTICS AT THE LEVEL OF THE CODE

Current regulations for the certification of safety-critical software are mostly based on thorough documentation of the development life cycle, and heavy reliance on simulation and test. Although codified, these documents are, for the most part, generated by humans, to be read by humans. In this chapter, we explore the possibility of a formal annotation language supporting the description of control properties evinced by code implementations, and their automatic verification by a machine. To our knowledge, few formal code annotation languages exist, none of which were expressive enough to assert, for instance, the closed-loop stability of a piece of code in its interaction with a plant. A lot of the work in this thesis is based on the Hoare Logic framework, which will be introduced in Section 3.1. A high-level example of how to use the framework to express control properties on C code, first suggested in [24], is given in Section 3.2. Then, in Section 3.3, we choose an existing annotation language and describe a library of symbols we developed to meet a given set of expressivity needs. Finally, Section 3.4 gives examples of how the library can be used to assert that various control properties of interest hold for a given piece of code.

3.1 Deductive Methods and Hoare Logic

Hoare Logic was introduced in 1969 [44], following work by Floyd [45]. It is a method of annotation to describe a program's behavior, along with a set of rules, or axioms, that define what a correct annotation is. A *Hoare triple* $\{P\}\ S\ \{Q\}$ is composed of:

- a statement S, which is actual code.
- a precondition P, which is a predicate on the state of the program prior to statement S

being executed.

- a postcondition Q, which is another predicate on the state of the program following the execution of S.

The triple is *valid* if, assuming P holds before the execution of S, then Q holds after the execution of S. In order to present the various rules that compose Hoare axiomatic, the notion of axiom schema is defined: an axiom schema is given in the following form:

$$\frac{\{P_{req,1}\}\ S_{req,1}\ \{Q_{req,1}\}, \{P_{req,2}\}\ S_{req,2}\ \{Q_{req,2}\}, ..., \{P_{req,m}\}\ S_{req,m}\ \{Q_{req,m}\}}{\{P_{impl,1}\}\ S_{impl,1}\ \{Q_{impl,1}\}, \{P_{impl,2}\}\ S_{impl,2}\ \{Q_{impl,2}\}, ..., \{P_{impl,n}\}\ S_{impl,n}\ \{Q_{impl,n}\}.}$$

Axiom schemas describe patterns which, if followed, produce valid Hoare triples. In the above schema, each $\{P_{req,i}\}\ S_{req,i}\ \{Q_{req,i}\}$ describes a pattern of Hoare triples, potentially parameterized by variables. If there exists a choice of variables such that for all i, $\{P_{req,i}\}\ S_{req,i}\ \{Q_{req,i}\}$ holds, then the schema (or rule) expresses that each $\{P_{impl,j}\}\ S_{impl,j}\ \{Q_{impl,j}\}$ must also hold for the same choice of variables. The top part of the schema can be left empty, indicating that the patterns in the bottom part of the schema hold for any variable choice.

Examples of the rules that define valid triples include:

- The assignment rule:

$$\frac{}{\{P[E/x]\}\ x := E\ \{P\},}$$

  which expresses that given a postcondition P, a valid precondition for the assignment of variable x to expression E is $P[E/x]$, that is, a rewriting of P where all free instances of variable x are replaced by E. For example, the triples in Figure 11 are valid.

- The sequencing rule,

$$\frac{\{P\}\ S\ \{Q\},\ \{Q\}\ T\ \{R\}}{\{P\}\ S\ ;\ T\ \{R\},}$$

$$\begin{array}{l|l} \{i+1+j=10\} & \{i+j-1=9\} \\ \quad i := i+1 & \quad j := j-1 \\ \{i+j=10\} & \{i+j=9\} \end{array}$$

Figure 11: Valid Hoare triples under the assignment rule.

which expresses that, if two statements are run sequentially, and the postcondition for the first is the precondition for the second, then the triples can be combined so that the precondition of the first is the precondition of the sequence, and the postcondition of the second is the postcondition for the sequence. Using the previous two examples and this rule, one can conclude on the validity of the following triple:

$$\begin{aligned} &\{i+j=9\} \\ &\quad i := i+1; j := j-1 \\ &\{i+j=9\}. \end{aligned} \tag{33}$$

- The more elaborate loop rule, which involves an important element called an inductive invariant I:

$$\frac{\{I \wedge B\}\ S\ \{I\}}{\{I\}\ \texttt{while}\ B\ \texttt{do}\ S\ \{I \wedge \neg B\}.}$$

It expresses that, if one execution of the body of the loop preserves property I – that is, for any possible state of the variables where I holds before executing S, then I holds after executing S – then executing the whole loop preserves I as well. Using this rule and the previous example, it can be shown that the following triple is valid:

$$\begin{aligned} &\{i+j=9\} \\ &\quad \texttt{while}\ i < 10\ \texttt{do} \\ &\qquad i := i+1; \\ &\qquad j := j-1 \\ &\{i+j=9 \wedge i \geq 10\}. \end{aligned} \tag{34}$$

More often than not, one is more intested in finding the 'minimal' precondition required to achieve a target postcondition Q after the execution of a program. This concept of weakest precondition was introduced by Dijsktra in 1976 [46]. It is a mechanized way of extracting a first order property P which, if checked, proves that Q will hold after execution. The difficulty lies in the loop rule, where a relevant invariant must be found. Indeed, consider a simple linear system $x_{k+1} = Ax_k$. As discussed in Chapter 2, looking for a matrix P such that $A^{\mathrm{T}}PA - P \leq 0$ can yield an invariant ellipsoid for the system. However, the typical postconditions that are of interest to computer science are, for example, bounds on the variables. This leap from a postcondition like $\|x_k\|_\infty < b$, which is not an inductive invariant for the update loop of the system, to the invariant $x_k^{\mathrm{T}}Px_k < b'$, which implies the first postcondition with a proper choice of values for b and b', is one example of how domain-specific knowledge can contribute to program analysis.

3.2 Using Hoare Logic to Express a Simple Control Property on Code

It was suggested in [24] how to use an invariant set derived from a Lyapunov function, as a loop invariant for the implementation of the controller. In this section, a walkthrough of this process is given for a very simple 2-state system with no inputs. While this is not realistic in practice, it gives an idea of the more general technique. Start with the Lyapunov stable system in (35):

$$x_{k+1} = \begin{bmatrix} -0.1 & 0.1 \\ -0.3 & -0.2 \end{bmatrix} x_k,\ x_0 = \begin{bmatrix} 1 \\ 2 \end{bmatrix}. \tag{35}$$

The control theory results in Section 2.2.1 enable the discovery of

$$P = \begin{bmatrix} 0.2009 & 0.0093 \\ 0.0093 & 0.1904 \end{bmatrix},$$

such that the following holds:

$$\forall n \in \mathbb{N} : x_n^{\mathrm{T}}Px_n \leq 1. \tag{36}$$

A simple pseudo-algorithm to implement the system is shown in Figure 12. All it takes

```
x_0 := 1
x_1 := 2
while TRUE do
    x_0c := x_0
    x_0 := −0.1x_0 + 0.1x_1
    x_1 := −0.3x_0c − 0.2x_1
end while
```

Figure 12: Pseudo-code describing a simple linear system.

```
x_0 := 1
x_1 := 2
{[x_0  x_1] ∈ E_P}
while TRUE do
    x_0c := x_0
    x_0 := −0.1x_0 + 0.1x_1
    x_1 := −0.3x_0c − 0.2x_1
end while
{FALSE}
```

Figure 13: Simple linear system annotated with valid loop invariant.

to express the quadratic invariant in (36) is the annotation seen in Figure 13. However, in the interest of making the proof of this annotation easier to discharge automatically, local contracts are added within the loop body. The line-by-line annotations are shown in Figure 14, with:

$$Q_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix} P^{-1} \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 1 & 0 \end{bmatrix}^{\mathrm{T}}, Q_2 = \begin{bmatrix} -0.1 & 0.1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix} Q_1 \begin{bmatrix} -0.1 & 0.1 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}^{\mathrm{T}},$$

and:

$$Q_3 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.2 & -0.3 \end{bmatrix} Q_2 \begin{bmatrix} 1 & 0 & 0 \\ 0 & -0.2 & -0.3 \end{bmatrix}^{\mathrm{T}} = AP^{-1}A^{\mathrm{T}}.$$

All that was used is the following property which holds for any positive semidefinite matrix Q, and is proven in Chapter 4:

$$\forall x \in \mathbb{R}^n, y \in \mathbb{R}^m, M \in \mathbb{R}^{m\mathrm{x}n} : x \in \mathcal{G}_Q \wedge y = Mx \implies y \in \mathcal{G}_{MQM^{\mathrm{T}}}. \tag{37}$$

$x_0 := 1$
$x_1 := 2$
$\left\{ \begin{bmatrix} x_0 & x_1 \end{bmatrix}^{\mathrm{T}} \in \mathcal{E}_P \right\}$
while TRUE **do**
$\quad \left\{ \begin{bmatrix} x_0 & x_1 \end{bmatrix}^{\mathrm{T}} \in \mathcal{G}_{P^{-1}} \right\}$
$\quad x_{0c} := x_0$
$\quad \left\{ \begin{bmatrix} x_0 & x_1 & x_{0c} \end{bmatrix}^{\mathrm{T}} \in \mathcal{G}_{Q_1} \right\}$
$\quad x_0 := -0.1x_0 + 0.1x_1$
$\quad \left\{ \begin{bmatrix} x_0 & x_1 & x_{0c} \end{bmatrix}^{\mathrm{T}} \in \mathcal{G}_{Q_2} \right\}$
$\quad x_1 := -0.3x_{0c} - 0.2x_1$
$\quad \left\{ \begin{bmatrix} x_0 & x_1 \end{bmatrix}^{\mathrm{T}} \in \mathcal{G}_{Q_3} \right\}$
$\quad$ `skip`
$\quad \left\{ \begin{bmatrix} x_0 & x_1 \end{bmatrix}^{\mathrm{T}} \in \mathcal{E}_P \right\}$
end while
$\{FALSE\}$

Figure 14: Fully annotated linear system.

While this example illustrates the process by which manual annotation of control code can be performed using results from control theory, it is missing one key element: in order to perform automatic verification activities on this code, the Hoare triples must be written in a machine-readable and, consequently, formal language.

3.3 A Formal Library for the Expression of Control Semantics on Code

We now seek to define a machine-readable language able to support the expression of control properties at code level. In order to obtain an actionable prototype in a reasonable amount of time, the choice was made to turn to an existing annotation language for the C programming language, called the ANSI/ISO C Specification Language (ACSL) [47]. ACSL is a formal annotation language which is *tooled*, that is, there exists a platform, called Frama-C [48], which can read both C code and ACSL annotations and provides an Application Program Interface (API) to perform analysis on annotated code. In addition, two Frama-C plugins, WP and Jessie, can handle Hoare Logic and weakest-precondition calculus. Finally, both tools can use a wide array of solvers, both automatic and manual theorem provers, in order to

C+ACSL

```
//@ requires i+j == 9;
//@ ensures i+j == 9;
{
 i= i+1;
 j= j-1;
}
```

Figure 15: Writing the Hoare triple of (33) in ACSL.

assess the validity of the analyzed code. The problem of proving the validity of annotations is studied in Chapter 5. Few similarly developed annotation languages exist. The closest would be JML [49] for Java,which also provides multiple tools for analysis. However, C remains more widely used in the area of embedded control systems.

A brief introduction of ACSL semantics is given in Section 3.3.1. Then, a library of ACSL symbols and axioms, which we developed to express and define a number of control theoretic concepts of interest, is described in Section 3.3.2

3.3.1 ACSL Syntax and Semantics

ACSL is closely connected to Hoare logic. The following introduces basic ACSL notations for the purpose of writing specific types of Hoare triples that will be used in subsequent developments.

3.3.1.1 Function and code contracts

A construct used pervasively in this work, and the basic specification tool of ACSL, is the function or code *contract*. A contract is a way to describe a pre- and post- condition, in the sense of Hoare (c.f. Section 3.1), for a piece of code or function. The keyword `requires` introduces a precondition, while the keyword `ensures` introduces a postcondition. For example, Figure 15 shows how the Hoare triple in (33) can be written in ACSL.

C+ACSL

```
//@ loop invariant i+j == 9;
while (i<10){
 i= i+1;
 j= j-1;
}
```

Figure 16: Writing the loop invariant of (34) in ACSL

3.3.1.2 Loop annotations

Before a loop, the keyword `loop invariant` specifies an inductive loop invariant, that is, a property which is true when first entering the loop, and which is preserved by any execution of the loop. For example, Figure 16 shows how the inductive invariant in (34) can be written in ACSL. The keyword `loop variant` enables the user to provide a positive integer quantity that strictly decreases at every iteration of the loop. This concept was not part of the original Hoare logic, as it pertains to termination: indeed the existence of such a quantity, if proven, will show that the loop is entered a finite number of times.

3.3.1.3 Behaviors

Some function outputs are expected to be significantly different depending on particular characteristics of their input. The keyword `behavior` enables to split contracts according to predicates on the input. For example, a contract describing the behavior of a saturation function is shown in Figure 17. The keyword `assumes` introduces the hypotheses that separate the various behaviors. Note that the keyword `\result` was also used in this example. It refers to the output of the function under consideration by the contract.

In this work, behaviors are also used in a simpler way: in the absence of `assumes` clause, behaviors can be used as a way of labelling contracts to keep track of them throughout the program.

```
C+ACSL
/*@ behavior above:
  @   assumes x > 1.;
  @   ensures \result == 1.;
  @ behavior below:
  @   assumes x < -1. ;
  @   ensures \result == -1.;
  @ behavior between:
  @   assumes x <= 1. && x >= -1.;
  @   ensures \result == x; */
float sat(float x){
  return (x>1.)?1.:((x<-1.)?-1.:x);
}
```

Figure 17: Using named behaviors to describe the semantics of the saturation function.

3.3.1.4 Axiomatic definitions of logic types, predicates and functions

In order to expand the range of properties that can be expressed within contracts and invariants, ACSL offers the possiblity to define axiomatized types, predicates and functions. This was used extensively in the development of the library that is presented in the next section. Such an axiomatization is preceded by the keyword `axiomatic`. Within it, the keyword `type` is used to introduce a new type. The keyword `logic` precedes functions and predicates, which can be introduced simply with their names and the types of their input and outputs. The semantics of such symbols can then be refined using the keyword `axiom`, which is followed by their properties of interest. Figure 18 shows the ACSL syntax for the expression of the Peano axioms defining natural numbers [50].

3.3.1.5 Ghost code

Ghost code is a way to introduce variables and operations on these variables without affecting the semantics of the code. Any valid C code can be written in ghost code as long it does not affect the actual variables, and thus change the semantics of the program. For example, one may want to use a loop counter for the unique purpose of expressing an upper bound on the number of times a specific loop is entered, without needing this counter to appear in the

ACSL
```
/*@ axiomatic peano {
  @ type axiomatic_integer;
  @ logic axiomatic_integer zero;
  @ logic axiomatic_integer successor(axiomatic_integer n);
  @ axiom successor_injection:
  @ \forall axiomatic_integer n, m;
  @   n == m <==> successor(n) == successor(m);
  @ axiom no_successor_is_zero:
  @ \forall axiomatic_integer n;
  @   successor(n) != zero;
  @ } */
```

Figure 18: Peano axioms for natural numbers in ACSL.

C+ACSL
```
//@ ghost int i = 0 ;
while ( ... ) {
//@ ghost i = i + 1 ;
...
}
//@ assert i <= 12 ;
```

Figure 19: Using ghost code to express a bound on the number of times a loop is run.

actual code. This can be achieved by using the ACSL code shown in Figure 19.

3.3.1.6 Floating-Point arithmetic

Computations in embedded applications are usually carried-out in the floating point domain, $\mathbb{F}$. This subset of $\mathbb{R}$ enables the representation of a wide array of real numbers in a fixed-length memory location, and most programming languages are equipped to efficiently perform the basic arithmetic operations, $+, -, \times, /$. However, $\mathbb{F}$ does not evince some of the most basic properties of $\mathbb{R}$, making reasoning on floating-point numbers difficult. Indeed, for example, $\mathbb{F}$ is not stable by any of the operations mentioned above. This leads to a semantic gap between the model analyses described in Chapter 2 and the code-level analysis. In this chapter and the next, we temporarily ignore this gap by assuming all variables and their

results are computed exactly. Although this treatment is unsound a priori, i.e., one could prove wrong results on the program, it is not vain: the developments in this chapter can be re-used in a framework that does take floating point computation errors into account. The approach is described in Chapter 5. Here some ACSL constructs used to distinguish between floating-point results and exact results of computations, introduced in [51], are defined.

Given a program variable `x`, the term `\exact(x)` refers to the value of `x` *had all computations been done in* $\mathbb{R}$, and *were all decimal numbers machine-representable*. For example, even though the number 0.1 is not machine representable (infinite binary representation), `\exact(0.1)` denotes the real number 0.1, and `\exact(2*0.1)` denotes the real number 0.2. More generally, the `\exact` value of an expression is obtained by replacing constants by their real value, variables by their `\exact` value, and operators by their real counterparts. In addition, the keyword `\error(x)` denotes the quantity `\abs(x - \exact(x))`, where `\abs` is the traditional absolute value function.

In some cases, a variable `x` is known at design time to approximate a certain desired quantity. For example, when estimating `\cos(x)` with a Taylor expansion `1-x*x/2`, one is more interested in the error `\abs(1-x*x/2 - \cos(x))`, which encompasses both the floating point error and the approximation error. Two ACSL keywords and a command exist to express this error. The command `//@ \set_model x e;` defines expression `e` as the quantity `x` is meant to approximate or model. From there on, one can use `\model(x)` to refer to `e`, and `total_error(x)` to refer to `\abs(x-\model(x))`. These keywords are used in Chapter 5 to propose an approach that leverages results in $\mathbb{R}$ and an approximation scheme to ensure that the semantic gap does not "break" the proofs, i.e. that the proposed framework is sound with respect to floating-point computations.

3.3.2 Linear Algebra in ACSL

An axiomatic library of ACSL symbols has been developed to express concepts and properties pertaining to linear algebra. Note that a more thorough and, arguably, more elegant linera

```
ACSL
/*@ type matrix;
  @ type vector; */
```

Figure 20: Vector and matrix types in ACSL.

```
ACSL
/*@ logic real mat_select(matrix A, integer i, integer j);
  @ logic real vect_select(vector x,integer i);
  @ logic integer vect_length(vector x);
  @ logic integer mat_row(matrix A);
  @ logic integer mat_col(matrix A);*/
```

Figure 21: Accessor functions in ACSL.

algebra library is described in Chapter 4 in PVS. Indeed PVS is a more expressive logic language than ACSL, as the latter only allows first-order logic. It is shown throughout this subsection how this limitation was circumvented, at the price of a sometimes cumbersome axiomatization.

3.3.2.1 Basic types, constructor and accessor functions

A matrix and a vector types are first introduced (Figure 20). Then, accessor functions can return the characteristics of a given structure, that is, the row-size (`mat_row`) or column-size (`mat_col`) of a matrix, the length of vector (`vect_length`), or an element at a certain position (Figure 21). ACSL does not support the use of arrays. This limitation makes the expression of a generic constructor for matrices impossible. Thus, for each matrix dimension $n \times m$ required in the annotated program, a constructor function `mat_of_nxm_scalar (real a_00,real a_01,...,real a_0m,real a_10,..., real a_nm);` is defined. For each such constructor, the functions `mat_select`, `mat_row` and `mat_col` must be axiomatized. The process of defining these multiple functions and axioms, a very tedious one, was itself automated as part of the development of the tool described in Chapter 5.

ACSL

```
/*@ axiom mat_of_1x2_scalar_select:
  @ \forall matrix A, real x0000, x0001 ;
  @ A == mat_of_1x2_scalar(x0000, x0001) ==>
  @
  @ mat_select(A, 0, 0) == x0000 \&\&
  @ mat_select(A, 0, 1) == x0001 ;
  @
  @ axiom mat_of_1x2_scalar_row:
  @ \forall matrix A, real x0000, x0001;
  @ A == mat_of_1x2_scalar(x0000, x0001) ==>
  @ mat_row(A) == 1;
  @
  @ axiom mat_of_1x2_scalar_col:
  @ \forall matrix A, real x0000, x0001;
  @ A == mat_of_1x2_scalar(x0000, x0001) ==>
  @ mat_col(A) == 2; */
```

Figure 22: Axiomatization of accessor functions for the constructor `mat_of_1x2_scalar`.

ACSL

```
/*@ logic matrix mat_add(matrix A, matrix B);
  @ logic matrix mat_mult(matrix A, matrix B);
  @ logic matrix mat_scalar_mult (real a, matrix A);
  @ logic matrix transpose(matrix A);
  @ logic matrix block_m(matrix a11, matrix a12, matrix a21, matrix
     a22);*/
```

Figure 23: Main matrix operators in ACSL.

These axioms are shown for a 1×2 matrix in Figure 22, for the sake of completeness.

3.3.2.2 Operators

Operators are introduced in order to express matrix addition, multiplication, scaling, transposition, and block matrix construction (Figure 23).

Each of these operators can be axiomatized using the accessor functions presented in the previous paragraph. The `mat_add` axiomatization is described here, as it is illustrative of the general process. The full library can be found in Appendix A. When two matrices have the same dimensions, the element of their sum at index i, j is simply the sum of their

ACSL

```
/*@ axiom mat_add_select:
  @   \forall matrix A, B;
  @     mat_row(A) == mat_row(B) ==>
  @     mat_col(A) == mat_col(B) ==>
  @       \forall integer i, j;
  @         0 <= i < mat_row(mat_add(A,B)) ==>
  @         0 <= j < mat_col(mat_add(A, B)) ==>
  @         mat_select(mat_add(A,B), i, j) ==
  @           mat_select(A, i, j) + mat_select(B, i, j);*/
```

Figure 24: Axiomatization of element accessor for the matrix addition function.

ACSL

```
/*@ axiom mat_add_row:
  @ \forall matrix A, B;
  @ mat_row(A) == mat_row(B) ==>
  @ mat_col(A) == mat_col(B) ==>
  @ mat_row(mat_add(A, B)) == mat_row(A);
  @ axiom mat_add_col:
  @ \forall matrix A, B;
  @ mat_row(A) == mat_row(B) ==>
  @ mat_col(A) == mat_col(B) ==>
  @ mat_col(mat_add(A, B)) == mat_col(A);
```

Figure 25: Axiomatization of dimension accessors for matrix addition function.

elements at i, j (see Figure 24).

When two matrices have the same dimensions, the dimension of the sum are those of either one (see Figure 25). One limitation of ACSL is that it does not allow one to define subtypes. As a consequence one cannot prevent, syntactically, the validity of the expression `mat_add(A,B)` where A and B do not have compatible dimensions. While this is not an issue for expressivity, it becomes one when trying to match the definition of matrix addition with that of the theorem prover used in Chapter 4. For this, and for all other symbols, we created a "throw-away axiom", which specifies that if compatible dimension conditions are not met, then the result of the operator has no meaning (see Figure 26). The symbol `mat_add_ext` has no semantics and is not axiomatized. As such, nothing can be proved

ACSL

```
/*@ logic matrix mat_add_ext(matrix A, matrix B);
  @ axiom mat_add_ext:
  @ \forall matrix A, matrix B;
  @ mat_row(A) != mat_row(B) ||
  @ mat_col(A) != mat_col(B) ==>
  @ mat_add(A,B) == mat_add_ext(A,B);*/
```

Figure 26: Axiomatizing the unspecified nature of matrix addition when dimensions are incompatible.

about it.

3.3.2.3 Quadratic inequality predicates

One crucial expressivity need in this work, which can encode numerous control properties of interest, is a predicate expressing that a given vector belongs to a certain ellipsoid. When a copy of a variable is made in a C program (as is necessary to perform an in-place matrix multiplication, for example), the Q-form of an ellipsoid as presented in Section 2.1.2 conveniently offers a way to store both the underlying ellipsoid invariant and linear dependencies between variables. Thus, we choose to use the so-called Q-form throughout this work.

In order to axiomatize a predicate expressing $x \in \mathcal{G}_Q$ (recall this is true if and only if Q is positive semidefinite and $\begin{bmatrix} 1 & x^\mathsf{T} \\ x & Q \end{bmatrix} \succeq 0$), we introduced and axiomatized predicates expressing that a matrix is symmetric (`predicate symmetric(matrix A);`) and positive semidefinite (`predicate semidefpos (matrix P);`). The definition and axiomatization of predicate `in_ellipsoidQ` is given in Figure 27. Note the presence of a converting symbol `V2Ml` used to convert a vector to a row-matrix.

3.3.3 Including Proof Elements

The contracts and annotations written in this work are often non-trivial, and the underlying mathematical reason for their validity is unlikely to be found by existing automatic tools such as SAT and SMT solvers. Indeed in all cases presented in this work, the solvers present

```
ACSL
/*@  predicate in_ellipsoidQ(matrix Q, vector x);
  @  predicate in_ellipsoidQ_ext(matrix Q, vector x);
  @  axiom in_ellipsoidQ_ext:
  @    \forall matrix Q, vector x;
  @      vect_length(x)!=mat_col(Q) ||
  @      mat_col(Q)!=mat_row(Q) ==>
  @      (in_ellipsoidQ(Q,x) <==> in_ellipsoidQ_ext( Q,  x));
  @  axiom in_ellipsoidQ:
  @    \forall matrix Q, vector x;
  @      vect_length(x)==mat_col(Q) &&
  @      mat_col(Q)==mat_row(Q) ==>
  @    ((symmetric(Q) && semidefpos(Q) &&
  @    semidefpos(block_m(eye(1),V2Ml(x),transpose(V2Ml(x)),Q))) <==>
  @        in_ellipsoidQ(Q, x));*/
```

Figure 27: Axiomatization of predicate `in_ellipsoidQ`

within Frama-C were unable to discharge the proof obligations automatically.

ACSL is an extensible language: plugins can be developed to describe new keywords added to its grammar. We used this feature in the development of a plug-in to Frama-C, called wp-local-tactic.The plugin introduces the keyword `PROOF_TACTIC` as a grammar extension within a contract. It enables to specify, for a given contract, an indication of a strategy to use in order to prove its validity.

This seemingly simple feature is of major value to the effort at hand: bringing domain-specific knowledge down to the code requires to be able to express facts, but also to provide reasons for these facts, especially if one intends to make the subsequent verification process automatic.

For example, the syntax in Figure 28 signals Frama-C to use the strategy `AffineEllipsoid` to prove the correctness of the local contract considered. This strategy refers to the useful result of (37), which makes the contract in question all but trivial.

C+ACSL

```
/*@ requires in_ellipsoidQ(Q1,vect_of_2_scalar(v_1,v_2));
  @ ensures in_ellipsoidQ(Q2,vect_of_3_scalar(v_1,v_2,v_3));
  @ PROOF_TACTIC(use_strategy (AffineEllipsoid)); */
{ // assignment of v_3
}
```

Figure 28: Example of the ACSL extension for proof strategies.

ACSL

```
/*@ logic matrix Q = mat_of_2x2_scalar(1.53,10.0,10.0,507);
  @ assert in_ellipsoidQ(Q,vect_of_2_scalar(v_1,v_2)); */
```

Figure 29: Annotation asserting that a vector is in an ellipsoid.

3.4 Using the Library

In order to illustrate the expressivity power of the ACSL symbols presented, a number of examples of their use in the expression of key control properties, for various types of systems is now presented.

3.4.1 Key Constructs in the Annotation Process

Equipped with the axiomatized library, one can now express that the vector composed of program variables v_1 and v_2 is in the set $\mathcal{G}_Q$ where $Q = \begin{pmatrix} 1.53 & 10.0 \\ 10.0 & 507 \end{pmatrix}$, using annotations in Figure 29.

The invariance of ellipsoid $\mathcal{G}_Q$ throughout any program execution can be expressed by the *loop invariant* in Figure 30. Within the loop, each line of code is annotated with a local contract, as in Figure 31.

3.4.2 Lyapunov Stability of a Linear System

We begin once again with an unforced system which, although unrealistic in an embedded setting, will give the reader a feel of how the ACSL linear algebra library can be used to

C+ACSL

```
//@ loop invariant in_ellipsoidQ(Q,vect_of_2_scalar(v_1,v_2));
while (true){
//loop body
}
```

Figure 30: Example of an ACSL annotation expressing an ellipsoidal invariant.

C+ACSL

```
/*@ requires in_ellipsoidQ(Q,vect_of_2_scalar(v_1,v_2));
  @ ensures in_ellipsoid(Q',vect_of_3_scalar(v_1,v_2,v_3));*/
{
// assignment of v_3
}
```

Figure 31: Example of a local ACSL contract, or Hoare triple.

annotate a program with its control semantics. The program computes the update for the discrete linear system $x_{k+1} = \begin{bmatrix} .4990 & -0.05 \\ 0.01 & 1.0 \end{bmatrix} x_k$, for which a valid invariant ellipsoid $\mathcal{G}_Q$, found using a method described in Section 2.2.1, is given by $Q = 10^3 \begin{bmatrix} 1.4849 & -0.0258 \\ -0.0258 & 0.4061 \end{bmatrix}$. The fully annotated version can be seen in Figures 32 and 33.

Remarks Note that this is only the update function, and it needs to be called within a loop in order to implement the discrete system. We focus, in this work, on showing the preservation of properties by the update function of the controller, as this is equivalent to showing that these properties will be inductive invariants on a loop calling the update function.

Note also, that as the dimension of the state-space increases, the number of annotations rapidly explodes. Some fully annotated examples of actual controllers are shown in Appendix B, but within this section and in subsequent examples the focus will be on showing how to use specific features of ACSL and the linear algebra library to handle the expression

C+ACSL

```
#include "base.h"

/*@ logic matrix QMat_0 =mat_of_2x2_scalar(1.4849e3,-.0258e3,-.0258e3,
                            0.4061e3);
  @ logic matrix QMat_1 = mat_mult(
                             mat_mult(
                             mat_of_3x2_scalar(1.,0.,0.,1.,1.,0.),
                             QMat_0),
                             transpose(mat_of_3x2_scalar(1.,0.,0
                                          .,1.,1.,0.)));
  @ logic matrix QMat_2 = mat_mult(
                             mat_mult(
                             mat_of_3x3_scalar(0.,-0.05,0.499,
                                0.,1.,0.,0.,0.,1.),
                             QMat_1),
                             transpose(mat_of_3x3_scalar(0.,-0.05,0
                                .499,
                                          0.,1.,0.,0.,0.,1.)));
  @ logic matrix QMat_3 = mat_mult(
                             mat_mult(mat_of_2x3_scalar(1.,0.,0.,
                                         0.,1.,0.01),
                             QMat_2),
                             transpose(mat_of_2x3_scalar(1.,0.,0.,
                                          0.,1.,0.01)));*/
```

Figure 32: Annotated program describing the linear update of state variables (Part 1).

of various control properties.

Finally, it is important to remember that the process shown here is not meant to be a manual one: this expressivity framework has been developed with the specific purpose of being used within an autocoding environment. As such, readability is at times sacrificed for functionality.

Walking through the example We turn our attention to the first part of the linear system under consideration. In Figure 32, the file “base.h” is first included. It contains the linear algebra library described in Section 3.3.2. Then, 4 matrices describing 4 ellipsoids at various stages of the update function are defined. The actual code of the function is shown in Figure 33. There, an outer function contract describes how the state of the system remains in

C+ACSL

```
/*@ requires (\valid(xc + (0..1)));
  @ requires in_ellipsoidQ(QMat_0,vect_of_2_scalar(xc[0],xc[1]));
  @ ensures in_ellipsoidQ(QMat_0,vect_of_2_scalar(xc[0],xc[1]));
*/
void inst_compute(float* xc){
  float xc_0;
  /*@
    behavior EllipsoidMain_1:
    requires in_ellipsoidQ(QMat_0,vect_of_2_scalar(xc[0],xc[1]));
    ensures in_ellipsoidQ(QMat_1,vect_of_3_scalar(xc[0],xc[1],xc_0));
    @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
  */
  {
  xc_0= xc[0];
  }
  /*@
    behavior EllipsoidMain_2:
    requires in_ellipsoidQ(QMat_1,vect_of_3_scalar(xc[0],xc[1],xc_0));
    ensures in_ellipsoidQ(QMat_2,vect_of_3_scalar(xc[0],xc[1],xc_0));
    @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
  */
  {
    xc[0]= .4990*xc_0 - 0.05*xc[1];
  }
  /*@
    behavior EllipsoidMain_3:
    requires in_ellipsoidQ(QMat_2,vect_of_3_scalar(xc[0],xc[1],xc_0));
    ensures in_ellipsoidQ(QMat_3,vect_of_2_scalar(xc[0],xc[1]));
    PROOF_TACTIC (use_strategy (AffineEllipsoid));
  */
  {
    xc[1]= 0.01*xc_0 + 1.0*xc[1];
  }
  /*@
     behavior EllipsoidMain_4:
     requires in_ellipsoidQ(QMat_3,vect_of_2_scalar(xc[0],xc[1]));
     ensures in_ellipsoidQ(QMat_0,vect_of_2_scalar(xc[0],xc[1]));
     PROOF_TACTIC (use_strategy (PosDef));
  */
  {

  }
}
```

Figure 33: Annotated program describing the linear update of state variables (Part 2).

the ellipsoid $\mathcal{G}_{\text{QMat}_0}$. The keyword `\valid` simply indicates that the pointer `xc` is correctly allocated and points to an array of size 2. Note that the body of the function only contains allocations of linear combinations of the variables. Equation (37) describes how to transform each ellipsoid into the next, thus the tactic `AffineEllipsoid` can be used for most of the contracts. Only the final contract, which stipulates that the propagated ellipsoid $\mathcal{G}_{\texttt{QMat_3}}$ is included in the initial ellipsoid $\mathcal{G}_{\texttt{QMat_0}}$, uses a different tactic. Indeed, to perform this check, one can look at the positive-semidefiniteness of matrix `QMat_0` − `QMat_3`. This is what the tactic `PosDef` suggests our proving mechanism to do.

3.4.3 Bounded-Input, Bounded-State Stability of a Linear Controller

We now gradually introduce more and more complex properties and systems handled by this expressivity framework. In this section, the expression of a useful result on an actual controller with inputs and outputs is discussed. The example system under consideration is the lead-lag controller introduced in Section 2.2.4.3. Using the method suggested in Section 2.2.3, nonnegative numbers λ and μ, and a positive definite matrix P are found such that, for all $x \in \mathbb{R}^2, u \in \mathbb{R}$:

$$\begin{bmatrix} A_c^{\mathrm{T}} P A_c - \lambda P & A_c^{\mathrm{T}} P B_c & 0 \\ B_c^{\mathrm{T}} P A_c & B_c^{\mathrm{T}} P B_c - \mu I_m & 0 \\ 0 & 0 & 1 - \lambda - \mu \end{bmatrix} \preceq 0.$$

For the running example, $\lambda = 0.9991$, $\mu = 1 - \lambda$, and $P = 10^3 \begin{bmatrix} 0.6742 & 0.0428 \\ 0.0428 & 2.4651 \end{bmatrix}$ are appropriate. If such values exist, one can not only conclude on the input-to-state (and thus BIBO) stability of the controller, but also use $Q = P^{-1}$ and λ to annotate its implementation with the inductive invariant $x_k \in \mathcal{G}_Q$. For this, the following formulation of the S-Procedure result, adapted to the Q-form of an ellipsoid, is used: given 2 vectors x and y in $\mathbb{R}^n$ and $\mathbb{R}^m$ respectively, 2 positive coefficients λ_1, λ_2 such that $\lambda_1 + \lambda_2 \leq 1$, and 2 matrices Q_1, Q_2,

positive semidefinite, and of appropriate dimension:

$$x \in \mathcal{E}_{Q_1} \wedge y \in \mathcal{E}_{Q_2} \implies \begin{bmatrix} x \\ y \end{bmatrix} \in \mathcal{E}_Q, Q = \begin{bmatrix} Q_1/\lambda_1 & 0 \\ 0 & Q_2/\lambda_2 \end{bmatrix}. \tag{38}$$

The proof of this known result is recalled in Chapter 4. This relaxation technique enables the combination of 2 ellipsoids. It is typically used here to combine the ellipsoid information on the input with that of the state. The fully annotated code of the controller given here is shown in Appendix B. In Figure 34, only the Hoare triple where this combination operation occurs is shown: indeed the rest of the program is annotated similarly as the one in Section 3.4.2. Note that the contract is written for an empty block of code: recall that this is a simple trick in order to weaken a precondition (i.e., replace it with a precondition that is implied by it). Note also, that this relaxation occurs just before the control output is computed, that is, the first time in the program that a linear combination involving input and state appears (the controller state update occurs later).

3.4.4 Closed-Loop Stability of a Linear Controller Interacting with a Linear System

In order to go beyond BIBO-stability, and into the realm of closed-loop properties, that is, properties pertaining to the behavior of a controller in its interaction with a plant whose mode is known, it is necessary to introduce this model into the annotations. The most accurate way to do so would require a hybrid system representation, given that the plant is commonly a continuous system, while the digital controller is a discrete one. A large body of work is devoted to proving meaningful properties of hybrid systems. It is a very difficult task, because even simple properties have been shown to be undecidable on a large class of hybrid systems. In order to obtain actionable results, on which proof can be carried out, we made the choice of representing the plant as a linear system, discretized at the same period as the controller. To achieve this, we use ACSL ghost code feature. For each state variable in the plant, a global ghost variable is introduced. Within the update function of the controller, ghost code describing the state update resulting from the control output is added.

C+ACSL

```
/*@logic matrix QMat_20 = block_m(mat_scalar_mult(1.00090082,QMat_19),
  @  zeros(6,2),zeros(2,6),mat_scalar_mult(1111.1,QMat_18)); */

...

/*@  behavior EllipsoidMain_9:
  @  requires in_ellipsoidQ(QMat_18,vect_of_2_scalar(Sum4,D11));
  @  requires in_ellipsoidQ(QMat_19,vect_of_6_scalar(xc_1,xc_2,
  @                                Integrator_1,C11,Integrator_2,Sum3));
  @  ensures in_ellipsoidQ(QMat_20,vect_of_8_scalar(xc_1,xc_2,
                        Integrator_1,C11,Integrator_2,Sum3,Sum4,D11));
  @  PROOF_TACTIC (use_strategy (SProcedure)); */
{

}
/*@  behavior EllipsoidMain_10:
  @  requires in_ellipsoidQ(QMat_20,vect_of_8_scalar(xc_1,xc_2,
  @                     Integrator_1,C11,Integrator_2,Sum3,Sum4,D11));
  @  ensures in_ellipsoidQ(QMat_21,vect_of_9_scalar(xc_1,xc_2,
  @                                  Integrator_1,C11,Integrator_2,Sum3,
  @                                  Sum4,D11,control_output));
  @ PROOF_TACTIC (use_strategy (AffineEllipsoid)); */
{
    control_output = D11 + C11;
}
```

Figure 34: Program step using the S-procedure to obtain a relaxed combination of 2 ellipsoids in order to keep the propagation possible. Variable names have been changed for the sake of readability. The definition of matrix `QMat_20` is also given for completeness.

A template of the structure of the code is given in Figure 35. Note that, since the resulting system remains linear (of a higher order), the annotation procedure described in the previous subsection is unchanged. Simply, some variables in the ellipsoids are now ghost variables, and others are actual code variables.

3.4.5 Closed-Loop Stability of a Non-Linear Controller Interacting with a Linear System under the Absolute Stability Framework

The culmination of this expressivity work, is the expression of the stability of a system with a sector-bounded non-linearity. In order to show stability of such a system under the framework, one needs to be able to annotate the non-linearity with a sector bound. We present an example where the non-linearity is a simple unit saturation. In general, individual tactics are required for different types of non-linearities.

Recall the example closed-loop system given in Section 2.2.4.3:

$$x_{k+1} = Ax_k + B\mathbf{SAT}(y_k - y_{d,k}), \tag{39}$$

$$y_k = Cx_k, \tag{40}$$

where A, B, and C were defined in (27), $y_{d,k}$ is an external command input, and y_k is the output of the plant interacting with the controller. Recall from Section 2.2.4.3 that nonnegative coefficients λ, μ, and ν, as well as a positive definite matrix P were obtained, such that, as in (38):

$$\mathcal{U} - \lambda\mathcal{V} - \mu\mathcal{W} - \nu\mathcal{Y} \leq 0,$$

where

$$\mathcal{U} := \begin{bmatrix} A^{\mathrm{T}}PA & A^{\mathrm{T}}PB & 0_{4\times1} & 0_{4\times1} \\ B^{\mathrm{T}}PA & B^{\mathrm{T}}PB & 0 & 0 \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 0 & -1 \end{bmatrix}, \mathcal{V} := \begin{bmatrix} P & 0_{4\times1} & 0_{4\times1} & 0_{4\times1} \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 0 & 0 \\ 0_{1\times4} & 0 & 0 & -1 \end{bmatrix},$$

C+ACSL

```
/*@     ghost REAL xp_0;
  @     ghost REAL xp_1;
  @     ...
  @     ghost REAL xp_0_tmp;
  @     ghost REAL xp_1_tmp;
  @     ...*/
/*@ requires in_ellipsoidQ(Q,vect_of_n_scalar(xc[0],
                                              xc[1],...,xp_0,xp_1,...));
  @ ensures in_ellipsoidQ(Q,vect_of_n_scalar(xc[0],
                                              xc[1],...,xp_0,xp_1,...));
                                                   */
void update_fun(t_example_io *_io_, t_example_state *xc){
...
/*@ requires pre_i
  @ ensures post_i
  @ PROOF_TACTIC (use_strategy ( strategy_i ) )*/
{
// instruction i;
}
...
/*@     behavior Plant_N:
        requires in_ellipsoidQ(QMat_N,vect_of_m_scalar(xc[0],xc[1],
                                          ...,xp_0,xp_1,...,u));
        ensures in_ellipsoidQ(QMat_{N+1},vect_of_n_scalar(xc[0],xc[1],
                                          ...,xp_0,xp_1,...));
         @ PROOF_TACTIC (use_strategy (AffineEllipsoid));*/
{
    /*@
         ghost xp_0_tmp = xp_0;;
         ghost xp_1_tmp = xp_1;;
         ... */
    /*@
         ghost xp_0 = a_11 * xp_0_tmp + a_12*xp_1_tmp +.. + b1*u;;
         ghost xp_1 = a_21 * xp_0_tmp + a_22*xp_1_tmp +.. + b2*u;;
         ...*/
}
    /*@ behavior Plant_{N+1}:
     requires in_ellipsoidQ(QMat_{N+1},vect_of_n_scalar(xc[0],xc[1],
                                              ...,xp_0,xp_1,...));
         ensures in_ellipsoidQ(Q,vect_of_n_scalar(xc[0],xc[1],
                                    ...,xp_0,xp_1,...));
         @ PROOF_TACTIC (use_strategy (PosDef));
    */
{

}
}
```

Figure 35: Template of the update function with added plant semantics in ghost code.

$$\mathcal{W} := \begin{bmatrix} \frac{2}{\tilde{\gamma}} C^{\mathrm{T}} C & -\left(1+\frac{1}{\tilde{\gamma}}\right) C^{\mathrm{T}} & -\frac{2}{\tilde{\gamma}} C^{\mathrm{T}} & 0_{4\times 1} \\ -\left(1+\frac{1}{\tilde{\gamma}}\right) C & 2 & 1+\frac{1}{\tilde{\gamma}} & 0 \\ -\frac{2}{\tilde{\gamma}} C & 1+\frac{1}{\tilde{\gamma}} & \frac{2}{\tilde{\gamma}} & 0 \\ 0_{1\times 4} & 0 & 0 & 0 \end{bmatrix}, \mathcal{Y} := \begin{bmatrix} 0_{4\times 4} & 0_{4\times 1} & 0_{4\times 1} & 0_{4\times 1} \\ 0_{1\times 4} & 0 & 0 & 0 \\ 0_{1\times 4} & 0 & 1 & 0 \\ 0_{1\times 4} & 0 & 0 & -0.5^2 \end{bmatrix}.$$

which implies $x_k^{\mathrm{T}} P x_k \leq 1$ for all $k \in \mathbb{N}$. To understand the following annotation process better, note the following.

- The inequality $\mathcal{V} \leq 0$ encodes the fact that x_k is initially in $\mathcal{E}_P$, and its associated coefficient is λ.
- The inequality $\mathcal{W} \leq 0$ encodes the sector-bounded non-linearity information on **SAT**, and its associated coefficient is μ.
- The inequality $\mathcal{Y} \leq 0$ encodes the bound on $y_{d,k}$, and its associated coefficient is ν.

We can now use $Q = P^{-1}$ to proceed with the annotation process. We begin from $x \in \mathcal{G}_Q$ and can immediately use the fact that $y = C_p x$ to write $\begin{bmatrix} x & y \end{bmatrix}^{\mathrm{T}} \in \mathcal{G}_R$ with

$$R = \begin{bmatrix} I_4 \\ \begin{bmatrix} 0_{1\times 2} & C_p \end{bmatrix} \end{bmatrix} Q \begin{bmatrix} I_4 \\ \begin{bmatrix} 0_{1\times 2} & C_p \end{bmatrix} \end{bmatrix}^{\mathrm{T}}.$$

This being a direct consequence of (37). We can also use a relaxation with the coefficients given by λ and ν to combine the resulting ellipsoid with the constraint on y_d, $y_d^2 \leq 0.25$ or, in Q-form, $y_d \in \mathcal{G}_{0.25}$, according to (38):

$$\begin{bmatrix} x^{\mathrm{T}} \\ y \\ y_d \end{bmatrix} \in \mathcal{G}_S, S = \begin{bmatrix} R/\lambda & 0 \\ 0 & 0.25/\nu \end{bmatrix}.$$

We are now ready to annotate the non-linear part of the code: a postcondition for `y_c = sat (y-y_d);` is the sector bound obtained as follows: with $y - y_d$ being a linear combination of the states in $\mathcal{G}_S$, the following useful result holds:

$$\forall c \in \mathbb{R}^n, x \in \mathcal{G}_S \implies |c^{\mathrm{T}} x| \leq \sqrt{c^{\mathrm{T}} S c}. \tag{41}$$

Indeed,

$$y - y_d = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix} \begin{bmatrix} x^{\mathrm{T}} & y & y_d \end{bmatrix}^{\mathrm{T}},$$

and thus, letting $c = \begin{bmatrix} 0 & 0 & 0 & 0 & 1 & -1 \end{bmatrix}^{\mathrm{T}}$, and $d = \sqrt{c^{\mathrm{T}} S c}$, the following valid sector bound involving $y_c = \mathbf{SAT}(y - y_d)$ holds:

$$(y_c - (y - y_d))(y_c - \frac{1}{d}(y - y_d) \leq 0.$$

or, in matrix form,

$$\begin{bmatrix} x \\ y \\ y_d \\ y_c \end{bmatrix}^{\mathrm{T}} G \begin{bmatrix} x \\ y \\ y_d \\ y_c \end{bmatrix} \leq 0, G = \begin{bmatrix} \frac{cc^{\mathrm{T}}}{d} & -\frac{1+\frac{1}{d}}{2} c^{\mathrm{T}} \\ -\frac{1+\frac{1}{d}}{2} c & 1 \end{bmatrix} = \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix} \underbrace{\begin{bmatrix} \frac{1}{d} & -\frac{1}{2}\frac{d+1}{d} \\ -\frac{1}{2}\frac{d+1}{d} & 1 \end{bmatrix}}_{g} \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}^{\mathrm{T}}.$$

The ACSL predicate `neg_quad(matrix G, vector x)` is introduced for the occasion and simply represents the quadratic inequality $x^{\mathrm{T}} G x \leq 0$. Finally, one more relaxation step enables the combination of said inequality with the current ellipsoid. It stems from the following lemma:

Lemma 2 *Let $c, x \in \mathbb{R}^m$ be two vectors, $z \in \mathbb{R}$ a number, $Q \in \mathbb{R}^{m \times m}$ a positive semidefinite matrix, and μ a positive coefficient. Assume $c^{\mathrm{T}} Q c > 0$. Define:*

$$d = \sqrt{c^{\mathrm{T}} Q c}, g = \begin{bmatrix} \frac{1}{d} & -\frac{1}{2}\frac{d+1}{d} \\ -\frac{1}{2}\frac{d+1}{d} & 1 \end{bmatrix}.$$

Then, if $\underbrace{4 - \mu(d-1)^2}_{D} > 0$, the following holds.

- *The matrix $\begin{bmatrix} \frac{1}{d^2} & 0 \\ 0 & 0 \end{bmatrix} + \mu g$ is invertible. Denote its inverse h.*

- *The implication $x \in \mathcal{G}_Q \wedge \begin{bmatrix} x \\ z \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix} g \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} x \\ z \end{bmatrix} \leq 0 \implies \begin{bmatrix} x \\ z \end{bmatrix} \in \mathcal{G}_H$ holds, where*

C+ACSL

```
/*@ logic matrix QMat_3 = block_m(mat_scalar_mult(1/0.9844,QMat_2),
  @   zeros(5,1),zeros(1,5),mat_scalar_mult(1/0.0601,QMat_0));
  @ logic vector c = vect_of_6_scalar(0.0,0.0,0.0,-1.,1.);
  @ logic real d = \sqrt(dot_prod(c,vect_mult(Q,c)));
  @ logic matrix GMat_0 =
  @   mat_mult(block_m(v2ml(c,6),zeros(6,1),zeros(1,1),eye(1)),
  @     mat_mult(mat_of_2x2_scalar(1/d,-(d+1)/(2*d),-(d+1)/(2*d),1),
  @         transpose(block_m(v2ml(c,6),zeros(6,1),zeros(1,1),eye(1)))
  @ logic matrix QMat_4 = ell_neg_quad_comb(QMat3,c,0.0235);*/
...
```

Figure 36: Relevant part of annotated code for closed loop stability in the absolute stability problem. Names of variables have been changed for the sake of readability. Part 1: introduction of variables.

matrix H is defined by

$$H = \begin{bmatrix} Q - \frac{Qcc^{\mathrm{T}}Q}{c^{\mathrm{T}}Qc} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \frac{Qc}{c^{\mathrm{T}}Qc} & 0 \\ 0 & 1 \end{bmatrix} h \begin{bmatrix} \frac{Qc}{c^{\mathrm{T}}Qc} & 0 \\ 0 & 1 \end{bmatrix}^{\mathrm{T}}$$

The proof of this lemma is given in Chapter 4. Thus the coefficient μ computed earlier can be used, and the lemma applied with S standing for Q to obtain a single ellipsoid after the non-linearity. Figures 36 and 37 show the relevant parts of the annotated code. The H construct is axiomatized in ACSL with the symbol `matrix ell_neg_quad_comb(matrix Q, vector c, real mu)`.

3.4.6 Application to Fault Detection and Gain-Scheduled Controllers

The library can be used to express properties other than pure control. For example, it is well suited for use in a fault detection context. The premise is as follows: a linear dynamic system to control is given, for which 2 models exist: a faulty one, and a nominal one. the code for a controller in such a setting can be annotated to show that both the faulty plant and the nominal plant cannot destabilize a given observer. We will not show examples of code in this section, as the resulted annotated code (recall we are working in an autocoding context) becomes quickly unreadable by a human. However, the basic ACSL features that enable the

C+ACSL

```
/*@ requires y == 1.0 * xp_0;
@ requires in_ellipsoidQ(Q_Mat_0,vect_of_1_scalar(yd));
@ requires in_ellipsoidQ(Q_Mat_1,vect_of_4_scalar(xc_0,xc_1,
@                                  xp_0,xp_1));
@ ensures in_ellipsoidQ(Q_Mat_1,vect_of_4_scalar(xc_0,xc_1,
@                                  xp_0,xp_1));
*/
void update_fun(REAL* xc_0, REAL* xc_1, REAL* yd, REAL* y){
/*@  behavior Plant_0:
@ requires in_ellipsoidQ(QMat_1,vect_of_4_scalar(xc_0,xc_1,
@                  xp_0,xp_1));
@ ensures in_ellipsoidQ(QMat_2,vect_of_5_scalar(xc_0,xc_1,
@                                  xp_0,xp_1,y));
@ PROOF_TACTIC (use_strategy (AffineEllipsoid)); */
{ }
/*@  behavior Plant_1:
@ requires in_ellipsoidQ(QMat_0,vect_of_1_scalar(yd));
@ requires in_ellipsoidQ(QMat_2,vect_of_5_scalar(xc_0,xc_1,
@                                  xp_0,xp_1,y));
@ ensures in_ellipsoidQ(QMat_3,vect_of_6_scalar(xc_0,xc_1,
@                                             xp_0,xp_1,y,yd));
@ PROOF_TACTIC (use_strategy (SProcedure)); */
{ }
...
/*@  behavior Plant_10:
@ requires in_ellipsoidQ(QMat_3,vect_of_6_scalar(xc_0,xc_1,
@                                              xp_0,xp_1,y,yd));
@ ensures in_ellipsoidQ(QMat_3,vect_of_6_scalar(xc_0,xc_1,
@                                             xp_0,xp_1,y,yd));
@ ensures neg_quad(GMat_0,vect_of_7_scalar(xc_0,xc_1,xp_0,
@                                       xp_1,y,yd,yc));
@ PROOF_TACTIC(use_strategy(SatSectBound)); */
yc = sat(y-y_d);
/*@  behavior Plant_11:
@ requires in_ellipsoidQ(QMat_3,vect_of_6_scalar(xc_0,xc_1,
@                                              xp_0,xp_1,y,yd));
@ requires neg_quad(GMat_0,vect_of_7_scalar(xc_0,xc_1,
@                                         xp_0,xp_1,y,yd,yc));
@ ensures in_ellipsoidQ(QMat_4,vect_of_7_scalar(xc_0,xc_1,
@                                             xp_0,xp_1,y,yd,yc));
@ PROOF_TACTIC(use_strategy(EllNegQuadComb)); */
{}
```

Figure 37: Relevant part of annotated code for closed loop stability in the absolute stability problem. Names of variables have been changed for the sake of readability. Part 2: code structure.

expression and propagation of 2 sets of ellipsoids for the 2 plant models are the behaviors: indeed we can use 2 behaviors in the function contract to express the 2 stable ellipsoids under the different plant models. Within the function, in order to distinguish between these 2 propagations, the keyword `for` followed by the name of a function behavior will specifically express that the following contract only holds under the assumptions made for the specified behavior.

Similarly, a gain scheduled linear controller can be represented using a different behavior for each underlying linear system. Should a common Lyapunov function exist for such a system, the ellipsoid propagation can be carried out independantly for each possible value of the gains. Once again, the process generates n contracts for each line of code, where n is the number of linear systems the gain scheduled controller can switch between. This makes the size of the annotated file explode quickly.

3.5 *Conclusion and Perspectives*

We have defined a library of ACSL symbols which can be used to express the stability of a number of different types of systems. It can be extended in many directions: in the problem of absolute stability, there is no generic way to extract a sector bound from an unknown non-linearity: more work is needed to support more non-linearities of interest. The choice of ACSL was made because of the existence of tools which enabled the subsequent developments of this work, but there are many possible developments of the tools to create a more expressive ACSL. In particular the lack of support for arrays makes the annotation process considerably heavier than it could be. The first-order logic limitation creates issues when working with more expressive backends such as theorem provers. Nevertheless, this is, to our knowledge, the first formal annotation environment supporting the expression of control properties at the level of the code. Combined with an appropriate proof backend, which can soundly decide on the validity of the annotations shown throughout this chapter, it demonstrates the feasability of the credible autocoding framework and offers the promise

of a faster safety-critical software development cycle with higher safety guarantees.

Chapter IV

A LIBRARY OF MACHINE-PROVEN LINEAR ALGEBRA AND CONTROLS RESULTS

Verification and Validation (V&V) activities for safety-critical software currently involve a great deal of manual proof-reading of their documentation, cross-checking it against the code, and heavy amounts of simulations. As regulations become tighter and aircraft designs become more and more complex, the costs involved in this approach to V&V have become prohibitive, and already represent more than 50% of a whole aircraft development cost. In this chapter, we discuss ways of machine-verifying the formal annotations presented in Chapter 3. We introduce the notion of theorem prover in Section 4.1. We present our approach to extending one such theorem prover, PVS, with results from linear algebra in Section 4.2, and results from control theory, in Section 4.3.

4.1 Interactive Theorem Provers

Interactive theorem provers, also named proof assistants, are software tools that usually consist of a language to express mathematical constructs and properties, and an environment to interactively prove theorems. Based on existing axioms and definitions, the interactive prover checks the validity of the proof steps provided by a human user. Their automated counterparts, SAT and SMT solvers, use heuristics to attempt to automatically discharge proofs, but are not guaranteed to succeed in finite time, especially for the type of domain-specific, high-level, functional properties this research is invested in proving. The power of theorem provers lie in their soundness: proofs can only be made using the axioms provided by the system. As long as the axiom basis is a widely accepted, minimal set of mathematical results, any new result is assured to be valid. Theorem provers usually provide a high level

of expressivity, in the form of higher-order logic: unlike in ACSL and most SAT and SMT solvers languages, one can write quantified formulas over sets and functions.

PVS is one such proof assistant [52, 53], and the one used in this research. PVS is a specification language. A specification consists of a collection of *theories*, which contain type definitions, operations on these types, axioms, and theorems. Axioms are considered defining features and do not require a proof. Thus, they must be introduced with care in order not to break the proof system. Theorems have the same syntax as an axiom, but PVS expects a proof for them, using existing, proven results, and axioms. The language has a powerful type checker, which generates verification conditions anywhere a function is called to make sure its arguments are of the proper type. An interactive mode enables the user to provide proof for these verification conditions, as well as all theorems. In the following, we present the language features and the interactive proof environment.

4.1.1 The Prototype Verification System: Syntax and Semantics

Some of the most commonly used PVS symbols are now introduced. This section is in no way an exhaustive presentation of the PVS grammar and semantics, simply an introduction for following developments. PVS specifications are organized in theories introduced by the keyword `THEORY`.

4.1.1.1 Types and dependent types

In ACSL, and generally in commonly used programming languages, type systems do not permit dependent types. That is, one cannot define a type T that would vary according to a parameter. The reason programming languages do not allow this is because it makes the problem of deciding the type of all variables undecidable. In PVS, it is accepted that the user can provide input, including in the typechecking process. Thus, dependent types are allowed.

Within a theory, types can be defined using the keyword `TYPE`. PVS has a rich set of options for introducing new types, of which here are a few examples:

- $\mathbb{B}, \mathbb{R}, \mathbb{R}^{+}, \mathbb{R}^{+*}, \mathbb{N}, \mathbb{N}^{*}$ are all defined by default (respectively as `bool`, `real`, `posreal`, `nnreal`, `nat`, `posnat`).
- If `T1` and `T2` are types, then `[T1 ->T2]` is the set of functions from `T1` to `T2`
- if `f?` : `T1` $\mapsto \mathbb{B}$ is a predicate on `T1` (predicate names are traditionally followed by a ? in PVS), then `(f?)` is the subtype of `T1` of those elements which verify `f?`.
- The natural mathematical notation, `{ x : T1 | f?(x)}` is also available.
- Records are the equivalent of C structures: they group together named variables of potentially different types into a single one. For example, `[\# length : posnat , vect : [below(length)->real] \#]` could be one way of defining a vector, with one value of the record being its length, and the other a function returning a given element.

When it is unable to decide on the correctness or consistency of types in a theory, PVS creates what are called *Type-Correctness Conditions (TCC)*. They are small theorems which must be proven, before anything else in the theory, to ensure it is consistent and properly expressed.

Theories can be parameterized by a type or constant to make them generic: for example, a theory on vector spaces could be parameterized by the underlying scalar field. Several results that apply for generic fields can then be expressed together at once. The theory of real vectors in the NASA library, for example, is parameterized by the length of the vector, simply by introducing it like this: `vectors[n: posnat] : THEORY`.

4.1.1.2 Defining functions and constants

A declaration follows the pattern `Name(T1 arg1, T2 arg2,...):TYPE(=definition) ;` There can be uninterpreted symbols and types, hence the optional nature of the definition. Functions can be defined using the classic lambda abstraction [54]. For example a squaring function can be defined in both of these equivalent ways:

PVS

```
square1(x: real) : real = x*x;
square2 : [real -> real] = LAMBDA (x:real) : x*x
```

4.1.1.3 Axioms, Lemmas and Theorems

The keyword AXIOM introduces a boolean formula that is assumed to be true, to be used as a basis for other developments. The theories developed in this chapter are devoid of axioms, since they only use some fundamental axioms already available in the PVS standard library.

The keywords LEMMA and THEOREM interchangeably introduce a boolean formula, which is claimed by the user to be true. PVS requires proof of such claims. For example, one could write the following (and would be in for some trouble):

PVS

```
Fermat: THEOREM FORALL (x,y,z : nzreal, n:posnat):
                  n>2 IMPLIES NOT z^n = x^n + y^n ;
```

Note the introduction of `nzreal`, the type of non-zero reals, as well as the self explanatory FORALL ($\forall$), IMPLIES ($\Longrightarrow$) and IFF ($\Longleftrightarrow$).

4.1.2 The PVS Proof Environment

A PVS proof consists in the manipulation of a *sequent* of formulas, as shown in Figure 38. The horizontal line separates the hypotheses (on top) from the proof objectives (at the bottom). The sequent is considered valid when it is established that $A_1 \wedge A_2 \wedge A_3 \Longrightarrow B_1 \vee B_2 \vee B_3$. The initial sequent, when trying to prove formula A, is simply:

|– – – –

[1] A

The whole proof process consists in manipulating the sequent, either by invoking existing lemmas to add to the hypotheses, or by manipulating the formula (instantiating quantifiers,

$$
\begin{array}{l}
[-1]\ A_1 \\
[-2]\ A_2 \\
[-3]\ A_3 \\
\quad\vdots \\
|----\\
\ [1]\ B_1 \\
\ [2]\ B_2 \\
\ [3]\ B_3 \\
\quad\vdots
\end{array}
$$

Figure 38: A typical PVS proof sequent.

splitting multiple proof objectives,...) until its truth is made trivial. For example, if one A_i equals one B_j, the sequent is trivially valid.This is also true if one of the proof objectives is `TRUE` or one of the hypotheses is `FALSE`. This fact exacerbates the need for care when creating axioms: an axiom that can be proven false will make it possible to validate any sequent. Note that there are, embedded in PVS, complex decision procedures that enable it to conclude on the validity of a sequent for more elaborate cases than the ones just mentioned.

4.2 Linear Algebra Library in PVS

In this section, we introduce the linear algebra theories that were developed in PVS to support the proof of the annotated programs shown in Chapter 3.

4.2.1 Types and Constructors

The `Matrix` type is introduced as a record type containing the dimensions of the matrix, and a function which returns its elements, given a row and column index:

PVS

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

PVS

```
M2Block(m,n,p,q)(A: Mat(m,p),B: Mat(n,p),C: Mat(m,q),D: Mat(n,q)):
Block_Matrix =
(# rows1 := A`rows,rows2 := B`rows,
   cols1 := A`cols,cols2 := C`cols,
matrix :=
            LAMBDA (i: below(A`rows + B`rows), j: below(A`cols + C`
               cols)):
            IF i < A`rows THEN
                 IF j < A`cols THEN A`matrix(i,j)
                    ELSE C`matrix(i,j - A`cols)
                 ENDIF

                 ELSE IF j < A`cols THEN B`matrix (i - A`rows,j)
                    ELSE D`matrix(i - A`rows,j - A`cols)
                 ENDIF
            ENDIF
            #)
```

Figure 39: Block matrix definition in PVS.

The subtype `Mat(m,n)` represents matrices of given dimensions:

PVS

```
Mat(m, n): TYPE = {M: Matrix | M`rows = m and M`cols = n}
```

Similarly, the `Block_Matrix` type, which implements 2×2 block matrices, is defined as a record containing the 4 necessary dimensions, and an access function:

PVS

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

Naturally, the constructor of interest for block matrices is one that takes 4 matrices and returns the resulting block matrix shown in Figure 39. Typical matrix constructors are defined, such as the identity matrix (`I(n:posnat)`), the zero matrix (`Zero_mat(m,n:posnat)`), and conversion symbols are introduced to make the link between block matrices

and matrices. The reader should be aware of a possible confusion: while in ACSL, `block_m (A,B,C,D)` corresponds to $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$, in PVS, `M2Block(m,n,p,q)(A,B,C,D)` corresponds to $\begin{bmatrix} A & C \\ B & D \end{bmatrix}$.

4.2.2 Operators and Subtypes

The traditional matrix operations are introduced: transposition (`transpose(Matrix A)`), sum (`+(A: Matrix, B:(sameDim?(A)))`), multiplication by a scalar (`*(A:matrix, r: real)`), matrix product (`*(A:Matrix, B: { M:Matrix | M‘rows =A‘cols})`, etc.

Note the fundamentally more powerful expressiveness available here: these operators are defined on subtypes, so that trying to add matrices of different dimensions is not just undefined: it would constitute a type error that would be caught by PVS. This removes some of the difficulties encountered in ACSL.

Some useful subtypes are defined based on predicates: square matrices (`(square?)`), square matrices of a given dimension (`(SquareMat?(n:posnat))`), symmetric matrices (`(symmetric?)`), positive semidefinite matrices (`(semidef_pos?)`), etc.

It should be mentionned that this library is built on top the NASA PVS library, which contained a number of useful preliminaries. In particular, a very complete theory on vectors, and one on summation symbols, were used heavily.

4.2.3 Some Useful Results

More than 100 lemmas were written to facilitate the proving process of theorems of interest. They establish the associativity, commutativity, and distributivity among the various operators. They express results on matrix dimensions in a simple form, which enables PVS to automatically solve a number of typechecking issues regarding proper dimensions. They provide simplification strategies when identity and zero matrices are involved. Appendix C

lists them all. Some notable results on positive semidefinite matrices are given here:

$$\forall m, n \in \mathbb{N}^* A \in \mathbb{R}^{n\times n}, N \in \mathbb{R}^{m\times n}, A \geq 0 \implies NAN^{\mathrm{T}} \geq 0, \tag{42}$$

or, in PVS:

PVS

```
semidef_qua_trans:LEMMA FORALL(A:(square?),
                        N:{M:Matrix|M`cols=A`rows}):
          semidef_pos?(A) IMPLIES semidef_pos?(N*A*transpose(N))
```

The convex conicity properties of the set of positive semidefinite matrices:

$$\forall n \in \mathbb{N}^*, A \in \mathbb{R}^{n\times n}, a \in \mathbb{R}^+, A \geq 0 \implies aA \geq 0, \tag{43}$$

and

$$\forall n \in \mathbb{N}^*, A, B \in \mathbb{R}^{n\times n}, A \geq 0 \wedge B \geq 0 \implies A + B \geq 0, \tag{44}$$

or, in PVS:

PVS

```
semidef_scal: LEMMA FORALL (a:posreal, A:(semidef_pos?)):
          semidef_pos?(a*A);

semidef_sum: LEMMA FORALL (A:(square?), B:(same_dim?(A))):
           semidef_pos?(A) AND semidef_pos?(B)
           IMPLIES semidef_pos?(A+B);
```

4.3 Control Semantics in PVS

In Chapter 3, we introduced the notion of proof tactic. These keywords, added to ACSL contracts, can give clues as to the way of proving the validity of a Hoare triple. In this section, we introduce the major theorems, proven in PVS, which support the application of each tactic.

Note the introduction of the `in_ellipsoid_Q?` predicate, expressing that a vector x belongs to an ellipsoid $\mathcal{E}_Q$ and defined as in Figure 40.

PVS

```
in_ellipsoid_Q?(n:posnat, Q:SquareMat(n), x:Vector[n]): bool =
      semidef_pos?(Q) AND
      symmetric?(Q) AND
      semidef_pos?(Block2M(M2Block(1,n,1,n)
                           (I(1),transpose(V2M1(n,x)),V2M1(n,x),Q)))
```

Figure 40: Definition of `in_ellipsoid_Q?` in PVS.

PVS

```
ellipsoid_general: THEOREM
 FORALL (n:posnat,m:posnat, Q:SquareMat(n),
           M: Mat(m,n), x:Vector[n], y:Vector[m]):
              in_ellipsoid_Q?(n,Q,x)
              AND y = M*x
        IMPLIES
        in_ellipsoid_Q?(m,M*Q*transpose(M),y)
```

Figure 41: Theorem describing the transformation of an ellipsoid by a linear map in PVS.

4.3.1 Affine Ellipsoid Combination

Recall the result in Equation (37), valid for any positive semidefinite matrix X:

$$\forall x \in \mathbb{R}^n, y \in \mathbb{R}^m, M \in \mathbb{R}^{mxn} : x \in \mathcal{G}_X \wedge y = Mx \implies y \in \mathcal{G}_{MXM^\mathrm{T}}.$$

Its equivalent in PVS is shown in Figure 41. Note that the paper proof of this result is quite simple: it results from applying Equation (42) to $A = \begin{bmatrix} 1 & x^\mathrm{T} \\ x & X \end{bmatrix}$ and $N = \begin{bmatrix} 1 & 0 \\ 0 & M \end{bmatrix}$. However, many underlying assumptions are made when writing this mathematical statement, and PVS must be given a very detailed, step-by-step walkthrough of this proof. Overall, above 300 proof steps were used to prove this result.

This theorem is the main necessary result to prove the validity of contracts annotated with the proof tactic `AffineEllipsoid`.

4.3.2 Ellipsoid Combination through S-Procedure Relaxation

Recall the result in Equation (38), valid for any $x \in \mathbb{R}^n$, $y \in \mathbb{R}^m$, $\lambda_1, \lambda_2 \in \mathbb{R}^{+*}$ such that $\lambda_1 + \lambda_2 \leq 1$, Q_1 and Q_2 positive semidefinite matrices of appropriate dimensions:

$$x \in \mathcal{G}_{Q_1} \wedge y \in \mathcal{G}_{Q_2} \implies \begin{bmatrix} x \\ y \end{bmatrix} \in \mathcal{G}_Q, Q = \begin{bmatrix} Q_1/\lambda_1 & 0 \\ 0 & Q_2/\lambda_2 \end{bmatrix}.$$

This theorem proves the validity of the relaxation step known as S-Procedure for ellipsoids in the Q-form. Its PVS equivalent is shown in Figure 42. An outline of the proof is given in the following, however we begin with a lemma:

Lemma 3 *Given a vector $x \in \mathbb{R}^n$, a positive semidefinite matrix $Q \in \mathbb{R}^{n \times n}$, and a real number $\mu \geq 1$, the following holds:*

$$x \in \mathcal{G}_Q \implies x \in \mathcal{G}_{\mu Q}.$$

Proof 2 *The proof is obtained by using the definition of positive semidefiniteness: by hypothesis, $x \in \mathcal{G}_Q$, i.e., $\begin{bmatrix} 1 & x^{\mathrm{T}} \\ x & Q \end{bmatrix} \succeq 0$, or:*

$$\forall z \in \mathbb{R}, t \in \mathbb{R}^n, z^2 + 2zt^{\mathrm{T}}x + t^{\mathrm{T}}Qt \geq 0. \tag{45}$$

We are trying to show $x \in \mathcal{G}_{\mu Q}$, in other words:

$$\forall z \in \mathbb{R}, t \in \mathbb{R}^n, z^2 + 2zt^{\mathrm{T}}x + \mu t^{\mathrm{T}}Qt \geq 0. \tag{46}$$

This is an obvious consequence of Equation (45), since $\mu \geq 1$ combined with $Q \succeq 0$ makes the left hand side of Equation (46) always greater than or equal to the left hand side of Equation (45).

The proof of this lemma was also entered in PVS, and required about 500 proof steps. We now prove the main result as follows:

PVS

```
ellipsoid_combination: THEOREM
 FORALL (n,m:posnat, lambda_1, lambda_2: posreal, Q_1: Mat(n,n),
          Q_2: Mat(m,m), x:Vector[n], y:Vector[m], z:Vector[m+n]):
             in_ellipsoid_Q?(n,Q_1,x)
             AND in_ellipsoid_Q?(m,Q_2,y)
             AND lambda_1+ lambda_2 <= 1
             AND z = Block2V(V2Block(n,m)(x,y))
        IMPLIES
        in_ellipsoid_Q?(n+m,Block2M(M2Block(n,m,n,m)(1/lambda_1*Q_1,
                       Zero_mat(m,n),Zero_mat(n,m),1/lambda_2*Q_2)),z)
```

Figure 42: PVS theorem describing how two ellipsoids can be combined with proper multipliers.

Proof 3 *We first establish the result in the special case where $\lambda_1 + \lambda_2 = 1$. Apply Equation (42) to $A = \begin{bmatrix} 1 & x^{\mathrm{T}} \\ x & Q_1 \end{bmatrix}$ and $N = \begin{bmatrix} \sqrt{\lambda_1} & 0_{1\times n} & 0_{1\times m} \\ 0_{n\times 1} & I/\sqrt{\lambda_1} & 0_{n\times m} \end{bmatrix}^{\mathrm{T}}$ to obtain the positive semidefiniteness of $R_1 = NAN^T = \begin{bmatrix} \lambda_1 & x^{\mathrm{T}} & 0_{1\times m} \\ x & Q_1/\lambda_1 & 0_{n\times m} \\ 0_{m\times 1} & 0_{m\times n} & 0_{m\times m} \end{bmatrix}$. Through a similar argument, obtain the positive semidefiniteness of $R_2 = \begin{bmatrix} \lambda_2 & 0_{1\times n} & y^{\mathrm{T}} \\ 0_{n\times 1} & 0_{n\times n} & 0_{n\times m} \\ y & 0_{m\times n} & Q_2/\lambda_2 \end{bmatrix}$. Now, using Equation (44), we have $R_1 + R_2 \geq 0$, which is by definition equivalent to $\begin{bmatrix} x \\ y \end{bmatrix} \in \mathcal{G}_Q$.*

Now, if $\lambda_1 + \lambda_2 < 1$, let $\epsilon = 1 - \lambda_1 - \lambda_2$, $\lambda_2' = \lambda_2 + \epsilon$, and $\mu = \lambda_2'/\lambda_2$. Apply Lemma 3 to y, Q_2 and μ, to obtain $y \in \mathcal{G}_{\mu Q_2}$. To conclude, use the result when $\lambda_1 + \lambda_2 = 1$ by substituting λ_2 with λ_2' and Q_2 with μQ_2.

The lack of native support in the linear algebra library, for arbitrary sizes of block matrices made the PVS proof to this theorem particularly tedious, requiring around 1200 proof steps. This theorem is at the heart of the proof tactic `SProcedure` used in Chapter 3.

PVS

```
ellipsoid_bound : THEOREM
 forall(n:posnat, Q:SquareMat(n), c,x:Vector[n]):
   in_ellipsoid_Q?(n,Q,x) IMPLIES abs(c*x)<= sqrt(c*(Q*c))
```

Figure 43: PVS Lemma for bound extraction on linear combination of variables in an ellipsoid.

4.3.3 Ellipsoid Combinations Using Sector-Bounds

Three useful results, used in Chapters 2 and 3, are mentioned here, and have been proven in PVS. The first one describes how to extract bounds on linear combination of variables, when these variables belong to a given ellipsoid. With Q a positive semidefinite matrix, the following holds:

Lemma 4

$$\forall x, c \in \mathbb{R}^n : x \in \mathcal{G}_Q \implies |c^{\mathrm{T}}x| \leq \sqrt{c^{\mathrm{T}}Qc}.$$

Lemma 4 can be expressed in PVS with the code in Figure 43.

Proof 4 *By hypothesis,* $x \in \mathcal{G}_Q$*, i.e.,* $\begin{bmatrix} 1 & x^{\mathrm{T}} \\ x & Q \end{bmatrix} \geq 0$*. So,* $\forall z \in \mathbb{R}^{n+1} : z^{\mathrm{T}} \begin{bmatrix} 1 & x^{\mathrm{T}} \\ x & Q \end{bmatrix} z \geq 0$*. In particular, for* $z = \begin{bmatrix} -c^{\mathrm{T}}x \\ c \end{bmatrix}$*:*

$$(-c^{\mathrm{T}}x)^2 + 2(-c^{\mathrm{T}}x)c^{\mathrm{T}}x + c^{\mathrm{T}}Qc \geq 0,$$

which simplifies to

$$c^{\mathrm{T}}Qc \geq (c^{\mathrm{T}}x)^2.$$

The conclusion comes by taking the square root of this last inequality.

Next, we mention the result on the specific sector-bound obtained in the case of a saturation applied to a variable for which a bound is known:

$$\forall x \in \mathbb{R}, d \in \mathbb{R}^+, |x| \leq d \implies (\mathbf{SAT}(x) - x)(\mathbf{SAT}(x) - (1/d)x) \leq 0.$$

PVS

```
sat_sect_bound_step: LEMMA FORALL (n:posnat, Q:SquareMat(n),
  c,x: Vector[n], d:posreal):
    in_ellipsoid_Q?(n,Q,x) AND d >= sqrt(c*(Q*c))
    IMPLIES
    (sat(c*x)-(c*x))*(sat(c*x)-(1/d)*(c*x))<=0
```

Figure 44: PVS lemma expressing the sector-bound on a saturation.

In PVS, the lemma is tailored to its use with a bound stemming from an ellipsoidal set, as shown in Figure 44.

Proof 5 *The graphical proof in Figure 9 is probably the most telling. In PVS, the proof is obtained by a case-splitting on the various possible outcomes of the saturation function. In each subcase the result comes immediately. Indeed if* $\mathbf{SAT}(x) = x$*, the left hand side of the inequality is zero, making it true. If* $\mathbf{SAT}(x) = 1$*, then it must be that* $x \geq 1$*, making* $\mathbf{SAT}(x) - x$ *nonpositive. On the other hand,* $\mathbf{SAT}(x) - (1/d)x = 1 - (1/d)x = \frac{d-x}{d}$ *must be nonnegative, since* $x \leq d$*. Hence the inequality. A similar argument can be made if* $\mathbf{SAT}(x) = -1$.

Finally, the lemma describing a valid ellipsoid combination obtained from a sector bound is given here.

Lemma 5 *Let* $c, x \in \mathbb{R}^m$ *be two vectors,* $z \in \mathbb{R}$ *a number,* $Q \in \mathbb{R}^{m\times m}$ *a positive semidefinite matrix, and* μ*,* d_1*, and* d_2 *positive coefficients. Assume* $c^{\mathrm{T}}Qc > 0$*. Define:*

$$g = \begin{bmatrix} \frac{1}{d_1 d_2} & -\frac{1}{2}\frac{d_1+d_2}{d_1 d_2} \\ -\frac{1}{2}\frac{d_1+d_2}{d_1 d_2} & 1 \end{bmatrix}.$$

Then, if $\underbrace{4(d_1 d_2)^2 - \mu c^{\mathrm{T}}Qc(d_1 - d_2)^2}_{D} > 0$*, the following holds.*

- *The matrix* $\begin{bmatrix} \frac{1}{c^{\mathrm{T}}Qc} & 0 \\ 0 & 0 \end{bmatrix} + \mu g$ *is invertible. Denote its inverse h.*

- *The implication* $x \in \mathcal{G}_Q \wedge \begin{bmatrix} x \\ z \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix} g \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} x \\ z \end{bmatrix} \leq 0 \implies \begin{bmatrix} x \\ z \end{bmatrix} \in \mathcal{G}_H$ *holds, where matrix H is defined by*

$$H = \begin{bmatrix} Q - \frac{Qc c^{\mathrm{T}} Q}{c^{\mathrm{T}} Q c} & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \frac{Qc}{c^{\mathrm{T}} Qc} & 0 \\ 0 & 1 \end{bmatrix} h \begin{bmatrix} \frac{Qc}{c^{\mathrm{T}} Qc} & 0 \\ 0 & 1 \end{bmatrix}^{\mathrm{T}}$$

The PVS equivalent to this lemma is not shown here, as it is split in multiple intermediate lemmas, but can be found in Appendix C. Note that Lemma 2 from Chapter 3 is but a consequence of Lemma 5 for $d_1 = \sqrt{c^{\mathrm{T}} Qc}$ and $d_2 = 1$. The proof follows.

Proof 6 *Let G denote* $\begin{bmatrix} \frac{1}{c^{\mathrm{T}} Qc} & 0 \\ 0 & 0 \end{bmatrix} + \mu g$*. The fact that G is invertible comes from its determinant being D, which is, by assumption, non-zero. Since determinants were not available in PVS at the time of this proving, the explicit inverse of the matrix, using Cramer's formulas, was used instead to prove this invertibility.*

Notice that $\begin{bmatrix} c^{\mathrm{T}} x \\ z \end{bmatrix} \in \mathcal{E}_{G,(c^{\mathrm{T}} x)^2/(c^{\mathrm{T}} Qc)}$*. This comes as a straight-forward consequence of* $\begin{bmatrix} x \\ z \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix} g \begin{bmatrix} c & 0 \\ 0 & 1 \end{bmatrix}^{\mathrm{T}} \begin{bmatrix} x \\ z \end{bmatrix} \leq 0$ *(hypothesis), combined with* $(c^{\mathrm{T}} x)^2 \leq c^{\mathrm{T}} Qc$ *(true by Lemma 4).*

In addition, since G is invertible, $\mathcal{E}_{G,(c^{\mathrm{T}} x)^2/(c^{\mathrm{T}} Qc)} = \mathcal{G}_{h,(c^{\mathrm{T}} x)^2/(c^{\mathrm{T}} Qc)}$*, which means:*

$$\begin{bmatrix} \frac{(c^{\mathrm{T}} x)^2}{c^{\mathrm{T}} Qc} & c^{\mathrm{T}} x & z \\ c^{\mathrm{T}} x & & \\ z & & h \end{bmatrix} \geq 0.$$

Apply (42) *with A as the left-hand side of this last inequality, and* $N = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \frac{Qc}{c^{\mathrm{T}} Qc} & 0 \\ 0 & 0 & 1 \end{bmatrix}$*, to*

obtain:

$$\begin{bmatrix} \frac{(c^\mathrm{T}x)^2}{c^\mathrm{T}Qc} & \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}c^\mathrm{T}Q \quad z \\ \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc & \begin{bmatrix} \frac{Qc}{c^\mathrm{T}Qc} & 0 \\ 0 & 1 \end{bmatrix} h \begin{bmatrix} \frac{Qc}{c^\mathrm{T}Qc} & 0 \\ 0 & 1 \end{bmatrix}^\mathrm{T} \\ z & \end{bmatrix} \geq 0. \tag{47}$$

On the other hand, apply once more (42) *with* $\begin{bmatrix} 1 & x \\ x^\mathrm{T} & Q \end{bmatrix}$ *as A and* $\begin{bmatrix} 1 & -\frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}c^\mathrm{T} \\ 0 & I_n \end{bmatrix}$ *as N to obtain:*

$$\begin{bmatrix} 1 - \frac{(c^\mathrm{T}x)^2}{c^\mathrm{T}Qc} & x^\mathrm{T} - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}c^\mathrm{T}Q \\ x - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc & Q \end{bmatrix} \geq 0. \tag{48}$$

Then, apply the affine ellipsoid combination result from Equation (37)*, with* $\underbrace{x - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc}_{x'}$ *as x, the left hand side of* (48) *as X, and* $\underbrace{I_n - \frac{Qcc^\mathrm{T}}{c^\mathrm{T}Qc}}_{\mathcal{M}}$ *as M, to obtain that* $\mathcal{M}x' \in \mathcal{G}_{\mathcal{M}Q\mathcal{M}^\mathrm{T}, 1-\frac{(c^\mathrm{T}x)^2}{c^\mathrm{T}Qc}}$.

Now,

$$\begin{aligned} \mathcal{M}x' &= \left(I_n - \frac{Qcc^\mathrm{T}}{c^\mathrm{T}Qc}\right)\left(x - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc\right) \\ &= x - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc - \frac{Qcc^\mathrm{T}x}{c^\mathrm{T}Qc} + \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}\frac{Qcc^\mathrm{T}Qc}{c^\mathrm{T}Qc} \\ &= x - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc, \end{aligned}$$

and

$$\begin{aligned} \mathcal{M}Q\mathcal{M}^\mathrm{T} &= \left(I_n - \frac{Qcc^\mathrm{T}}{c^\mathrm{T}Qc}\right)Q\left(I_n - \frac{Qcc^\mathrm{T}}{c^\mathrm{T}Qc}\right)^\mathrm{T} \\ &= Q - \frac{Qcc^\mathrm{T}Q}{c^\mathrm{T}Qc} - \frac{Q(Qcc^\mathrm{T})^\mathrm{T}}{c^\mathrm{T}Qc} + \frac{Qcc^\mathrm{T}Qcc^\mathrm{T}Q}{(c^\mathrm{T}Qc)^2} \\ &= Q - \frac{Qcc^\mathrm{T}Q}{c^\mathrm{T}Qc}. \end{aligned}$$

Thus, $\mathcal{M}x' \in \mathcal{G}_{\mathcal{M}Q\mathcal{M}^\mathrm{T}, 1-\frac{(c^\mathrm{T}x)^2}{c^\mathrm{T}Qc}}$ *means:*

$$\begin{bmatrix} 1 - \frac{(c^\mathrm{T}x)^2}{c^\mathrm{T}Qc} & x^\mathrm{T} - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}c^\mathrm{T}Q \\ x - \frac{c^\mathrm{T}x}{c^\mathrm{T}Qc}Qc & Q - \frac{Qcc^\mathrm{T}Q}{c^\mathrm{T}Qc} \end{bmatrix} \geq 0. \tag{49}$$

Add (47) *and* (49) *to conclude.*

4.4 Conclusion and Perspectives

We have developed a set of PVS theories that contain useful results in linear algebra, and major theorems for control theory, in particular for the analysis of control law implementations. The full contribution can be found in Appendix C and online[1].

The process of formalizing mathematical results and adding them to a theorem prover is a long and tedious one. There is, however, great value in adding mathematical knowledge within these tools: once added, the results can be safely and soundly used forever afterwards.

There is room for the addition of numerous results pertaining to control theory. One possible direction is to build off the NASA PVS library initial probability theories, which could open the way, for example, for proofs of optimal behavior of Kalman filters in the presence of noise. In general, the certification of control strategies and algorithms based on randomness is a worthwhile endeavor: at the moment, the federal regulator is reluctant to allow any form of nondeterminism in the national airspace. However, the whole certification process is based on attaining a rate of catastrophic failure of less than 10^{-9} per flight hour. This probabilistic safety measure indicates the realization by the regulator that there is no 0-risk, who should also conclude that controlled randomness is less risky than uncontrolled determinism.

[1] `http://github.com/rjobredeaux/genecheck`

Chapter V

A TOOL FOR THE AUTOMATIC VERIFICATION OF FUNCTIONAL CONTROL PROPERTIES ON CODE

We are now equipped with a formal expressivity framework, suited to the automatic generation of annotated code, and with results from control theory proven in an interactive theorem prover. We develop a tool, which is able to fully automatically generate a certificate for the correctness of the annotations presented in Chapter 3. The tool, called *Genecheck*, is presented in this chapter. It is the glue that sticks together a number of different formal verification tools in order to achieve the desired verification activity.

It is assumed that the controller under analysis is given in the form of two C functions. One of them is an initialization function. The other implements a single execution of the control loop. It acquires inputs and updates the state variables and the outputs of the controller. This choice is motivated by the output format of the autocoder GeneAuto [55], an open-source Simulink to C translator.

Indeed it is the author's belief that the tools provided by this research must be used in conjunction with proper model-based design practices. Concurrent research [56] involves the extension of GeneAuto to generate code annotated with the type of annotations described in Chapter 3.

The properties of open and closed loop stability, as well as state-boundedness, can be established by solely considering the update function, which the rest of this section will focus upon. The function is assumed to essentially follow the template shown in Figure 46.

Given this annotated function, it remains to be proven that the annotations are correct with respect to the code. This is achieved by checking that each individual Hoare triple holds. Figure 45 presents an overview of the checking process. First, the WP plugin of

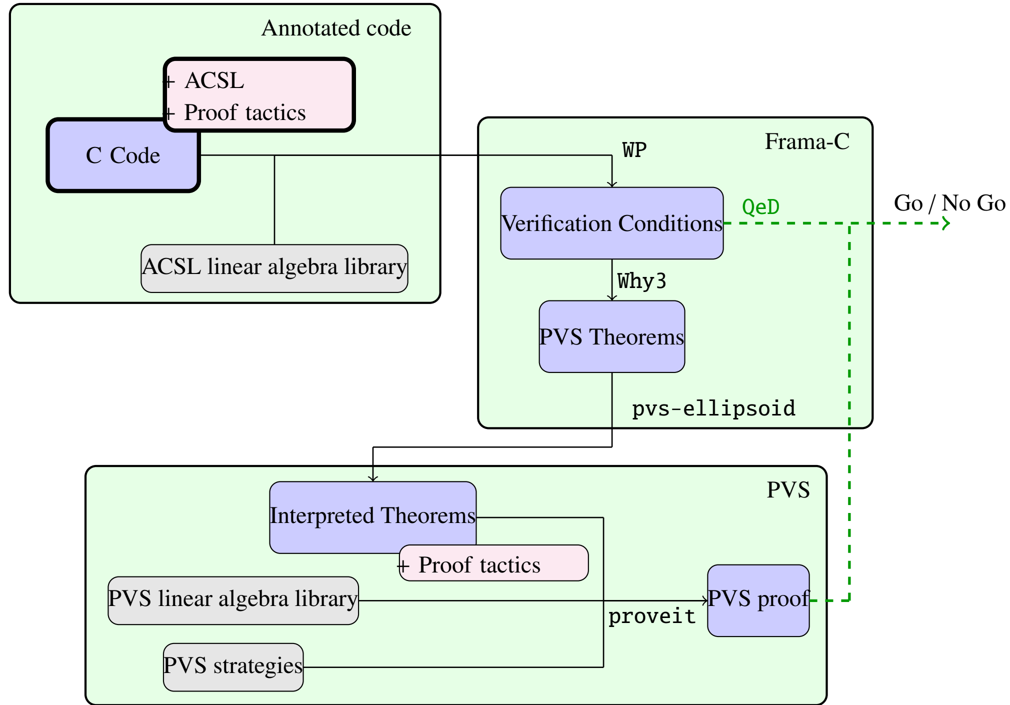

Figure 45: General view of the automated verification process. The annotated code is processed by Frama-C. PVS theorems that correspond to the verification conditions are generated. Automatic strategies are used to discharge the proofs, and fed back to Frama-C to conclude on the validity of the annotations.

C+ACSL

```
/*@ requires in_ellipsoidQ(Q,vect_of_n_scalar(_state_->s_1,
                                              _state_->s_2,
                                              ...));
  @ ensures in_ellipsoidQ(Q,vect_of_n_scalar(_state_->s_1,
                                             _state_->s_2,
                                             ...));*/
void example_compute(t_example_io *_io_, t_example_state *_state_){
...
/*@ requires pre_i
  @ ensures post_i
  @ PROOF_TACTIC (use_strategy ( strategy_i ) )*/
{
// instruction i;
}
...
}
```

Figure 46: Template of the generated loop update function.

Frama-C generates verification conditions for each Hoare triple, and discharges the trivial ones with its internal prover QeD. Then, the remaining conditions are translated into PVS theorems for further processing, as described in Section 5.1. It is then necessary to match the types and predicates introduced in ACSL in Chapter 3 to their equivalent representation in PVS, introduced in Chapter 4. This is done through theory interpretation [57] and explained in Section 5.1.3. Once interpreted, the theorems can be generically proven thanks to custom-made PVS strategies, as described in Section 5.2. In order to automate these various tasks and integrate our framework within the Frama-C platform, which provides graphical support to display the status of a verification condition (proved/unproved), we wrote a Frama-C plugin named pvs-ellipsoid, described in Section 5.3. Finally, the last verification condition in the program is of particular interest and handled differently from the others. It holds, by itself, the key to the floating-point correctness of the properties under consideration in this work, and is discussed in Section 5.4.

5.1 From C code to PVS Theorems

Frama-C is a collaborative platform designed to analyze the source code of software written in C. Before delving further into the verification process, the notion of weakest precondition is defined, which WP relies on.

5.1.1 Hoare Logic and Weakest Preconditions

When Hoare triples and axioms were introduced, no mechanical process or algorithm was provided to generically discharge them. It was up to the human user to manually pick relevant axioms to prove the validity of a given triple $\{P\}S\{Q\}$. In order to mechanize the proof process Edsger Dijkstra introduced the notion of weakest precondition [46]. Given a postcondition Q and a piece of code S, $wp(S,Q)$ is the "weakest" precondition which ensures that Q holds after executing S. It is always true that $\{wp(S,Q)\}S\{Q\}$ is a valid triple. In addition, the precondition is called "weakest" in the sense that any property P' such that $\{P'\}S\{Q\}$ holds verifies $P' \implies wp(S,Q)$. From a set-theoretical perspective $wp(S,Q)$

represents the maximal (in the sense of inclusion) set of variable states which verifies Q after executing S, and P' represents any set included in it. This approach to Hoare logic is not more expressive of more powerful than the original presentation, but rather gives a mechanical approach to the verification of Hoare triples. For example, recall the sequencing rule:

$$\frac{\{P\}\ S\ \{Q\},\ \{Q\}\ T\ \{R\}}{\{P\}\ S\ ;\ T\ \{R\}.}$$

This axiom can be reformulated in the weakest-precondition framework as follows:

$$wp(S\ ;\ T, R) = wp(S, wp(T, R)).$$

5.1.2 Converting Verification Conditions to PVS Theorems

The WP plugin enables deductive verification of C programs, annotated with ACSL. For each Hoare triple $\{pre_i\}\mathrm{inst}_i\{post_i\}$, it generates a first order logic formula expressing $pre_i \implies wp(\mathrm{inst}_i, post_i)$. Through the Why3 platform, these formulas can be expressed as theorems in PVS, so that, for example, the ACSL/C triple shown in Figure 47, taken directly from our running example, becomes the theorem shown in Figure 48.

Note that, for the sake of readability, part of the hypotheses of this theorem, including hypotheses on the nature of variables, as well as hypotheses originating from Hoare triples present earlier in the code, are ommitted here. Note also that in the translation process, functions like `malloc_0` or `mflt_1` have appeared. They describe the memory state of the program at different execution points.

5.1.3 Theory Interpretation

At the ACSL level, a set of linear algebra symbols has been introduced, along with axioms defining their semantics, in Chapter 3. Each generated PVS theorem is written within a theory that contains a translation 'as is' of these definitions and axioms, along with some constructs specific to handling the semantics of C programs. For example, the ACSL axiom

C+ACSL

```
/*@
requires in_ellipsoidQ(QMat_4,
                       vect_of_3_scalar(_state_->Integrator_1_memory,
                                        _state_->Integrator_2_memory,
                                        Integrator_1));
ensures in_ellipsoidQ(QMat_5,
                      vect_of_4_scalar(_state_->Integrator_1_memory,
                                       _state_->Integrator_2_memory,
                                       Integrator_1,
                                       C11));
PROOF_TACTIC (use_strategy (AffineEllipsoid));
*/
{
    C11 = 564.48 * Integrator_1;
}
```

Figure 47: Typical example of an ACSL Hoare Triple.

PVS

```
wp: THEOREM
FORALL (integrator_1_0: real):
  FORALL (malloc_0: [int -> int]):
    FORALL (mflt_2: [addr -> real], mflt_1: [addr -> real],
            mflt_0: [addr -> real]):
      FORALL (io_2: addr, io_1: addr, io_0: addr, state_2: addr,
              state_1: addr, state_0: addr):
             ...
          => p_in_ellipsoidq(l_qmat_4,
                             l_vect_of_3_scalar(mflt_2(shift
                                                       (state_2, 0)),
                                                mflt_2(shift
                                                       (state_2, 1)),
                                                integrator_1_0))
          => p_in_ellipsoidq(l_qmat_5,
                             l_vect_of_4_scalar(mflt_2(shift
                                                       (state_2, 0)),
                                                mflt_2(shift
                                                       (state_2, 1)),
                                                integrator_1_0,
                                                (14112/25 *
                                                  integrator_1_0)))
```

Figure 48: Excerpt of the PVS translation of the triple shown in Figure 47.

```
ACSL
/*@ axiom mat_of_2x2_scalar_row:
  @ \forall matrix A, real x0101, x0102, x0201, x0202;
  @ A == mat_of_2x2_scalar(x0101, x0102, x0201, x0202) ==>
  @ mat_row(A) == 2; /*
```

Figure 49: ACSL axiom for the row-size of a 2x2 matrix.

```
PVS
q_mat_of_2x2_scalar_row:
 AXIOM FORALL (x0101_0:real, x0102_0:real,x0201_0:real, x0202_0:real)
    :
         FORALL (a_0:a_matrix):
   (a_0 = l_mat_of_2x2_scalar(x0101_0, x0102_0, x0201_0, x0202_0)) =>
   (2 = l_mat_row(a_0))
```

Figure 50: ACSL axiom in Figure 49 translated to PVS.

expressing the number of rows of a 2 by 2 matrix, shown in Figure 49 becomes, after translation to PVS, the axiom shown in Figure 50

In order to leverage the existing results on matrices and ellipsoids in PVS, theory interpretation is used. It is a logical technique used to relate one axiomatic theory to another. It is used here to map types introduced in ACSL, such as vectors and matrices, to their counterparts in PVS, as well as the operations and predicates on these types. To ensure soundness, PVS requires that what was written as axioms in the ACSL library be proven in the interpreted PVS formalism.

Recall, however, that ACSL only allows for total functions, and that subtypes cannot be introduced in ACSL. Thus, the straightforward theory interpretation, connecting `mat_add` with the + of PVS fails the basic TCC imposing that arguments to the latter function be of the same dimensions. The beginning of a solution was described in Section 3.3.2, with the introduction of “throw-away axioms”. Similarly, in order to reflect the axiomatization of `mat_add` in PVS, a total function is introduced, as in Figure 51. This process is carried out for each interpreted function. It carries the advantage of ensuring the match between

```
PVS
Matrix_add_ext(M1, M2: Matrix): Matrix
Matrix_add(M1, M2: Matrix): Matrix =
    if M1`cols = M2`cols and M1`rows = M2`rows
          then M1+M2
          else Matrix_add_ext(M1,M2) endif
```

Figure 51: The PVS counterpart to ACSL's `mat_add`, in the purpose of carrying out theory interpretation.

ACSL symbols and PVS symbols. Note that the expresseness work done in Chapter 3 did not require the axioms included in the library. Only in the context of theory interpretation do they become useful in ensuring a sound connection between libraries.

The interpreted symbols and soundness checks are the same for each proof objective, making the mechanization of such a process easier. Syntactically, a new theory is created, in which the theory interpretation is carried out, and the theorem to be proven is automatically rewritten by PVS in terms of its own linear algebra symbols. These manipulations on the generated PVS code are carried out by a Frama-C plugin we have written. It is called pvs-ellipsoid, and is described below.

5.2 Generically Discharging the Proofs in PVS

Once the theorem is in its interpreted form, all that remains to do is to apply the proper lemma to the proper arguments. Two different types of Hoare triples can be generated in ACSL. Two PVS strategies were written to handle these possible cases. A PVS proof strategy is a generic function describing a set of basic steps communicated to the interactive theorem prover in order to facilitate or even fully discharge the proof of a lemma. The `AffineEllipsoid` strategy handles any ellipsoid update stemming from a linear assignment of t<he variables. The theorem `ellipsoid_general`, introduced in Section 4.3, is recalled in Figure 52. To apply this theorem properly, the first step of the strategy consists of parsing the objectives and hypotheses of the theorem to acquire the name and the dimensions of

```
PVS
ellipsoid_general: LEMMA
 FORALL (n:posnat,m:posnat, Q:SquareMat(n),
           M: Mat(m,n), x:Vector[n], y:Vector[m]):
              in_ellipsoid_Q?(n,Q,x)
              AND y = M*x
        IMPLIES
        in_ellipsoid_Q?(m,M*Q*transpose(M),y)
```

Figure 52: The `ellipsoid_general` theorem in PVS.

the relevant variables, and to isolate the necessary hypotheses. The second step consists of a case splitting on the dimensions of the variable: they are given to the prover in order to complete the main proof, and then established separately using the proper interpreted axioms. Next, it is established that $y = Mx$ through a case decomposition and numerous calls to relevant interpreted axioms. All the hypotheses are then present for a direct application of the theorem. The difficulties in proof strategy design lie in intercepting and anticipating the typechecking constraints (TCC) that PVS introduces throughout the proof. A third strategy was specifically written to handle them.

The S-Procedure strategy follows a very similar pattern. It is somewhat simpler, since the associated instruction in the Hoare triple is a `skip`. It uses `ellipsoid_combination`, the other main theorem presented in Section 4.3.

5.3 The PVS-Ellipsoid Plugin to Frama-C

One major contribution developed as part of this dissertation is the pvs-ellipsoid plug-in to Frama-C, written in OCaml and available online[1]. The plugin automates the steps mentionned in the previous sections. It calls the WP plugin on the provided C file, then, whenever QeD fails to prove a step, it creates the PVS theorem for the verification condition through Why3 and modifies the generated code to apply theory interpretation. It extracts the proof tactic to be used on this specific verification condition, and uses it to signify what

[1] `http://github.com/rjobredeaux/genecheck`

strategy to use to prove the theorem at hand to the next tool in the chain, `proveit` [58]. `proveit` is a command line tool that can be called on a PVS file and attempts to prove all the theories in it, possibly using user guidance such as the one discussed in Section 5.2. When the execution of `proveit` terminates, a report is produced, enabling the plugin to decide whether the verification condition is discharged or not. If it is, a proof file is generated, making it possible for the proof to be replayed in PVS.

5.4 Checking Inclusion of the Propagated Ellipsoid, and Floating-Point Soundness

The final verification condition expresses that the state remains in the initial ellipsoid $\mathcal{G}_Q$. Through a number of transformations, it has been established that the state lies in some propagated ellipsoid $\mathcal{G}'_Q$. The conclusion of the verification lies in the final test $Q - Q' \geq 0$. The current state of the linear algebra library in PVS does not permit to make such a test, but a number of reliable external tools, like the INTLAB package of the MATLAB software suite, can operate this check. In the case of our framework, the pvs-ellipsoid plugin intercepts this final verification condition before translating it to PVS, and uses custom code from [59] to soundly ensure positive definiteness of the matrix.

Interestingly, the problem of checking the correctness of the annotations under floating-point computations can be concentrated in this final check. We give here an introduction to the nature of the floating-point problem, and outline a suggested process by which it can be handled.

There are two sources of floating-point errors to consider when assessing a verification framework. The first one comes from the computations in the program under analysis. They are made with a finite-memory machine which uses floating-point computations: as such the results of every computation is likely to differ from its ideal value in $\mathbb{R}$. The second source of error is the verification tool itself: it is also running on a machine and cannot perform exact real computations.

In the framework here presented, we suggest circumventing the second source of error

C+ACSL

```
/*@ requires in_ellipsoidQ(Q_1,vect_of_2_scalar(v_1,v_2));
  @ ensures in_ellipsoid(Q_2,vect_of_3_scalar(v_1,v_2,v_3));*/
{
v_3 = c_1*v_1 + c_2*v_2
}
```

Figure 53: Local ACSL contract before floating-point extension.

C+ACSL

```
//@ \set_model v_3 c_1*v_1 + c_2*v_2;
/*@ requires in_ellipsoidQ(Q_1,vect_of_2_scalar(v_1,v_2));
  @ ensures in_ellipsoid(Q_2,vect_of_3_scalar(v_1,v_2,\model(v_3)))
  @ ensures \total_error(v_3)<= e_1;*/
{
v_3 = c_1*v_1 + c_2*v_2
}
```

Figure 54: Local ACSL contract after floating-point extension.

by using rational numbers in the annotations: the current toolset actually automatically presents all numbers within annotations as rational numbers when converting the verification conditions to PVS. The operations involved are naturally more computationally intensive (one of the main reason for the wide use of floating-point computations is their efficiency), but the verification process is not as time-sensitive as the embedded program under analysis.

To handle the first source of errors, we propose the following approach. Define $\oplus : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$, and $\otimes : \mathbb{F} \times \mathbb{F} \to \mathbb{F}$ respectively as the addition and multiplication operators actually used by the program. Consider a typical contract within the loop body, as in Figure 53. The unsoundness is brought about by the fact that `v_3` is not equal to `c_1v_1 + c_2v_2`, but to (`c_1` $\otimes$ `v_1`) $\oplus$ (`c_2` $\otimes$ `v_2`). To correct this, introduce `\model(v_3)` as `c_1v_1 + c_2v_2`. This is made possible by the fact that, within ACSL comments, operators like +, *, etc. have their real sematics as operators in $\mathbb{R} \times \mathbb{R} \to \mathbb{R}$. Using the keywords introduced in Section 3.3.1.6, the contract in Figure 53 is rewritten as the sound

ACSL

```
/*@ requires in_ellipsoidQ(Q_prop,vect_of_n_scalar(\model(x1),\model(
    x2),...,\model(xn));
  @ requires \total_error(x_1) <= e_1;
  @ requires \total_error(x_2) <= e_2;
  @ ...
  @ requires \total_error(x_n) <= e_n;
  @ ensures in_ellipsoidQ(Q_inv,
                          vect_of_n_scalar(x_1,x_2,...,x_n));
  @ PROOF_TACTIC(use_strategy(PosDef))
*/
```

Figure 55: Final contract in the program: checking that the propagated ellipsoid is included in the invariant, taking floating-point computations into account.

contract in Figure 54. The main difference is that `v_3` is replaced by its counterpart `\model(v_3)`, which is defined above the contract. In addition, an error term is introduced by `\total_error(v_3)<=e_1`. Recall that `\total_error` is the distance between `v_3` and its model. `e_1` is typically obtained using floating-point analysis tools like Gappa [28], or Fluctuat [60].

The contract transformation now outlined can be carried out throughout the loop, and each individual error bound can be tracked, so that, when reaching the end of the loop, a contract similar to Figure 55 is obtained. The matrix `Q_prop` represents the propagated ellipsoid, and the matrix `Q_inv` represents the ellipsoid whose invariance we are proving. Variable `vect_of_n_scalar(\model(x1), \model(x2),..., \model(xn))` is denoted as $\tilde{x}$, variable `vect_of_n_scalar(x_1,x_2,...,x_n)` as x, variable `vect_of_n_scalar(e_1,e_2,...,e_n)` as `eps`, and variable ϵ is introduced and defined as $\epsilon = x - \tilde{x}$. Then, the contract holds if

$$\tilde{x} \in \mathcal{G}_{Q_{prop}} \implies x \in \mathcal{G}_{Q_{inv}}. \tag{50}$$

For this exercise, it is assumed (as is always the case in practice) that Q_{inv} and Q_{prop} are invertible and introduce $P_1 = Q_{prop}^{-1}$ and $P_2 = Q_{inv}^{-1}$. Note that if Q_{inv} and Q_{prop} are given with rational entries, as suggested, P_1 and P_2 can be computed exactly. Recall that $\mathcal{E}_{P_1} = \mathcal{G}_{Q_{prop}}$

and $\mathcal{E}_{P_2} = \mathcal{G}_{Q_{inv}}$. Equation (50) is thus equivalent to

$$\tilde{x}^{\mathrm{T}} P_1 \tilde{x} \leq 1 \implies (\tilde{x} + \epsilon)^{\mathrm{T}} P_2 (\tilde{x} + \epsilon) \leq 1. \tag{51}$$

Now,

$$(\tilde{x} + \epsilon)^{\mathrm{T}} P_2 (\tilde{x} + \epsilon) = \tilde{x}^{\mathrm{T}} P_2 \tilde{x} + 2\epsilon^T P_2 \tilde{x} + \epsilon^{\mathrm{T}} P_2 \epsilon.$$

A bound on $||\epsilon||_2$ is given by $||\texttt{eps}||_2$. The theoretically optimal bound for $||\tilde{x}||_2$ that can be drawn from $\tilde{x}^{\mathrm{T}} P_1 \tilde{x} \leq 1$ is $1/\lambda_{\min}(P_1)$, where $\lambda_{\min}(P_1)$ is the minimum eigenvalue of P_1. However, to keep the development simple, we do not seek a sound way of computing this quantity. Instead we use the following approach. First check that $P_1 \geq P_2$ using the sound checker in [59]. This is the necessary check of final inclusion when ignoring floating-point computations. If it fails, there is likely an error in the annotations. Then, compute $R = (r_{i,j})_{1 \leq i,j \leq n}$ so that $R^{\mathrm{T}} R$ is the Cholesky decomposition of P_2. Under the assumption of rational coefficients, this can also be done exactly. Note then, that

$$||R\tilde{x}||_2^2 = \tilde{x}^{\mathrm{T}} R^{\mathrm{T}} R \tilde{x} = \tilde{x}^{\mathrm{T}} P_2 \tilde{x} \leq \tilde{x}^{\mathrm{T}} P_1 \tilde{x} \leq 1.$$

In addition,

$$||R\epsilon||_2^2 \leq ||R||_F^2 ||\epsilon||_2^2 \leq ||R||_F^2 ||\texttt{eps}||_2^2,$$

by the submultiplicative property of the Frobenius norm. All quantities on the left-hand side can also be computed exactly here. Finally, let $\eta = 2||R||_F ||\texttt{eps}||_2 + ||R||_F^2 ||\texttt{eps}||_2^2$, and note by the Cauchy-Schwartz inequality that $\left|2\epsilon^T P_2 \tilde{x} + \epsilon^{\mathrm{T}} P_2 \epsilon\right| \leq \eta$. Using the sound positive definiteness checker discussed earlier [59], it can be checked check that $P_1 - P_2/(1 - \eta) \geq 0$. If this check succeeds, the following can be claimed:

$$\begin{aligned}
(\tilde{x} + \epsilon)^{\mathrm{T}} P_2 (\tilde{x} + \epsilon) &= |\tilde{x}^{\mathrm{T}} P_2 \tilde{x} + 2\epsilon^T P_2 \tilde{x} + \epsilon^{\mathrm{T}} P_2 \epsilon| \\
&\leq \tilde{x}^{\mathrm{T}} P_2 \tilde{x} + \eta \\
&\leq \tilde{x}^{\mathrm{T}} (1 - \eta) P_1 \tilde{x} + \eta \\
&\leq (1 - \eta) \tilde{x}^{\mathrm{T}} P_1 \tilde{x} + \eta \\
&\leq 1 - \eta + \eta = 1.
\end{aligned}$$

Note that this check is not guaranteed to succeed: indeed if it does not, it may mean that floating point computations are "breaking" the stability of the system, or it may simply mean that the error bounds computed are not tight enough to conclude. Recall that this research is focused on soundness, that is, the checker can never claim the system is safe if it is not.

5.5 Application and Benchmarking

Genecheck was successfully used to verify the systems described in Sections 3.4.3 and 3.4.4, which full code is provided in Appendix B. It takes an average of 30 seconds to check an individual contract, making the full verification process potentially long. However, note that there is a lot of room for improvement on this individual contract verification time, and that the total time increases linearly with the number of lines of code, which is itself roughly linear in the number of states of the system: these results suggest a decent level of scalability.

Note that the tool has been developed concurrently with the extended version of GeneAuto, GeneAuto+. As a consequence, the closed loop stability of any linear controller interacting with a linear plant can be proven by this tool, so long as its code was generated using GeneAuto+. This includes controllers and plants of any dimensions.

5.6 Conclusion and Perspectives

This chapter presented a tool, the sum of a development effort and the use of multiple formal analysis software, which takes as input a formally annotated control program as described in Chapter 3, and uses the library in Chapter 4 along with other analysis tools to conclude on the validity of the annotations.

The framework proposed to carry out floating-point verification, while sound in principle, would require some more development in order to integrate yet another tool into Genecheck. Nevertheless, to our knowledge, aside from the work presented in [61], no other endeavor offers to soundly verify the type of properties discussed in this work.

Chapter VI

CONCLUSION

This research has striven to bridge the gap between computer science and control theory, and to use the best of both fields with the objective of advancing V&V techniques.

Chapter 2 recalls basic elements and heuristics from control theory, used to establish the closed loop stability of various systems of interest. These heuristics were used to extend an existing abstract interpretation tool with the ability to automatically carry out these proofs on code. The results of this activity were published in [61]. Chapter 3 developed a formal language to describe control properties on C code. Initial results of this effort were published in [62]. The extension of the expressivity framework to the field of fault detection was presented in [63]. Chapter 4 described a library of linear algebra and controls results, proven in an interactive theorem prover. An account on this library and its potential for controller implementation verification was published in [64]. Finally, Chapter 5 implemented a tool to demonstrate the feasability of the credible autocoding framework: the tool can successfully, soundly, and automatically prove the validity of control annotations expressing closed loop stability of a system. The structure of the tool, as well as its integration with an extended code generation tool, are in press in [65].

In addition to showing the validity and the feasability of the credible autocoding framework, the prototype presented in this thesis also paves the way for numerous research directions.

One first possibility is to expand the type of properties and systems handled by the framework. In particular, robustness properties are close at hand: they are the natural next step from stability properties, and can usually be formulated using LMIs, which were already at the heart of this work.

Optimization algorithms are becoming more and more pervasive in path planning and efficient control strategies. Being able to provide the type of guarantees we presented on this type of algorithms would be a valuable endeavor: there are plenty of mathematical results that support the validity of such programs.

The general topic of probabilistic guarantees deserves more attention. All the properties considered in this work are based on deterministic systems. It would be useful to be able to express and prove results involving random perturbations on the input, or to express and prove a probability of failure. It is more realistic to claim an extremely low probability of failure, rather than to claim total safety of a system.

Engine manufacturers are considering a move to distributed architectures. The problem of certification for an engine controller running on multiple processors is a very challenging one. Similarly, formally proven graceful degradation on a manycore architecture, although a far away objective, is worth exploring.

More generally, it is this author's hope that this work will be one of many to involve deep collaboration between the field of computer science and that of control theory. The promise of higher safety standards, faster development cycles, and industry sensitization to formal methods, requires such collaboration to be brought about. The techniques presented have the potential to challenge engineers to think *formally* about the safety of their designs, and provide tangible, mathematical evidence for it.

Appendix A

ACSL LINEAR ALGEBRA LIBRARY

The following is the full ACSL linear algebra library of symbols and their axiomatization, developed as part of this thesis. It is publically available[1]. Note that the matrix constructors such as `mat_of_2x2_scalar` are not given here: they are generated by a Python script which parses through the file under analysis to check which dimensions are necessary, and appends these definitions at the end of the library.

```
/*@  axiomatic ellipsoids_proof_tactics {
 @   type ellipsoids_tactics = Intuition | AffineEllipsoid | SProcedure
 @    | Identity | PosDef | SectBoundSat | SProcSat | EllipsoidBound;
 @   predicate use_strategy (ellipsoids_tactics t);
 @ }
 @ axiomatic matrices {
 @ type matrix;
 @ type vector;
 @  logic real mat_select(matrix A, integer i, integer j);
 @  logic real vect_select(vector x,integer i);
 @  logic integer vect_length(vector x);
 @  logic integer mat_row(matrix A);
 @  logic integer mat_col(matrix A);
 @
 @  logic matrix block_m (matrix a11, matrix a12,
 @                        matrix a21, matrix a22);
 @  logic matrix block_m_ext (matrix a11, matrix a12,
 @                            matrix a21, matrix a22);
 @  axiom block_m_ext:
 @    \forall matrix A11, A12, A21, A22;
 @    mat_row(A11) != mat_row(A12) ||
 @    mat_col(A11) != mat_col(A21) ||
 @    mat_row(A22) != mat_row(A21) ||
 @    mat_col(A22) != mat_col(A12) ==>
 @    block_m(A11, A12,A21,A22) == block_m_ext(A11, A12,A21,A22);
 @  axiom block_m_select:
 @    \forall matrix A11, A12, A21, A22;
 @    mat_row(A11) == mat_row(A12) ==>
 @    mat_col(A11) == mat_col(A21) ==>
 @    mat_row(A22) == mat_row(A21) ==>
 @    mat_col(A22) == mat_col(A12) ==>
 @    \forall integer i, j;
```

[1] http://github.com/rjobredeaux/genecheck

```
@    (0 <= i < mat_row(A11) ==>
@    0 <= j < mat_col(A11) ==>
@    mat_select(block_m(A11, A12,A21,A22),i, j) ==
@      mat_select(A11, i, j))
@      && (mat_row(A11)<= i < mat_row(A11)+ mat_row(A21) ==>
@      0<= j <mat_col(A11) ==>
@      mat_select(block_m(A11,A12,A21,A22),i,j) ==
@        mat_select(A21,i-mat_row(A11),j))
@        && (0 <= i < mat_row(A11) ==>
@        mat_col(A11) <= j < mat_col(A11)+mat_col(A12) ==>
@        mat_select(block_m(A11,A12,A21,A22),i,j) ==
@          mat_select(A12,i, j-mat_col(A11)))
@          && (mat_row(A11)<= i < mat_row(A11)+ mat_row(A21) ==>
@          mat_col(A11) <= j < mat_col(A11)+mat_col(A12) ==>
@          mat_select(block_m(A11,A12,A21,A22),i,j) ==
@            mat_select(A22,i-mat_row(A11),j-mat_col(A11)));
@  axiom block_m_row:
@    \forall matrix A11, A12, A21, A22;
@    mat_row(A11) == mat_row(A12) ==>
@    mat_col(A11) == mat_col(A21) ==>
@    mat_row(A22) == mat_row(A21) ==>
@    mat_col(A22) == mat_col(A12) ==>
@    mat_row(block_m(A11,A12,A21,A22)) == mat_row(A11)+mat_row(A21);
@  axiom block_m_col:
@    \forall matrix A11, A12, A21, A22;
@    mat_row(A11) == mat_row(A12) ==>
@    mat_col(A11) == mat_col(A21) ==>
@    mat_row(A22) == mat_row(A21) ==>
@    mat_col(A22) == mat_col(A12) ==>
@    mat_col(block_m(A11,A12,A21,A22)) == mat_col(A11)+mat_col(A12);
@
@  logic vector block_v (vector v1, vector v2);
@  axiom block_v_select:
@    \forall vector v1, v2;
@    \forall integer i;
@    (0 <= i < vect_length(v1) ==>
@    vect_select(block_v(v1, v2),i) ==
@      vect_select(v1, i))
@      && (vect_length(v1)<= i < vect_length(v1)+ vect_length(v2) ==>
@      vect_select(block_v(v1,v2),i) ==
@        vect_select(v2,i-vect_length(v1)));
@  axiom block_v_length:
@    \forall vector v1, v2;
@    vect_length(block_v(v1,v2)) == vect_length(v1)+vect_length(v2);
@
@  logic matrix zeros(integer row, integer col);
@  logic matrix zeros_ext(integer row, integer col);
@  axiom zero_ext:
@    \forall int row, col;
@    row<=0 || col<=0 ==> zeros(row,col) == zeros_ext(row,col);
@  axiom zero_select:
@    \forall int row, col, i, j;
@    0<= i <row ==>
@    0<= j <col ==>
```

```
@    mat_select(zeros(row,col),i,j) == 0;
@  axiom zero_row:
@    \forall int row, col;
@    0 <row ==>
@    0 <col ==>
@    mat_row(zeros(row,col))==row;
@  axiom zero_col:
@    \forall int row, col;
@    0 <row ==>
@    0 <col ==>
@    mat_col(zeros(row,col))==col;
@
@  logic matrix mat_scalar_mult (real a, matrix A);
@  axiom mat_scalar_mult_select:
@    \forall real a, matrix A, int i,j;
@    0<= i < mat_row(A) ==>
@    0<= j < mat_col(A) ==>
@    mat_select(mat_scalar_mult(a,A),i,j) == a* mat_select(A,i,j);
@  axiom mat_scalar_mult_row:
@    \forall real a, matrix A;
@    mat_row(mat_scalar_mult(a,A)) == mat_row(A);
@  axiom mat_scalar_mult_col:
@    \forall real a, matrix A;
@    mat_col(mat_scalar_mult(a,A)) == mat_col(A);
@
@  logic vector vect_scalar_mult (real a, vector v);
@  axiom vect_scalar_mult_select:
@    \forall real a, vector v, int i;
@    0<= i < vect_length(v) ==>
@    vect_select(vect_scalar_mult(a,v),i) == a* vect_select(v,i);
@  axiom vect_scalar_mult_length:
@    \forall real a, vector v;
@    vect_length(vect_scalar_mult(a,v)) == vect_length(v);
@
@  logic matrix mat_add(matrix A, matrix B);
@  logic matrix mat_add_ext(matrix A, matrix B);
@  axiom mat_add_ext:
@    \forall matrix A, matrix B;
@    mat_row(A) != mat_row(B) ||
@    mat_col(A) != mat_col(B) ==>
@    mat_add(A,B) == mat_add_ext(A,B);
@  axiom mat_add_select:
@    \forall matrix A, B;
@    mat_row(A) == mat_row(B) ==>
@    mat_col(A) == mat_col(B) ==>
@    \forall integer i, j;
@    0 <= i < mat_row(mat_add(A, B)) ==>
@    0 <= j < mat_col(mat_add(A, B)) ==>
@    mat_select(mat_add(A, B), i, j) ==
@      mat_select(A, i, j) + mat_select(B, i, j);
@  axiom mat_add_row:
@    \forall matrix A, B;
@    mat_row(A) == mat_row(B) ==>
@    mat_col(A) == mat_col(B) ==>
```

```
@     mat_row(mat_add(A, B)) == mat_row(A);
@   axiom mat_add_col:
@     \forall matrix A, B;
@     mat_row(A) == mat_row(B) ==>
@     mat_col(A) == mat_col(B) ==>
@     mat_col(mat_add(A, B)) == mat_col(A);
@
@   logic matrix mat_mult(matrix A, matrix B);
@   logic matrix mat_mult_ext(matrix A, matrix B);
@   axiom mat_mult_ext:
@     \forall matrix A, matrix B;
@     mat_col(A) != mat_row(B) ==>
@     mat_mult(A,B) == mat_mult_ext(A,B);
@   logic real mat_mult_aux(matrix A, matrix B,
@   integer i, integer j, integer k);
@   axiom mat_mult_aux_below_zero:
@     \forall matrix A, B; \forall integer i, j, k;
@     k < 0 ==> mat_mult_aux(A, B, i, j, k) == 0;
@   axiom mat_mult_aux_below_ind:
@     \forall matrix A, B; \forall integer i, j, k;
@     k >= 0 ==> mat_mult_aux(A, B, i, j, k) ==
@       mat_mult_aux(A, B, i, j, k-1) +
@         mat_select(A, i, k)*mat_select(B, k, j);
@   axiom mat_mult_select:
@     \forall matrix A, B; mat_col(A) == mat_row(B) ==>
@     \forall integer i, j;
@     0 <= i < mat_row(mat_mult(A, B)) ==>
@     0 <= j < mat_col(mat_mult(A, B)) ==>
@     mat_select(mat_mult(A, B), i, j) ==
@       mat_mult_aux(A, B, i, j, mat_col(A)-1);
@   axiom mat_mult_row:
@     \forall matrix A, B; mat_col(A) == mat_row(B) ==>
@     mat_row(mat_mult(A, B)) == mat_row(A);
@   axiom mat_mult_col:
@     \forall matrix A, B; mat_col(A) == mat_row(B) ==>
@     mat_col(mat_mult(A, B)) == mat_col(B);
@
@   logic vector vect_mult(matrix A, vector x);
@   logic vector vect_mult_ext(matrix A, vector x);
@   axiom vect_mult_ext:
@     \forall matrix A, vector x;
@     vect_length(x)!=mat_col(A) ==>
@       vect_mult(A,x) == vect_mult_ext(A,x);
@   logic real vect_mult_aux(matrix A, vector x, integer i, integer k);
@   axiom vect_mult_aux_below_zero:
@     \forall matrix A, vector x; \forall integer i, k;
@     k < 0 ==> vect_mult_aux(A, x, i, k) == 0;
@   axiom vect_mult_aux_below_ind:
@     \forall matrix A, vector x; \forall integer i, k;
@     k >= 0 ==> vect_mult_aux(A, x, i, k) ==
@     vect_mult_aux(A, x, i, k-1)+mat_select(A, i, k)*vect_select(x, k);
@   axiom vect_mult_select:
@     \forall matrix A, vector x, integer i;
@     mat_col(A) == vect_length(x) ==>
```

```
@    0< i < mat_row(A) ==>
@    vect_select(vect_mult(A,x),i) ==
@       vect_mult_aux(A,x,i,vect_length(x)-1);
@  axiom vect_mult_length:
@    \forall matrix A, vector x;
@    mat_col(A) == vect_length(x) ==>
@    vect_length(vect_mult(A,x)) == mat_row(A);
@
@  logic matrix eye(integer i);
@  logic matrix eye_ext(integer i);
@  axiom eye_ext:
@    \forall integer i;
@    i<=0 ==> eye(i) == eye_ext(i);
@  axiom eye_select:
@    \forall integer n, integer i, integer j;
@    n>0 ==>
@    (((0<=i<n && 0<=j<n && i==j) ==> mat_select(eye(n),i,j) == 1)
@    && ((0<=i<n && 0<=j<n && i!=j) ==> mat_select(eye(n),i,j) == 0));
@  axiom eye_row:
@    \forall integer n;
@    n>0 ==>
@    mat_row(eye(n)) == n;
@  axiom eye_col:
@    \forall integer n;
@    n>0 ==>
@    mat_col(eye(n)) == n;
@
@  logic matrix transpose(matrix A);
@  axiom transpose_select:
@    \forall matrix A; \forall integer i, j;
@    0 <= i < mat_row(transpose(A)) ==>
@    0 <= j < mat_col(transpose(A)) ==>
@    mat_select(transpose(A), i, j) == mat_select(A, j, i);
@  axiom mat_transpose_row:
@    \forall matrix A; mat_row(transpose(A)) == mat_col(A);
@  axiom mat_transpose_col:
@    \forall matrix A; mat_col(transpose(A)) == mat_row(A);
@
@  logic matrix mat_of_array{L}(float *A, integer row, integer col);
@  axiom mat_of_array_select:
@    \forall float *A; \forall integer i, j, k, l;
@    mat_select(mat_of_array(A, k, l), i, j) == A[l*i+j];
@  axiom mat_of_array_size:
@    \forall float *A; \forall integer i,j;
@    mat_row(mat_of_array(A,i,j))==i &&
@    mat_col(mat_of_array(A,i,j))==j;
@
@  logic vector vect_of_array{L}(float *x, integer length);
@  axiom vect_of_array_select:
@    \forall float *x; \forall integer n,i;
@    vect_select(vect_of_array(x,n),i) == x[i];
@  axiom vect_of_array_length:
@    \forall float *x; \forall integer i;
@    vect_length(vect_of_array(x,i))==i;
```

```
@
@  logic real dot_prod (vector x, vector y);
@  logic real dot_prod_ext (vector x, vector y);
@  axiom dot_prod_ext:
@    \forall vector x, vector y;
@    vect_length(x)!= vect_length(y) ==>
@        dot_prod(x,y) == dot_prod_ext(x,y);
@  logic real dot_prod_aux (vector x, vector y, integer i);
@  axiom dot_prod_below_0:
@    \forall vector x, y, int i;
@    vect_length(x)==vect_length(y) ==>
@    i < 0 ==>
@    dot_prod_aux(x,y,i) ==0;
@  axiom dot_prod_ind:
@    \forall vector x,y,int i;
@    vect_length(x)==vect_length(y) ==>
@    i>=0 ==>  dot_prod_aux(x,y,i) == dot_prod_aux(x,y,i-1) +
@                  vect_select(x,i)*vect_select(y,i);
@  axiom dot_prod_val:
@    \forall vector x,y;
@    vect_length(x)==vect_length(y) ==>
@    dot_prod(x,y) == dot_prod_aux(x,y,vect_length(x)-1);
@
@ predicate square( matrix P);
@  axiom square_def:
@    \forall matrix P;
@    square(P) <==> mat_row(P)==mat_col(P);
@
@  predicate invertible(matrix P);
@  axiom inv_def:
@    \forall matrix P;
@    invertible(P) <==> square(P) &&
@    \exists matrix Q;
@    mat_row(Q) == mat_row(P) &&
@    mat_col(Q) == mat_col(P) &&
@    mat_mult(Q,P) == eye(mat_row(P)) &&
@    mat_mult(P,Q) == eye(mat_col(P));
@
@  logic matrix inverse(matrix P);
@  logic matrix inverse_ext(matrix P);
@  axiom inverse_def:
@    \forall matrix P;
@    invertible(P) ==> mat_mult(P,inverse(P)) == eye(mat_row(P)) &&
@    mat_mult(inverse(P),P) == eye(mat_col(P));
@  axiom inverse_row:
@    \forall matrix P;
@    invertible(P) ==> mat_row(inverse(P)) == mat_row(P);
@  axiom inverse_col:
@    \forall matrix P;
@    invertible(P) ==> mat_col(inverse(P)) == mat_col(P);
@  axiom inverse_ext :
@    \forall matrix P;
@    !invertible(P) ==> inverse(P) == inverse_ext(P);
@
```

```
@  predicate symmetric (matrix P);
@  axiom sym_def:
@    \forall matrix P;
@    (square(P) &&
@    transpose(P)==P) <==>
@    symmetric(P);
@
@  predicate semidefpos (matrix P);
@  predicate semidefpos_ext(matrix P);
@  axiom semidefpos_ext:
@    \forall matrix P;
@    !square(P) ==> (semidefpos(P) <==> semidefpos_ext(P));
@  axiom semidefpos_def:
@    \forall matrix P;
@    square(P) ==>
@    (\forall vector x;
@    (vect_length(x)==mat_row(P) ==>
@    dot_prod(x,vect_mult(P,x))>=0)) <==> semidefpos(P);
@
@  logic matrix V2Ml(vector x);
@  axiom V2Ml_select:
@    \forall vector x, integer j;
@    0<=j<vect_length(x) ==>
@    mat_select(V2Ml(x),0,j) == vect_select(x,j);
@  axiom V2Ml_row:
@    \forall vector x;
@    mat_row(V2Ml(x))==1;
@  axiom V2Ml_col:
@    \forall vector x;
@    mat_col(V2Ml(x)) == vect_length(x);
@
@  predicate in_ellipsoidQ(matrix Q, vector x);
@  predicate in_ellipsoidQ_ext(matrix Q, vector x);
@  axiom in_ellipsoidQ_ext:
@    \forall matrix Q, vector x;
@      vect_length(x)!=mat_col(Q) ||
@      mat_col(Q)!=mat_row(Q) ==>
@      (in_ellipsoidQ(Q,x) <==> in_ellipsoidQ_ext( Q,  x));
@  axiom in_ellipsoidQ:
@    \forall matrix Q, vector x;
@      vect_length(x)==mat_col(Q) &&
@      mat_col(Q)==mat_row(Q) ==>
@    ((symmetric(Q) && semidefpos(Q) &&
@    semidefpos(block_m(eye(1),V2Ml(x),transpose(V2Ml(x)),Q))) <==>
@    in_ellipsoidQ(Q, x));
@
@    predicate neg_quad(matrix G, vector x);
@    predicate neg_quad_ext(matrix G, vector x);
@    axiom neg_quad_ext:
@      \forall matrix G, vector x;
@      vect_length(x)!=mat_col(G) ||
@      mat_col(G)!=mat_row(G) ==>
@      neg_quad(G,x) <==> neg_quad_ext(G,x);
@  axiom neg_quad:
```

```
 @    \forall matrix G, vector x;
 @    vect_length(x)==mat_col(G) &&
 @      mat_col(G)==mat_row(G) ==>
 @      neg_quad(G,x) <==> dot_prod(x,vect_mult(G,x)) <= 0;}
*/
```

Appendix B

EXEMPLE ANNOTATED PROGRAMS

This appendix shows full examples of annotated programs using the annotation framework described in chapter 3. They can be analyzed by the tool presented in chapter 5. They are both available as test cases in the distribution of the tool, which can be found online[1].

B.1 Open Loop Stability

This is the fully annotated program for the system described in section 3.4.3.

```
/*
    simple_olg.c
    Generated by Gene-Auto toolset ver 2.4.10
    (launcher GALauncher)
    Generated on: 25/04/2014 10:26:30.965
    source model: simple_olg
    model version: 7.2
    last saved by:
    last saved on:
*/

/* Includes */

#include "base.h"
#include "simple_olg.h"

/* Variable definitions */

REAL simple_olg_yd = 0.0;
REAL simple_olg_y = 0.0;

/* Function definitions */

/*@
        requires \valid(_state_);
*/
void simple_olg_init(t_simple_olg_state *_state_) {
    _state_->Integrator_1_memory = 1;
    _state_->Integrator_2_memory = 1;
}
```

[1] http://github.com/rjobredeaux/genecheck

```
/*@
        logic matrix QMat_0 = mat_of_1x1_scalar(1.0);
*/
/*@
        logic matrix QMat_1 = mat_of_2x2_scalar(1710.0449662492558, -41
           .101885746811455, -41.101885746811455, 499.17657993449376);
*/
/*@
        logic matrix QMat_2 = mat_mult(mat_mult(mat_of_3x2_scalar(1.0,
           0.0, 0.0, 1.0, 1.0, 0.0), QMat_1), transpose(
           mat_of_3x2_scalar(1.0, 0.0, 0.0, 1.0, 1.0, 0.0)));
*/
/*@
        logic matrix QMat_3 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
           QMat_0), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_4 = mat_mult(mat_mult(mat_of_4x3_scalar(1.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 564.48),
           QMat_2), transpose(mat_of_4x3_scalar(1.0, 0.0, 0.0, 0.0, 1.0
           , 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 564.48)));
*/
/*@
        logic matrix QMat_5 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
           QMat_3), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_6 = mat_mult(mat_mult(mat_of_5x4_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.01, 0.0), QMat_4), transpose(
           mat_of_5x4_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.01, 0.0)
           ));
*/
/*@
        logic matrix QMat_7 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
           QMat_5), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_8 = mat_mult(mat_mult(mat_of_6x5_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 1.0, 0.0, 0.0, 0.0), QMat_6), transpose(
           mat_of_6x5_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0)));
*/
/*@
        logic matrix QMat_9 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
           QMat_7), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
```

```
        logic matrix QMat_10 = mat_mult(mat_mult(mat_of_7x6_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 1.0), QMat_8), transpose(
            mat_of_7x6_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)));
*/
/*@
        logic matrix QMat_11 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
             QMat_9), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_12 = QMat_11;
*/
/*@
        logic matrix QMat_13 = mat_mult(mat_mult(mat_of_6x7_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 1.0), QMat_10), transpose(
            mat_of_6x7_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_14 = QMat_12;
*/
/*@
        logic matrix QMat_15 = mat_mult(mat_mult(mat_of_6x6_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0),
            QMat_13), transpose(mat_of_6x6_scalar(1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_16 = QMat_14;
*/
/*@
        logic matrix QMat_17 = mat_mult(mat_mult(mat_of_6x6_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0),
            QMat_15), transpose(mat_of_6x6_scalar(1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
```

```
/*@
        logic matrix QMat_18 = mat_mult(mat_mult(mat_of_2x1_scalar(1.0,
           -1280.0), QMat_16), transpose(mat_of_2x1_scalar(1.0, -1280
           .0)));
*/
/*@
        logic matrix QMat_19 = mat_mult(mat_mult(mat_of_6x6_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0),
           QMat_17), transpose(mat_of_6x6_scalar(1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_20 = block_m(mat_scalar_mult
           (1.0009008207386647, QMat_19), zeros(6, 2), zeros(2, 6),
           mat_scalar_mult(1111.111122222222, QMat_18));
*/
/*@
        logic matrix QMat_21 = mat_mult(mat_mult(mat_of_9x8_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0),
           QMat_20), transpose(mat_of_9x8_scalar(1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_22 = mat_mult(mat_mult(mat_of_8x9_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0),
           QMat_21), transpose(mat_of_8x9_scalar(1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
```

```
        logic matrix QMat_23 = mat_mult(mat_mult(mat_of_7x8_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 50.1, 0.0, 0.0, 0.0, 0.0, 0.0), QMat_22), transpose(
           mat_of_7x8_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 50.1, 0.0, 0.0, 0.0, 0.0, 0.0)
           ));
*/
/*@
        logic matrix QMat_24 = mat_mult(mat_mult(mat_of_8x7_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 5.0, 0.0, 0.0, 0.0), QMat_23), transpose(
           mat_of_8x7_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0.0, 0.0))
           );
*/
/*@
        logic matrix QMat_25 = mat_mult(mat_mult(mat_of_8x8_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, -1.0, -1.0), QMat_24), transpose(mat_of_8x8_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, -1.0, -1.0)));
*/
/*@
```

```
        logic matrix QMat_26 = mat_mult(mat_mult(mat_of_7x8_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01), QMat_25), transpose(
           mat_of_7x8_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01)
           ));
*/
/*@
        logic matrix QMat_27 = mat_mult(mat_mult(mat_of_7x7_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0),
            QMat_26), transpose(mat_of_7x7_scalar(1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_28 = mat_mult(mat_mult(mat_of_6x7_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 1.0), QMat_27), transpose(
           mat_of_6x7_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_29 = mat_mult(mat_mult(mat_of_4x6_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0),
           QMat_28), transpose(mat_of_4x6_scalar(1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_30 = mat_mult(mat_mult(mat_of_2x4_scalar(0.0,
           0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0), QMat_29), transpose(
           mat_of_2x4_scalar(0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0)));
*/
/*@
        requires \valid(_io_) && \valid(_state_);
        requires in_ellipsoidQ(QMat_0, vect_of_1_scalar(_io_->y -
           _io_->yd));
```

```
        requires in_ellipsoidQ(QMat_1, vect_of_2_scalar(
           _state_->Integrator_1_memory, _state_->Integrator_2_memory))
           ;
        requires \separated(_io_, _state_);
        ensures in_ellipsoidQ(QMat_1, vect_of_2_scalar(
           _state_->Integrator_1_memory, _state_->Integrator_2_memory))
           ;
        ensures \separated(_io_, _state_);
*/
void simple_olg_compute(t_simple_olg_io *_io_,  t_simple_olg_state *
   _state_) {
    REAL A11;
    REAL A12;
    REAL A21_dt;
    REAL C11;
    REAL D11;
    REAL Integrator_1;
    REAL Integrator_2;
    REAL Sum1;
    REAL Sum2;
    REAL Sum3;
    REAL Sum4;
    REAL Sum5;
    REAL control_output;
    REAL dt_;
    REAL x_t_[2];
    REAL yd_t_[1];
    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior EllipsoidInput_0:
            requires in_ellipsoidQ(QMat_0, vect_of_1_scalar(_io_->y -
               _io_->yd));
            ensures in_ellipsoidQ(QMat_3, vect_of_1_scalar(_io_->y -
               _io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));

            behavior ElllipsoidMain_0:
            requires in_ellipsoidQ(QMat_1, vect_of_2_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory));
            ensures in_ellipsoidQ(QMat_2, vect_of_3_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    Integrator_1 = _state_->Integrator_1_memory;
 }

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);
```

```
            behavior EllipsoidInput_1:
            requires in_ellipsoidQ(QMat_3, vect_of_1_scalar(_io_->y -
               _io_->yd));
            ensures in_ellipsoidQ(QMat_5, vect_of_1_scalar(_io_->y -
               _io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));

            behavior ElllipsoidMain_1:
            requires in_ellipsoidQ(QMat_2, vect_of_3_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1));
            ensures in_ellipsoidQ(QMat_4, vect_of_4_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, C11));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   C11 = 564.48 * Integrator_1;
}

   /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior EllipsoidInput_2:
            requires in_ellipsoidQ(QMat_5, vect_of_1_scalar(_io_->y -
               _io_->yd));
            ensures in_ellipsoidQ(QMat_7, vect_of_1_scalar(_io_->y -
               _io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));

            behavior ElllipsoidMain_2:
            requires in_ellipsoidQ(QMat_4, vect_of_4_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, C11));
            ensures in_ellipsoidQ(QMat_6, vect_of_5_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, C11, A21_dt)
               );
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   A21_dt = 0.01 * Integrator_1;
}

   /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior EllipsoidInput_3:
            requires in_ellipsoidQ(QMat_7, vect_of_1_scalar(_io_->y -
               _io_->yd));
```

```
        ensures in_ellipsoidQ(QMat_9, vect_of_1_scalar(_io_->y -
           _io_->yd));
        @ PROOF_TACTIC (use_strategy (Identity));

        behavior ElllipsoidMain_3:
        requires in_ellipsoidQ(QMat_6, vect_of_5_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11, A21_dt)
           );
        ensures in_ellipsoidQ(QMat_8, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11, A21_dt,
            Integrator_2));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   Integrator_2 = _state_->Integrator_2_memory;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior EllipsoidInput_4:
        requires in_ellipsoidQ(QMat_9, vect_of_1_scalar(_io_->y -
           _io_->yd));
        ensures in_ellipsoidQ(QMat_11, vect_of_1_scalar(_io_->y -
           _io_->yd));
        @ PROOF_TACTIC (use_strategy (Identity));

        behavior ElllipsoidMain_4:
        requires in_ellipsoidQ(QMat_8, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11, A21_dt,
            Integrator_2));
        ensures in_ellipsoidQ(QMat_10, vect_of_7_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11, A21_dt,
            Integrator_2, Sum3));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   Sum3 = A21_dt + Integrator_2;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior EllipsoidInput_5:
        requires in_ellipsoidQ(QMat_11, vect_of_1_scalar(_io_->y -
           _io_->yd));
        ensures in_ellipsoidQ(QMat_12, vect_of_1_scalar(simple_olg_y
            - _io_->yd));
```

```
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

        behavior ElllipsoidMain_5:
        requires in_ellipsoidQ(QMat_10, vect_of_7_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11, A21_dt,
            Integrator_2, Sum3));
        ensures in_ellipsoidQ(QMat_13, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   simple_olg_y = _io_->y;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior EllipsoidInput_6:
        requires in_ellipsoidQ(QMat_12, vect_of_1_scalar(
           simple_olg_y - _io_->yd));
        ensures in_ellipsoidQ(QMat_14, vect_of_1_scalar(simple_olg_y
            - simple_olg_yd));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

        behavior ElllipsoidMain_6:
        requires in_ellipsoidQ(QMat_13, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        ensures in_ellipsoidQ(QMat_15, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        @ PROOF_TACTIC (use_strategy (Identity));
   */
{
   simple_olg_yd = _io_->yd;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior EllipsoidInput_7:
        requires in_ellipsoidQ(QMat_14, vect_of_1_scalar(
           simple_olg_y - simple_olg_yd));
        ensures in_ellipsoidQ(QMat_16, vect_of_1_scalar(Sum4));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

        behavior ElllipsoidMain_7:
```

```
        requires in_ellipsoidQ(QMat_15, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        ensures in_ellipsoidQ(QMat_17, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        @ PROOF_TACTIC (use_strategy (Identity));
   */
{
   Sum4 = simple_olg_y - simple_olg_yd;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior EllipsoidInput_8:
        requires in_ellipsoidQ(QMat_16, vect_of_1_scalar(Sum4));
        ensures in_ellipsoidQ(QMat_18, vect_of_2_scalar(Sum4, D11));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

        behavior EllipsoidMain_8:
        requires in_ellipsoidQ(QMat_17, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        ensures in_ellipsoidQ(QMat_19, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        @ PROOF_TACTIC (use_strategy (Identity));
   */
{
   D11 = -1280.0 * Sum4;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior EllipsoidMain_9:
        requires in_ellipsoidQ(QMat_18, vect_of_2_scalar(Sum4, D11))
           ;
        requires in_ellipsoidQ(QMat_19, vect_of_6_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3));
        ensures in_ellipsoidQ(QMat_20, vect_of_8_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Integrator_1, C11,
           Integrator_2, Sum3, Sum4, D11));
        @ PROOF_TACTIC (use_strategy (SProcedure));
```

```
   */
{

}
   /*@
          requires \separated(_io_, _state_);
          ensures \separated(_io_, _state_);

          behavior ElllipsoidMain_10:
          requires in_ellipsoidQ(QMat_20, vect_of_8_scalar(
             _state_->Integrator_1_memory,
             _state_->Integrator_2_memory, Integrator_1, C11,
             Integrator_2, Sum3, Sum4, D11));
          ensures in_ellipsoidQ(QMat_21, vect_of_9_scalar(
             _state_->Integrator_1_memory,
             _state_->Integrator_2_memory, Integrator_1, C11,
             Integrator_2, Sum3, Sum4, D11, control_output));
          @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   control_output = D11 + C11;
}

   /*@
          requires \separated(_io_, _state_);
          ensures \separated(_io_, _state_);

          behavior ElllipsoidMain_11:
          requires in_ellipsoidQ(QMat_21, vect_of_9_scalar(
             _state_->Integrator_1_memory,
             _state_->Integrator_2_memory, Integrator_1, C11,
             Integrator_2, Sum3, Sum4, D11, control_output));
          ensures in_ellipsoidQ(QMat_22, vect_of_8_scalar(
             _state_->Integrator_1_memory,
             _state_->Integrator_2_memory, Integrator_1,
             Integrator_2, Sum3, Sum4, control_output, _io_->u));
          @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   _io_->u = control_output;
}

   /*@
          requires \separated(_io_, _state_);
          ensures \separated(_io_, _state_);

          @ behavior EllipsoidMain_12:
          @ requires in_ellipsoidQ(
              QMat_22, vect_of_8_scalar(_state_->Integrator_1_memory,
                                  _state_->Integrator_2_memory,
                                              Integrator_1,
                                                 Integrator_2, Sum3,
                                                 Sum4,
```

```
                                                          control_output, _io_->u))
                                                             ;
      @ ensures in_ellipsoidQ(
                     QMat_23, vect_of_7_scalar(
                        _state_->Integrator_1_memory,
                                _state_->Integrator_2_memory,
                                   Integrator_1, Integrator_2, Sum3,
                                   Sum4, A11));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    A11 = 50.1 * Integrator_1;
 }

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_13:
            requires in_ellipsoidQ(QMat_23, vect_of_7_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1,
               Integrator_2, Sum3, Sum4, A11));
            ensures in_ellipsoidQ(QMat_24, vect_of_8_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1,
               Integrator_2, Sum3, Sum4, A11, A12));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    A12 = 5.0 * Integrator_2;
 }

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_14:
            requires in_ellipsoidQ(QMat_24, vect_of_8_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1,
               Integrator_2, Sum3, Sum4, A11, A12));
            ensures in_ellipsoidQ(QMat_25, vect_of_8_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, Sum3, Sum4,
               A11, A12, Sum1));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    Sum1 = -A12 - A11;
 }

    /*@
            requires \separated(_io_, _state_);
```

```
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_15:
            requires in_ellipsoidQ(QMat_25, vect_of_8_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, Sum3, Sum4,
               A11, A12, Sum1));
            ensures in_ellipsoidQ(QMat_26, vect_of_7_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, Sum3, Sum4,
               Sum1, dt_));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    dt_ = 0.01 * Sum1;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_16:
            requires in_ellipsoidQ(QMat_26, vect_of_7_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, Sum3, Sum4,
               Sum1, dt_));
            ensures in_ellipsoidQ(QMat_27, vect_of_7_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Integrator_1, Sum3, Sum4,
               dt_, Sum2));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    Sum2 = dt_ + Integrator_1;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_17:
            @ requires in_ellipsoidQ(QMat_27,
                    vect_of_7_scalar(_state_->Integrator_1_memory,
                         _state_->Integrator_2_memory,
             Integrator_1, Sum3, Sum4, dt_, Sum2));
      @ ensures in_ellipsoidQ(QMat_28,
                    vect_of_6_scalar(_state_->Integrator_1_memory,
                                     _state_->Integrator_2_memory,
             Sum3, Sum4, Sum2, Sum5));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    Sum5 = Sum4 + Sum2;
}
```

```
    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_18:
            requires in_ellipsoidQ(QMat_28, vect_of_6_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Sum3, Sum4, Sum2, Sum5));
            ensures in_ellipsoidQ(QMat_29, vect_of_4_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Sum3, Sum5));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    _state_->Integrator_2_memory = Sum3;
 }

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_19:
            requires in_ellipsoidQ(QMat_29, vect_of_4_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Sum3, Sum5));
            ensures in_ellipsoidQ(QMat_30, vect_of_2_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    _state_->Integrator_1_memory = Sum5;
 }

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior ElllipsoidMain_20:
            requires in_ellipsoidQ(QMat_30, vect_of_2_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory));
            ensures in_ellipsoidQ(QMat_1, vect_of_2_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory));
            @ PROOF_TACTIC (use_strategy (PosDef));
    */
 {

 }
}
```

B.2 *Closed Loop Stability*

This is the fully annotated program for the system described in section 3.4.4.

```
/*
    simple_clg.c
    Generated by Gene-Auto toolset ver 2.4.10
    (launcher GALauncher)
    Generated on: 25/04/2014 09:55:50.798
    source model: simple_clg
    model version: 7.2
    last saved by:
    last saved on:
*/

/* Includes */

#include "simple_clg.h"

/* Variable definitions */

REAL simple_clg_yd = 0.0;
REAL simple_clg_y = 0.0;

/* Function definitions */

/*@
        requires \valid(_state_);
*/
void simple_clg_init(t_simple_clg_state *_state_) {
    _state_->Integrator_1_memory = 1;
    _state_->Integrator_2_memory = 1;
}

/*@
        ghost REAL Plant_0;
*/
/*@
        ghost REAL Plant_1;
*/
/*@
        ghost REAL Plant_xp_0_tmp;
*/
/*@
        ghost REAL Plant_xp_1_tmp;
*/
/*@
        logic matrix QMat_1 = mat_of_1x1_scalar(1.0);
*/
/*@
```

```
        logic matrix QMat_2 = mat_of_4x4_scalar(302.489180697473, -6
            .517333308825514, 38.42584456456831, -3488.8814244777536, -6
            .5173333088254735, 3.902945409520518, -4.233543080803806,
            26.159788880690265, 38.425844564568266, -4.233543080803765,
            38.03680122438909, -241.69936880801467, -3488.881424477755,
            26.159788880690428, -241.69936880801467, 42532.50094858428);
*/
/*@
        logic matrix QMat_3 = mat_mult(mat_mult(mat_of_5x4_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0), QMat_2), transpose(
            mat_of_5x4_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0, 0.0))
            );
*/
/*@
        logic matrix QMat_4 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_1), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_5 = mat_mult(mat_mult(mat_of_6x5_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            1.0, 0.0, 0.0, 0.0, 0.0), QMat_3), transpose(
            mat_of_6x5_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 1.0, 1.0, 0.0, 0.0, 0.0, 0.0)));
*/
/*@
        logic matrix QMat_6 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_4), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_7 = mat_mult(mat_mult(mat_of_7x6_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 564.48), QMat_5), transpose(
            mat_of_7x6_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 564.48)));
*/
/*@
        logic matrix QMat_8 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_6), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
```

```
        logic matrix QMat_9 = mat_mult(mat_mult(mat_of_8x7_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.01, 0.0), QMat_7), transpose(
           mat_of_8x7_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01, 0.0)
           ));
*/
/*@
        logic matrix QMat_10 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_8), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_11 = mat_mult(mat_mult(mat_of_9x8_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0),
           QMat_9), transpose(mat_of_9x8_scalar(1.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
           , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0)));
*/
/*@
        logic matrix QMat_12 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_10), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
```

```
        logic matrix QMat_13 = mat_mult(mat_mult(mat_of_10x9_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 1.0), QMat_11), transpose(
           mat_of_10x9_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 1.0)));
*/
/*@
        logic matrix QMat_14 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_12), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_15 = mat_mult(mat_mult(mat_of_10x10_scalar(1.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
           , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
           , 0.0, 0.0, 0.0), QMat_13), transpose(mat_of_10x10_scalar
           (1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0)));
*/
/*@
        logic matrix QMat_16 = mat_mult(mat_mult(mat_of_1x1_scalar(1.0),
            QMat_14), transpose(mat_of_1x1_scalar(1.0)));
*/
/*@
        logic matrix QMat_17 = mat_mult(mat_mult(mat_of_2x1_scalar(1.0,
           1.0), QMat_16), transpose(mat_of_2x1_scalar(1.0, 1.0)));
*/
/*@
```

```
        logic matrix QMat_18 = mat_mult(mat_mult(mat_of_9x10_scalar(1.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 1.0), QMat_15), transpose(
            mat_of_9x10_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_19 = block_m(mat_scalar_mult
            (1.0090817457114025, QMat_18), zeros(9, 2), zeros(2, 9),
            mat_scalar_mult(111.1111122222222, QMat_17));
*/
/*@
        logic matrix QMat_20 = mat_mult(mat_mult(mat_of_11x11_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, -1
            .0), QMat_19), transpose(mat_of_11x11_scalar(1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, -1.0)));
*/
/*@
```

```
        logic matrix QMat_21 = mat_mult(mat_mult(mat_of_10x11_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , -1280.0), QMat_20), transpose(mat_of_10x11_scalar(1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            -1280.0)));
*/
/*@
        logic matrix QMat_22 = mat_mult(mat_mult(mat_of_11x10_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
            , 1.0), QMat_21), transpose(mat_of_11x10_scalar(1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 1.0))
            );
*/
/*@
```

```
        logic matrix QMat_23 = mat_mult(mat_mult(mat_of_10x11_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0), QMat_22), transpose(mat_of_10x11_scalar(1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0))
            );
*/
/*@
        logic matrix QMat_24 = mat_mult(mat_mult(mat_of_10x10_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 50.1, 0.0,
            0.0, 0.0, 0.0, 0.0), QMat_23), transpose(mat_of_10x10_scalar
            (1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 50.1, 0.0
            , 0.0, 0.0, 0.0, 0.0)));
*/
/*@
```

```
        logic matrix QMat_25 = mat_mult(mat_mult(mat_of_11x10_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0.0, 0.0
            , 0.0), QMat_24), transpose(mat_of_11x10_scalar(1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 5.0, 0.0, 0.0, 0.0, 0.0))
            );
*/
/*@
        logic matrix QMat_26 = mat_mult(mat_mult(mat_of_11x11_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, -1
            .0), QMat_25), transpose(mat_of_11x11_scalar(1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, -1.0, -1.0)));
*/
/*@
```

```
        logic matrix QMat_27 = mat_mult(mat_mult(mat_of_10x11_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.01), QMat_26), transpose(mat_of_10x11_scalar(1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.01)
            ));
*/
/*@
        logic matrix QMat_28 = mat_mult(mat_mult(mat_of_10x10_scalar(1.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0
            , 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0
            , 0.0, 0.0, 1.0), QMat_27), transpose(mat_of_10x10_scalar
            (1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
             0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
             0.0, 0.0, 0.0, 1.0)));
*/
/*@
```

```
        logic matrix QMat_29 = mat_mult(mat_mult(mat_of_9x10_scalar(1.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
            0.0, 1.0, 0.0, 0.0, 1.0), QMat_28), transpose(
           mat_of_9x10_scalar(1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_30 = mat_mult(mat_mult(mat_of_7x9_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0), QMat_29), transpose(mat_of_7x9_scalar(1.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           1.0)));
*/
/*@
        logic matrix QMat_31 = mat_mult(mat_mult(mat_of_6x7_scalar(0.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0), QMat_30), transpose(
           mat_of_6x7_scalar(0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0,
           1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0)));
*/
/*@
        logic matrix QMat_32 = mat_mult(mat_mult(mat_of_4x6_scalar(1.0,
           0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0,
           0.0, 1.0, 0.01, 5.0E-5, 0.0, 0.0, 0.0, -0.01, 1.0, 0.01,
           0.0), QMat_31), transpose(mat_of_4x6_scalar(1.0, 0.0, 0.0,
           0.0, 0.0, 0.0, 0.0, 1.0, 0.0, 0.0, 0.0, 0.0, 0.0, 0.0, 1.0,
           0.01, 5.0E-5, 0.0, 0.0, 0.0, -0.01, 1.0, 0.01, 0.0)));
*/
/*@
        requires \valid(_io_) && \valid(_state_);
```

```
        requires _io_->y == 1.0 * Plant_0;
        requires in_ellipsoidQ(QMat_1, vect_of_1_scalar(_io_->yd));
        requires in_ellipsoidQ(QMat_2, vect_of_4_scalar(
           _state_->Integrator_1_memory, _state_->Integrator_2_memory,
           Plant_0, Plant_1));
        requires \separated(_io_, _state_);
        ensures in_ellipsoidQ(QMat_2, vect_of_4_scalar(
           _state_->Integrator_1_memory, _state_->Integrator_2_memory,
           Plant_0, Plant_1));
        ensures \separated(_io_, _state_);
*/
void simple_clg_compute(t_simple_clg_io *_io_,  t_simple_clg_state *
   _state_) {
    REAL A11;
    REAL A12;
    REAL A21_dt;
    REAL C11;
    REAL D11;
    REAL Integrator_1;
    REAL Integrator_2;
    REAL Sum1;
    REAL Sum2;
    REAL Sum3;
    REAL Sum4;
    REAL Sum5;
    REAL control_output;
    REAL dt_;
    REAL x_t_[4];
    REAL yd_t_[1];
    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_0:
            requires in_ellipsoidQ(QMat_2, vect_of_4_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1));
            ensures in_ellipsoidQ(QMat_3, vect_of_5_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y)
               );
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

            behavior EllipsoidInput_0:
            requires in_ellipsoidQ(QMat_1, vect_of_1_scalar(_io_->yd));
            ensures in_ellipsoidQ(QMat_4, vect_of_1_scalar(_io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));
    */
 {

 }
    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);
```

```
            behavior Plant_1:
            requires in_ellipsoidQ(QMat_3, vect_of_5_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y)
                );
            ensures in_ellipsoidQ(QMat_5, vect_of_6_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

            behavior EllipsoidInput_1:
            requires in_ellipsoidQ(QMat_4, vect_of_1_scalar(_io_->yd));
            ensures in_ellipsoidQ(QMat_6, vect_of_1_scalar(_io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));
    */
{
    Integrator_1 = _state_->Integrator_1_memory;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_2:
            requires in_ellipsoidQ(QMat_5, vect_of_6_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1));
            ensures in_ellipsoidQ(QMat_7, vect_of_7_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1, C11));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

            behavior EllipsoidInput_2:
            requires in_ellipsoidQ(QMat_6, vect_of_1_scalar(_io_->yd));
            ensures in_ellipsoidQ(QMat_8, vect_of_1_scalar(_io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));
    */
{
    C11 = 564.48 * Integrator_1;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_3:
            requires in_ellipsoidQ(QMat_7, vect_of_7_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1, C11));
```

```
        ensures in_ellipsoidQ(QMat_9, vect_of_8_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
            Integrator_1, C11, A21_dt));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

        behavior EllipsoidInput_3:
        requires in_ellipsoidQ(QMat_8, vect_of_1_scalar(_io_->yd));
        ensures in_ellipsoidQ(QMat_10, vect_of_1_scalar(_io_->yd));
        @ PROOF_TACTIC (use_strategy (Identity));
   */
{
   A21_dt = 0.01 * Integrator_1;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior Plant_4:
        requires in_ellipsoidQ(QMat_9, vect_of_8_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
            Integrator_1, C11, A21_dt));
        ensures in_ellipsoidQ(QMat_11, vect_of_9_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
            Integrator_1, C11, A21_dt, Integrator_2));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

        behavior EllipsoidInput_4:
        requires in_ellipsoidQ(QMat_10, vect_of_1_scalar(_io_->yd));
        ensures in_ellipsoidQ(QMat_12, vect_of_1_scalar(_io_->yd));
        @ PROOF_TACTIC (use_strategy (Identity));
   */
{
   Integrator_2 = _state_->Integrator_2_memory;
}

   /*@
        requires \separated(_io_, _state_);
        ensures \separated(_io_, _state_);

        behavior Plant_5:
        requires in_ellipsoidQ(QMat_11, vect_of_9_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
            Integrator_1, C11, A21_dt, Integrator_2));
        ensures in_ellipsoidQ(QMat_13, vect_of_10_scalar(
           _state_->Integrator_1_memory,
           _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
            Integrator_1, C11, A21_dt, Integrator_2, Sum3));
        @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
```

```
            behavior EllipsoidInput_5:
            requires in_ellipsoidQ(QMat_12, vect_of_1_scalar(_io_->yd));
            ensures in_ellipsoidQ(QMat_14, vect_of_1_scalar(_io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));
    */
{
    Sum3 = A21_dt + Integrator_2;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_6:
            requires in_ellipsoidQ(QMat_13, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1, C11, A21_dt, Integrator_2, Sum3));
            ensures in_ellipsoidQ(QMat_15, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1, C11, Integrator_2, Sum3, simple_clg_y));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

            behavior EllipsoidInput_6:
            requires in_ellipsoidQ(QMat_14, vect_of_1_scalar(_io_->yd));
            ensures in_ellipsoidQ(QMat_16, vect_of_1_scalar(_io_->yd));
            @ PROOF_TACTIC (use_strategy (Identity));
    */
{
    simple_clg_y = _io_->y;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_7:
            requires in_ellipsoidQ(QMat_15, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->y,
                 Integrator_1, C11, Integrator_2, Sum3, simple_clg_y));
            ensures in_ellipsoidQ(QMat_18, vect_of_9_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, C11, Integrator_2, Sum3, simple_clg_y));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));

            behavior EllipsoidInput_7:
            requires in_ellipsoidQ(QMat_16, vect_of_1_scalar(_io_->yd));
            ensures in_ellipsoidQ(QMat_17, vect_of_2_scalar(_io_->yd,
                simple_clg_yd));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
```

```
{
   simple_clg_yd = _io_->yd;
}

   /*@
           requires \separated(_io_, _state_);
           ensures \separated(_io_, _state_);

           behavior Plant_8:
           requires in_ellipsoidQ(QMat_17, vect_of_2_scalar(_io_->yd,
              simple_clg_yd));
           requires in_ellipsoidQ(QMat_18, vect_of_9_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, C11, Integrator_2, Sum3, simple_clg_y));
           ensures in_ellipsoidQ(QMat_19, vect_of_11_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, C11, Integrator_2, Sum3, simple_clg_y,
              _io_->yd, simple_clg_yd));
           @ PROOF_TACTIC (use_strategy (SProcedure));
   */
{

}
   /*@
           requires \separated(_io_, _state_);
           ensures \separated(_io_, _state_);

           behavior Plant_9:
           requires in_ellipsoidQ(QMat_19, vect_of_11_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, C11, Integrator_2, Sum3, simple_clg_y,
              _io_->yd, simple_clg_yd));
           ensures in_ellipsoidQ(QMat_20, vect_of_11_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, C11, Integrator_2, Sum3, simple_clg_y,
              simple_clg_yd, Sum4));
           @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   Sum4 = simple_clg_y - simple_clg_yd;
}

   /*@
           requires \separated(_io_, _state_);
           ensures \separated(_io_, _state_);

           behavior Plant_10:
```

```
            requires in_ellipsoidQ(QMat_20, vect_of_11_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1,
               Integrator_1, C11, Integrator_2, Sum3, simple_clg_y,
               simple_clg_yd, Sum4));
            ensures in_ellipsoidQ(QMat_21, vect_of_10_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1,
               Integrator_1, C11, Integrator_2, Sum3, Sum4, D11));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   D11 = -1280.0 * Sum4;
}

   /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_11:
            requires in_ellipsoidQ(QMat_21, vect_of_10_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1,
               Integrator_1, C11, Integrator_2, Sum3, Sum4, D11));
            ensures in_ellipsoidQ(QMat_22, vect_of_11_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1,
               Integrator_1, C11, Integrator_2, Sum3, Sum4, D11,
               control_output));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   control_output = D11 + C11;
}

   /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_12:
            requires in_ellipsoidQ(QMat_22, vect_of_11_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1,
               Integrator_1, C11, Integrator_2, Sum3, Sum4, D11,
               control_output));
            ensures in_ellipsoidQ(QMat_23, vect_of_10_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1,
               Integrator_1, Integrator_2, Sum3, Sum4, control_output,
               _io_->u));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   _io_->u = control_output;
```

```
}

   /*@
           requires \separated(_io_, _state_);
           ensures \separated(_io_, _state_);

           behavior Plant_13:
           requires in_ellipsoidQ(QMat_23, vect_of_10_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, Integrator_2, Sum3, Sum4, control_output,
              _io_->u));
           ensures in_ellipsoidQ(QMat_24, vect_of_10_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, Integrator_2, Sum3, Sum4, _io_->u, A11));
           @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   A11 = 50.1 * Integrator_1;
}

   /*@
           requires \separated(_io_, _state_);
           ensures \separated(_io_, _state_);

           behavior Plant_14:
           requires in_ellipsoidQ(QMat_24, vect_of_10_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, Integrator_2, Sum3, Sum4, _io_->u, A11));
           ensures in_ellipsoidQ(QMat_25, vect_of_11_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, Integrator_2, Sum3, Sum4, _io_->u, A11,
              A12));
           @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
   */
{
   A12 = 5.0 * Integrator_2;
}

   /*@
           requires \separated(_io_, _state_);
           ensures \separated(_io_, _state_);

           behavior Plant_15:
           requires in_ellipsoidQ(QMat_25, vect_of_11_scalar(
              _state_->Integrator_1_memory,
              _state_->Integrator_2_memory, Plant_0, Plant_1,
              Integrator_1, Integrator_2, Sum3, Sum4, _io_->u, A11,
              A12));
```

```
            ensures in_ellipsoidQ(QMat_26, vect_of_11_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, Sum3, Sum4, _io_->u, A11, A12, Sum1));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    Sum1 = -A12 - A11;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_16:
            requires in_ellipsoidQ(QMat_26, vect_of_11_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, Sum3, Sum4, _io_->u, A11, A12, Sum1));
            ensures in_ellipsoidQ(QMat_27, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, Sum3, Sum4, _io_->u, Sum1, dt_));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    dt_ = 0.01 * Sum1;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_17:
            requires in_ellipsoidQ(QMat_27, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, Sum3, Sum4, _io_->u, Sum1, dt_));
            ensures in_ellipsoidQ(QMat_28, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, Sum3, Sum4, _io_->u, dt_, Sum2));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    Sum2 = dt_ + Integrator_1;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_18:
```

```
            requires in_ellipsoidQ(QMat_28, vect_of_10_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1,
                Integrator_1, Sum3, Sum4, _io_->u, dt_, Sum2));
            ensures in_ellipsoidQ(QMat_29, vect_of_9_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, Sum3,
                Sum4, _io_->u, Sum2, Sum5));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    Sum5 = Sum4 + Sum2;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_19:
            requires in_ellipsoidQ(QMat_29, vect_of_9_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, Sum3,
                Sum4, _io_->u, Sum2, Sum5));
            ensures in_ellipsoidQ(QMat_30, vect_of_7_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, Sum3,
                _io_->u, Sum5));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    _state_->Integrator_2_memory = Sum3;
}

    /*@
            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_20:
            requires in_ellipsoidQ(QMat_30, vect_of_7_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, Sum3,
                _io_->u, Sum5));
            ensures in_ellipsoidQ(QMat_31, vect_of_6_scalar(
                _state_->Integrator_1_memory,
                _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->u,
                 Sum5));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
{
    _state_->Integrator_1_memory = Sum5;
}

    /*@
            requires \separated(_io_, _state_);
```

```
            ensures \separated(_io_, _state_);

            behavior Plant_21:
            requires in_ellipsoidQ(QMat_31, vect_of_6_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1, _io_->u,
                Sum5));
            ensures in_ellipsoidQ(QMat_32, vect_of_4_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1));
            @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
    */
 {
    /*@

            ghost Plant_xp_0_tmp = Plant_0;;
    */
    /*@

            ghost Plant_xp_1_tmp = Plant_1;;
    */
    /*@

            ghost Plant_0 = 1.0 * Plant_xp_0_tmp + 0.01 * Plant_xp_1_tmp
                + 5.0E-5 * _io_->u;;
    */
    /*@

            ghost Plant_1 = -0.01 * Plant_xp_0_tmp + 1.0 *
               Plant_xp_1_tmp + 0.01 * _io_->u;;
    */

 }
    /*@

            requires \separated(_io_, _state_);
            ensures \separated(_io_, _state_);

            behavior Plant_22:
            requires in_ellipsoidQ(QMat_32, vect_of_4_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1));
            ensures in_ellipsoidQ(QMat_2, vect_of_4_scalar(
               _state_->Integrator_1_memory,
               _state_->Integrator_2_memory, Plant_0, Plant_1));
            @ PROOF_TACTIC (use_strategy (PosDef));
    */
 {

 }
}
```

Appendix C

PVS LIBRARIES

The following is the list of PVS definitions and lemmas introduced as part of this work. All these results were proven in PVS. These libraries, as well as the proof for all theorems and lemmas, are available online[1]. Are only included here the theories this work significantly contributed to. The online library includes a few more which were borrowed (with due credit) from the Nasa linear algebra library.

C.1 Matrix definitions and common results

The following two theories introduce the matrix type and prove classic results on matrices. These theories were initiated by Heber Herencia and significantly built upon as part of this work.

```
% ---------------------------------------------------------------
%  File Name: matrices
%  Author Names: Romain Jobredeaux & Heber Herencia-Zapana
%
% ---------------------------------------------------------------
% This theory introduces the matrix type and common subtypes,
% operations, and results on matrices.

matrices: THEORY
BEGIN

IMPORTING vectors+vectors, vect_of_vect, reals+sigma_below,reals+
   sigma_swap

% A preliminary lemma for sum conversions
sigma_trick:
LEMMA
 FORALL (n:posnat, high:{high:nat|high<n},
   low:{low:nat|low<=high}, F:[below(n)->real]):
     sigma[below(n)](low,high,F) =
     sigma[nat](low,high,LAMBDA (i:nat): IF (i>=n) THEN 0 ELSE F(i)
        ENDIF) ;
```

[1] http://github.com/rjobredeaux/genecheck

```
 conversion- b2n
 n, m,e: VAR nat
 pn, pm: VAR posnat
 c, r, s: VAR real

conversion- b2n

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

Mat(m, n):
TYPE = {M: Matrix | M‘rows = m and M‘cols = n}

M, N, P: VAR Matrix

rowV(M)(i: below(M‘rows)): Vector[M‘cols] =
   lambda (j: below(M‘cols)): M‘matrix(i, j)

 colV(M)(j: below(M‘cols)): Vector[M‘rows] =
   lambda (i: below(M‘rows)): M‘matrix(i, j)

 rowM(M)(i: below(M‘rows)): Matrix =
   (# rows := 1, cols := M‘cols,
      matrix := lambda (k: below(1), j: below(M‘cols)): M‘matrix(i, j)
         #)

 colM(M)(j: below(M‘cols)): Matrix =
   (# rows := M‘rows, cols := 1,
      matrix := lambda (i: below(M‘rows), k: below(1)): M‘matrix(i, j)
         #)

 cols(M): Vector_of_Vectors[M‘rows, M‘cols] =
   lambda (i: below(M‘cols)):
     lambda (j: below(M‘rows)): M‘matrix(j, i)

 rows(M): Vector_of_Vectors[M‘cols, M‘rows] =
   lambda (i: below(M‘rows)):
     lambda (j: below(M‘cols)): M‘matrix(i, j)

 Mr2V(M: Matrix | M‘rows = 1): Vector[M‘cols] =
   lambda (j: below(M‘cols)): M‘matrix(0, j)

 Mc2V(M: Matrix | M‘cols = 1): Vector[M‘rows] =
   lambda (j: below(M‘rows)): M‘matrix(j, 0)

 vectors2matrix(pn, pm)(V2: Vector_of_Vectors[pn, pm]): Matrix =
   (# cols := pm, rows := pn,
      matrix := lambda (i: below(pn), j: below(pm)): V2(j)(i) #)

 conversion+ vectors2matrix, cols

 transpose(M): Matrix =
```

```
    (# rows := M`cols, cols := M`rows,
       matrix := lambda (i: below(M`cols), j: below(M`rows)): M`matrix(j,
            i) #)

  square?(M): bool = M`rows = M`cols

  Square: TYPE = (square?)

  squareMat?(n: posnat)(M: Square): bool = M`rows = n

  SquareMat(n: posnat): TYPE = (squareMat?(n))

  zero?(M): bool =
    FORALL (i: below(M`rows), j: below(M`cols)): M`matrix(i, j) = 0

  nonzero?(M): bool = not zero?(M)

  Zero: TYPE = (zero?)

  delta(i, j: nat): MACRO nat = if i = j then 1 else 0 endif

  identity?(M: Matrix): bool =
    square?(M) and
    FORALL (i, j: below(M`rows)):
      M`matrix(i, j) = delta(i, j)

  I(n: posnat):
  (identity?) =
    (# rows := n, cols := n,
       matrix := lambda (i, j: below(n)): delta(i, j) #)

 Zero_mat(m:posnat,n:posnat):
  Zero =
  (# rows := m, cols := n,
     matrix := lambda (i:below(m), j: below(n)): 0 #)

  upper_triangular?(M): bool = square?(M)
    AND FORALL (i, j: below(M`rows)): i > j => M`matrix(i,j) = 0

  lower_triangular?(M): bool = square?(M)
    AND FORALL (i, j: below(M`rows)): i < j => M`matrix(i,j) = 0

  diagonal?(M): bool = square?(M)
    AND FORALL (i, j: below(M`rows)): i /= j => M`matrix(i,j) = 0;

 -(M):
 Matrix =
   M WITH
     [`matrix := LAMBDA (i: below(M`rows), j: below(M`cols)): -M`matrix(i
        , j)]

  symmetric?(M): bool = square?(M) AND transpose(M) = M

  skew_symmetric?(M): bool = square?(M) AND transpose(M) = -M
```

```
 transpose2: LEMMA transpose(transpose(M)) = M

 same_dim?(M, N): macro bool = M`rows = N`rows AND M`cols = N`cols

 same_dim?(M)(N): macro bool = M`rows = N`rows AND M`cols = N`cols;

 +(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) ];

 -(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) ];

 plus_assoc: LEMMA
   FORALL (M, (N, P: (same_dim?(M)))): M + (N + P) = (M + N) + P;

 plus_comm: LEMMA
   FORALL (M, (N: (same_dim?(M)))): M + N = N + M;

 minuses: LEMMA
 FORALL (M, (N: (same_dim?(M)))): M + (-N) = M - N;

 zero_left_ident: LEMMA
   FORALL (Z: Zero, M: (same_dim?(Z))): Z + M = M;

 zero_right_ident: LEMMA
   FORALL (Z: Zero, M: (same_dim?(Z))): M + Z = M;

 *(r, M): Matrix =
    M WITH [`matrix := LAMBDA (i: below(M`rows), j: below(M`cols)):
     r * M`matrix(i, j)];

 *(M, r): Matrix = r * M;

 *(M, (N: Matrix | M`cols = N`rows)): Matrix =
   (# rows := M`rows, cols := N`cols,
      matrix := lambda (i: below(M`rows), j: below(N`cols)):
         sigma[below(M`cols)](0, M`cols-1, lambda (k: below(M`cols)):
      M`matrix(i, k) * N`matrix(k, j)) #);

 *(M, (v:Vector[M`cols])): Vector[M`rows] = lambda (i: below(M`rows)):
                  sigma[below(M`cols)](0, M`cols-1, lambda (k: below(M`
                     cols)):
                                     M`matrix(i, k) * v(k));

add_dim_row:
LEMMA
 FORALL (A:Matrix, B:Matrix | A`rows=B`rows AND A`cols = B`cols):
       (A+B)`rows = A`rows
add_dim_col:
LEMMA
```

```
 FORALL (A:Matrix, B:Matrix | A`rows=B`rows AND A`cols = B`cols):
       (A+B)`cols = A`cols

minus_scal :
LEMMA -M = (-1)*M

mult_dim:
LEMMA
 FORALL (A:Matrix, B:Matrix | A`cols=B`rows):
     (A*B)`rows=A`rows AND (A*B)`cols = B`cols

mult_dim_row: LEMMA FORALL (A:Matrix, B:Matrix | A`cols=B`rows):
     (A*B)`rows=A`rows
mult_dim_col: LEMMA FORALL (A:Matrix, B:Matrix | A`cols=B`rows):
     (A*B)`cols = B`cols

 zero_times_left: LEMMA
   FORALL (M, (Z: Zero | Z`cols = M`rows)): zero?(Z * M)

 zero_times_right: LEMMA
   FORALL (Z: Zero, (M | M`cols = Z`rows)): zero?(M * Z)

zero_neg_id : LEMMA FORALL (M:Matrix) : zero?(M-M)

zero_times_left_1: LEMMA
   FORALL (M:Matrix): Zero_mat(pn,M`rows) * M = Zero_mat(pn,M`cols)

zero_times_right_1: LEMMA
    FORALL (M:Matrix):  M * Zero_mat(M`cols,pn) = Zero_mat(M`rows,pn)

 sigma_lem: LEMMA
   FORALL (n: posnat, j: below(n), r: real):
    sigma[below(n)](0, n - 1, (lambda (k: below(n)): 0) WITH [(j) := r])
        = r

 right_mult_ident: LEMMA
   FORALL (M, (I: Matrix | identity?(I) AND M`cols = I`rows)):
               M * I = M

 left_mult_ident: LEMMA
   FORALL (M, (I: Matrix | identity?(I) AND M`rows = I`cols)):
               I * M = M

 right_mult_ident_1: LEMMA
               M * I(M`cols) = M

 left_mult_ident_1: LEMMA
               I(M`rows) * M = M

ident_vect:
LEMMA
 FORALL (I:Matrix|identity?(I)): FORALL (x:Vector[I`cols]):I*x=x

 ident_trans:LEMMA transpose(I(pn))=I(pn)
```

```
zero_trans: LEMMA FORALL (m,n:posnat): transpose(Zero_mat(m,n))=Zero_mat
   (n,m);

trans_scal : LEMMA FORALL (x:real):
     transpose(x*M) = x*transpose(M)

trans_sum : LEMMA FORALL (M,(N |same_dim?(M)(N))):
    transpose(M) + transpose(N) = transpose(N+M)

sigma_prop:
LEMMA
 FORALL (m, n: posint, f: [below(m) -> real], g: [below(m), below(n) ->
    real]):
    sigma[below(m)](0, m-1,
      lambda (i: below(m)): f(i) * sigma[below(n)](0, n-1,
  lambda (j: below(n)): g(i,j)))
     = sigma[below(m)](0, m-1, lambda (i: below(m)):
  sigma[below(n)](0, n-1, lambda(j: below(n)): f(i) * g(i,j)))

 sigma_comm: LEMMA
   FORALL (m, n: posnat, f: [below(m), below(n) -> real]):
     sigma[below(m)](0, m-1, lambda (i: below(m)):
                    sigma[below(n)](0, n-1, lambda (j: below(n)): f(i,
                       j)))
      = sigma[below(n)](0, n-1, lambda (j: below(n)):
                    sigma[below(m)](0, m-1, lambda (i: below(m)): f(i,
                       j)))

 mult_assoc: LEMMA
   FORALL (M, (N | M`cols = N`rows), (P | N`cols = P`rows)):
    M * (N * P) = (M * N) * P

right_scal_shift: LEMMA
  FORALL (M,(N | M`cols = N`rows), x:real):
   M*(x*N) = x*(M*N)

left_scal_shift: LEMMA
  FORALL (M,(N | M`cols = N`rows), x:real):
   (x*M)*N = x*(M*N)

 mult_assoc_vect:
 LEMMA
 FORALL (M, (N | M`cols = N`rows), V:Vector[N`cols]): (M * N) * V = M *
    (N * V)

left_scal_shift_vect:
LEMMA
FORALL (M:Matrix, V:Vector[M`cols], x:real): (x*M) * V = x*(M *V)

right_scal_shift_vect:
LEMMA
FORALL (M:Matrix, V:Vector[M`cols], x:real): M* (x*V) = x*(M * V)
```

```
scal_assoc :
LEMMA FORALL (M:Matrix, x,y:real) : x*(y*M) = (x*y)*M

 left_distributive:
 LEMMA
   FORALL (M, (N | M`cols = N`rows), (P: (same_dim?(N)))):
    M * (N + P) = (M * N) + (M * P)

 right_distributive:
 LEMMA
   FORALL (M, (N: (same_dim?(M))), (P | M`cols = P`rows)):
    (M + N) * P = (M * P) + (N * P)

left_distributive_add_vect:
LEMMA
 FORALL (A:Mat(m,n),B:(same_dim?(A)),v:Vector[n]): (A+B)*v = A*v+B*v;

left_distributive_min_vect:
LEMMA
 FORALL (A,B:Mat(m,n), v:Vector[n]): (A-B)*v = A*v - B*v

 left_distributive_vect:
 LEMMA
  FORALL (M, (v1, v2:Vector[M`cols])):  M * v1 - M * v2 = M * (v1 - v2)

scal_dist :
LEMMA
 FORALL (A,B:Mat(m,n), x:real): x*A+x*B = x*(A+B)

scal_dist_left : LEMMA FORALL(x,y:real,A:Mat(m,n)) : (x+y)*A = x*A + y*A

scal_dist_left_min : LEMMA FORALL(x,y:real,A:Mat(m,n)) : (x-y)*A = x*A -
    y*A

zero_scal : LEMMA 0*M = Zero_mat(M`rows,M`cols)

 transpose_product: LEMMA
   FORALL (M, (N | M`cols = N`rows)):
    transpose(M * N) = transpose(N) * transpose(M)

 trans_mat_scal:
 LEMMA
  FORALL (A:Matrix, x:Vector[A`rows], y:Vector[A`cols]):
   x * (A * y) = (transpose(A) * x) * y

trans_dim:
LEMMA
 FORALL (A:Matrix) : A`cols=transpose(A)`rows AND A`rows=transpose(A)`
    cols

trans_eq :
LEMMA
 FORALL(M, (N: (same_dim?(M)))): transpose(M)=transpose(N) IFF M = N
```

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

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

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

 inverse(M: (invertible?)): {N: Square | N‘rows = M‘rows}
   = the! (N: Square | N‘rows = M‘rows): inverse?(M)(N)
 invertible_product: lemma
   FORALL (M: (invertible?), N: (invertible?) | N‘rows = M‘rows):
     invertible?(M * N)

 product_inverse: lemma
   FORALL (M: (invertible?), N: (invertible?) | N‘rows = M‘rows):
     inverse(M * N) = inverse(N) * inverse(M)

 trace(M: Square):
 real =
   sigma[below(M‘rows)](0, M‘rows-1, lambda (k: below(M‘rows)): M‘matrix
      (k, k))

tr_symm:
 LEMMA
  FORALL (A,B : Matrix):
  A‘cols = B‘rows AND A‘rows = B‘cols IMPLIES trace(A*B) = trace(B*A)

ortho?(N:Matrix):bool= N*transpose(N)=I(N‘rows)


symmetric_part(M: Square): (symmetric?) =
  1/2 * (M + transpose(M))

skew_symmetric_part(M: Square): (skew_symmetric?) =
  1/2 * (M - transpose(M))

square_decomp: LEMMA
  FORALL (M: Square): M = symmetric_part(M) + skew_symmetric_part(M)

 square_matrix_induct: LEMMA
   forall (p: pred[Square]):
    (forall (M: Square):
      (forall (N: Square): N‘rows < M‘rows implies p(N)) implies p(M))
    implies (forall (M: Square): p(M))

zero_vect_mult:
LEMMA
 FORALL (V:Vector[pn]): Zero_mat(pm,pn)*V = vectors[pm].zero;

symmetric_scal:
LEMMA
 FORALL (a:real): symmetric?(M) IMPLIES symmetric?(a*M);
```

```
symmetric_sum:
LEMMA
 FORALL (N:(same_dim?(M))):
   symmetric?(M) AND symmetric?(N) IMPLIES symmetric?(M+N)

symmetric_prod_1: LEMMA symmetric?(M*transpose(M))
symmetric_prod_2: LEMMA symmetric?(transpose(M)*M)

concat_V(M:Matrix,N:{N:Matrix|M`cols=N`cols}):
 Matrix =
   (# rows:= M`rows+N`rows, cols:= M`cols,
      matrix := LAMBDA (i:below(M`rows+N`rows),j:below(M`cols)):
  IF (i<M`rows) THEN M`matrix(i,j) ELSE N`matrix(i-M`rows,j) ENDIF #)

concat_v_row:
LEMMA
 FORALL (M:Matrix,N:{N:Matrix|M`cols=N`cols}):
   concat_V(M,N)`rows = M`rows+N`rows;
concat_v_col:
LEMMA
 FORALL (M:Matrix,N:{N:Matrix|M`cols=N`cols}): concat_V(M,N)`cols = M`
    cols;

eye_row: LEMMA I(pn)`rows = pn
eye_col: LEMMA I(pn)`cols = pn

zero_row: LEMMA Zero_mat(pn,pm)`rows = pn
zero_col: LEMMA Zero_mat(pn,pm)`cols = pm

mult_scal_dim_row: LEMMA (c*M)`rows = M`rows
mult_scal_dim_col: LEMMA (c*M)`cols = M`cols

transp_row : LEMMA transpose(M)`rows = M`cols
transp_col : LEMMA transpose(M)`cols = M`rows

left_scal_id : LEMMA FORALL (x:real):
       M * (x * I(M`cols)) = x*M

right_scal_id : LEMMA FORALL (x:real):
       (x * I(M`rows)) * M = x*M

V2Mc(n:posnat, V: Vector[n]): Matrix =
    (# rows := n, cols := 1,

         matrix := lambda (k: below(n),j: below(1)): V(k) #)

V2Ml(n:posnat, V: Vector[n]): Matrix =
    (# rows := 1, cols := n,

         matrix := lambda (k: below(1),j: below(n)): V(j) #)

v2ml_row :LEMMA FORALL (V:Vector[pn]): V2Ml(pn,V)`rows = 1;
v2ml_col :LEMMA FORALL (V:Vector[pn]): V2Ml(pn,V)`cols = pn;
```

```
v2ml_quad :
LEMMA
 FORALL (n:posnat, MM:Mat(n,n), v:Vector[n]):
   V2Ml(n,v)*(MM*transpose(V2Ml(n,v))) = (v*(MM*v)) * I(1)
v2ml_quad_2:
LEMMA
 FORALL (n:posnat, MM:Mat(n,n), v,w:Vector[n]):
   V2Ml(n,v)*(MM*transpose(V2Ml(n,w))) = (v*(MM*w)) * I(1)

conv_vect_mat:
LEMMA
 FORALL (m,n:posnat, M:Mat(m,n), V:Vector[n]): V2Mc(m,M*V) = M*V2Mc(n,V)

v2ml_scal :
LEMMA
 FORALL (n:posnat, v:Vector[n], a:real): V2Ml(n,a*v) = a*V2Ml(n,v)
v2ml_sum:
LEMMA
 FORALL (n:posnat, v1,v2:Vector[n]) : V2Ml(n,v1+v2) = V2Ml(n,v1)+V2Ml(n,
    v2)
v2ml_dot : LEMMA FORALL (x,y:Vector[pn]): (V2Ml(pn,x)*y)(0) = x*y

AUTO_REWRITE+ zero_times_right_1
AUTO_REWRITE+ zero_times_left_1
AUTO_REWRITE+ zero_left_ident
AUTO_REWRITE+ zero_right_ident
AUTO_REWRITE+ right_mult_ident_1
AUTO_REWRITE+ left_mult_ident_1
AUTO_REWRITE+ ident_trans
AUTO_REWRITE+ zero_trans
AUTO_REWRITE+ eye_row
AUTO_REWRITE+ eye_col
AUTO_REWRITE+ zero_row
AUTO_REWRITE+ zero_col
AUTO_REWRITE+ add_dim_row
AUTO_REWRITE+ add_dim_col
AUTO_REWRITE+ mult_dim_row
AUTO_REWRITE+ mult_dim_col
AUTO_REWRITE+ mult_scal_dim_row
AUTO_REWRITE+ mult_scal_dim_col
AUTO_REWRITE+ transp_row
AUTO_REWRITE+ transp_col
AUTO_REWRITE+ concat_v_row
AUTO_REWRITE+ concat_v_col

END matrices

matrix_lemmas: THEORY

% Linear Algebra library
% Heber Herencia-Zapana NIA
% Romain Jobredeaux     Georgia Institute of Technology
% Gilberto Perez        University of Coruna Spain
```

```
% Pablo Ascariz         University of Coruna Spain
% Felicidad >Aguado      University of Coruna Spain
% Date: May, 2015

 BEGIN

 IMPORTING matrix_operator


%-----------------------------------------------------------------
%-----------------------------------------------------------------
% Matrix Lemmas
%-----------------------------------------------------------------
%-----------------------------------------------------------------


%-----------------------------------------------------------------
% Multiplying a matrix and the zero vector yields the zero vector
%-----------------------------------------------------------------

 matrix_prod_zero: LEMMA FORALL (A:Matrix): A * zero[A`cols] = zero[A`
    rows]


%-----------------------------------------------------------------
% Rows and cols of a subtraction of matrices
%-----------------------------------------------------------------

 minus_rows: LEMMA FORALL (A:Matrix,B: (same_dim?(A))): (A - B)`rows = A
    `rows

 minus_cols: LEMMA FORALL (A:Matrix,B: (same_dim?(A))): (A - B)`cols = A
    `cols


%-----------------------------------------------------------------
% Rows and cols of the transpose matrix
%-----------------------------------------------------------------

 transpose_rows: LEMMA FORALL (A: Matrix): transpose(A)`rows = A`cols

 transpose_cols: LEMMA FORALL (A: Matrix): transpose(A)`cols = A`rows

%-----------------------------------------------------------------
% Multiplying a matrix by its inverse yields the identity matrix
%-----------------------------------------------------------------

 inverse_ident: LEMMA FORALL (N: (invertible?)): inverse(N) * N = I(N`
    rows)

 ident_inverse: LEMMA FORALL (N: (invertible?)): N * (inverse(N)) = I(N`
    rows)
```

```
%------------------------------------------------------------
% Multiplying the identity matrix by a vector yields the vector
%------------------------------------------------------------

 ident_mat_prod: LEMMA FORALL (n: posnat, x: Vector[n]): matrices.I(n) *
     x = x


%------------------------------------------------------------
% Rows and cols of the inverse matrix
%------------------------------------------------------------

 inverse_rows: LEMMA FORALL (N: (invertible?)): (inverse(N))`rows = N`
    rows

 inverse_cols: LEMMA FORALL (N: (invertible?)): (inverse(N))`cols = N`
    cols

 inverse_invertible : LEMMA FORALL (N:(invertible?)) : invertible?(
    inverse(N))

 inverse2 : LEMMA FORALL(N:(invertible?)): inverse(inverse(N)) = N

 invertible_scal: LEMMA FORALL(N:(invertible?), a:nzreal): invertible?(a
    *N)

 inverse_scal: LEMMA FORALL(N:(invertible?), a:nzreal):
  inverse(a*N)= (1/a)*inverse(N)

 inv_prodvect: LEMMA FORALL (A:(invertible?), x:Vector[A`cols]) :
  A*x = zero[A`rows] IFF x = zero[A`rows]

tr_sim: LEMMA FORALL (A: Square, B: (same_dim?(A))):
  invertible?(B) IMPLIES trace(inverse(B)*A*B) = trace(A)

invertible22 : LEMMA FORALL (p11,p22,p12,p21:real, MM:Mat(2,2)):
  MM = (# cols := 2, rows :=2,
 matrix:= LAMBDA(i,j:below(2)) : COND
(i=0 AND j=0) -> p11, (i=0 AND j=1) -> p12,
(i=1 AND j=0) -> p21, (i=1 AND j=1) -> p22
ENDCOND #)AND (NOT p11*p22-p12*p21 =0) IMPLIES
 invertible?(MM);

inv22 : LEMMA FORALL (p11,p22,p12,p21:real, MM:Mat(2,2)):
      MM = (# cols := 2, rows :=2,
 matrix:= LAMBDA(i,j:below(2)) : COND
(i=0 AND j=0) -> p11, (i=0 AND j=1) -> p12,
(i=1 AND j=0) -> p21, (i=1 AND j=1) -> p22
ENDCOND #) AND (NOT p11*p22-p12*p21 =0) IMPLIES
 inverse(MM) =(1/(p11*p22-p12*p21))* (# cols := 2, rows :=2,
 matrix:= LAMBDA(i,j:below(2)) :COND
(i=0 AND j=0) -> p22, (i=0 AND j=1) -> -p12,
(i=1 AND j=0) -> -p21, (i=1 AND j=1) -> p11
ENDCOND#);
```

```
%------------------------------------------------------------------
% The transpose of the identity matrix is the identity matrix
%------------------------------------------------------------------

 transp_ident: LEMMA FORALL (n: posnat): transpose(I(n)) = I(n)



%------------------------------------------------------------------
% Transpose and inverse "commute"
%------------------------------------------------------------------
transp_invertible:
LEMMA
  FORALL (M: (invertible?)):
    invertible?(transpose(M))


transp_inv:
LEMMA
  FORALL (M: (invertible?)):
    transpose(inverse(M))=inverse((transpose(M)))

sym_inv: LEMMA FORALL (M:(invertible?)):
  symmetric?(M) IMPLIES symmetric?(inverse(M))

%------------------------------------------------------------------
% Distributive rules for matrices and vectors
%------------------------------------------------------------------

 distr_mat_vect:
LEMMA
  FORALL (A: Matrix, (B: (same_dim?(A))), y: Vector[A`cols]):
    (A + B) * y = A * y + B * y

 distr_vect_mat: LEMMA FORALL (A: Matrix, x, y: Vector[A`cols]):
    A * x + A * y = A * (x + y)



%------------------------------------------------------------------
% Technical lemmas about applying "minus" to matrices
%------------------------------------------------------------------

 matrix_sum_minus:
 LEMMA
   FORALL (A: Matrix, (B: (same_dim?(A)))): A - B = A + (-B)

 matrix_prod_minus:
 LEMMA
  FORALL (A: Matrix, x: Vector[A`cols]): -(A * x) = (-A) * x

neg_vect_mult : LEMMA FORALL (M:Matrix, (v1:Vector[M`cols])): M*(-v1) =
   -(M*v1)
```

```
END matrix_lemmas
```

C.2 Block Matrices

The following theory introduces block matrices and block vectors. Useful results are also proved. The definitions were introduced by Heber Herencia.

```
% ----------------------------------------------------------------
%  File Name: block_matrices
%  Author Names: Romain Jobredeaux & Heber Herencia-Zapana
%
% ----------------------------------------------------------------
% This theory introduces block matrices and useful results on them,
% such as rank one update inverse and proof by induction.

block_matrices: THEORY

 BEGIN

 IMPORTING matrices,reals+sigma_below_sub, reals+sigma_swap,
    matrix_lemmas

 m,n,p,q: VAR posnat

%-----------------------------------------------------------
% Definition of Block Matrices
%-----------------------------------------------------------

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

Block_Mat(m,n,p,q):
TYPE = {M: Block_Matrix | M`rows1 = m AND M`rows2 = n AND
        M`cols1 = p AND M`cols2 = q}

 M,N: VAR Block_Matrix

%-----------------------------------------------------------
% Conversions
%-----------------------------------------------------------

 Block2M(M):
 Matrix =
 (# rows := M`rows1 + M`rows2,
    cols := M`cols1 + M`cols2,
    matrix := LAMBDA (i: below(M`rows1 + M`rows2), j: below(M`cols1 + M`
       cols2)):
      M`matrix(i,j) #)
```

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

 Block2M2(M): %lower left block
Matrix =
 (# rows := M`rows2, cols := M`cols1,matrix :=
 LAMBDA (i: below(M`rows2), j: below(M`cols1)): M`matrix(i + M`rows1,j)
    #)

 Block2M3(M): %upper right block
Matrix =
 (# rows := M`rows1,
    cols := M`cols2,
    matrix := LAMBDA (i: below(M`rows1), j: below(M`cols2)):
      M`matrix(i,j + M`cols1) #)

 Block2M4(M): %lower right block
Matrix =
 (# rows := M`rows2, cols := M`cols2,matrix :=
 LAMBDA (i: below(M`rows2), j: below(M`cols2)):
    M`matrix(i + M`rows1,j + M`cols1) #)

M2Block(m,n,p,q)(A: Mat(m,p),B: Mat(n,p),C: Mat(m,q),D: Mat(n,q)):
Block_Matrix =
(# rows1 := A`rows,rows2 := B`rows,cols1 := A`cols,cols2 := C`cols,
   matrix :=
      LAMBDA (i: below(A`rows + B`rows), j: below(A`cols + C`cols)):
      IF i < A`rows THEN
     IF j < A`cols THEN A`matrix(i,j)
        ELSE C`matrix(i,j - A`cols)
     ENDIF
     ELSE IF j < A`cols THEN B`matrix (i - A`rows,j)
        ELSE D`matrix(i - A`rows,j - A`cols)
     ENDIF
      ENDIF
      #)

% (A C)
% (B D)

 conversion Block2M

access_m_1:
 LEMMA
 FORALL (A: Mat(m,p),B: Mat(n,p),C: Mat(m,q),D: Mat(n,q)):
      Block2M1(M2Block(m,n,p,q)(A,B,C,D))=A ;
access_m_2:
 LEMMA
 FORALL (A: Mat(m,p),B: Mat(n,p),C: Mat(m,q),D: Mat(n,q)):
      Block2M2(M2Block(m,n,p,q)(A,B,C,D))=B ;
access_m_3:
```

```
 LEMMA
 FORALL (A: Mat(m,p),B: Mat(n,p),C: Mat(m,q),D: Mat(n,q)):
      Block2M3(M2Block(m,n,p,q)(A,B,C,D))=C ;
access_m_4:
 LEMMA
 FORALL (A: Mat(m,p),B: Mat(n,p),C: Mat(m,q),D: Mat(n,q)):
      Block2M4(M2Block(m,n,p,q)(A,B,C,D))=D ;

%-------------------------------------------------------------
% Block vectors
%-------------------------------------------------------------

 Block_Vector: TYPE = [# comp1: posnat,comp2: posnat,
           vector: [below(comp1 + comp2) -> real] #]

 Block_Vector2: TYPE = [# comp1: posnat,comp2: posnat,
      vector1: [below(comp1) -> real],
      vector2: [below(comp2) -> real] #]

 Block_Vect(m,n): TYPE = {V: Block_Vector | V`comp1 = m AND V`comp2 = n}

 V: Var Block_Vector

 BV1toBV2(V): Block_Vector2 = (# comp1 := V`comp1, comp2 := V`comp2,
 vector1 := LAMBDA (i: below(V`comp1)): V`vector(i),
 vector2 := LAMBDA (i: below(V`comp2)): V`vector(i + V`comp1)#)

 BV2toBV1(V: Block_Vector2):
 Block_Vector = (# comp1 := V`comp1, comp2 := V`comp2,
 vector := LAMBDA (i: below(V`comp1 + V`comp2)): IF i < V`comp1
      THEN V`vector1(i) ELSE V`vector2(i - V`comp1) ENDIF #)

 conversion BV1toBV2,BV2toBV1

 Block2V(V): Vector[V`comp1 + V`comp2] =
 LAMBDA (i: below[V`comp1 + V`comp2]): V`vector(i)

 Block2V1(V): Vector[V`comp1] = LAMBDA (i: below[V`comp1]): V`vector(i)

 Block2V2(V): Vector[V`comp2] =
 LAMBDA (i: below[V`comp2]): V`vector(i + V`comp1)

V2Block(m,n:posnat)(x: Vector[m],y: Vector[n]):
Block_Vector = (# comp1 := m,comp2 := n,vector :=
      LAMBDA (i: below(m + n)):
      IF i < m THEN x(i)
      ELSE y(i - m) ENDIF
      #);

V2 : Var Block_Vector2

access_v_1:
LEMMA
 Block2V1(BV2toBV1(V2)) = V2`vector1;
```

```
access_v_2:
LEMMA
 Block2V2(BV2toBV1(V2)) = V2`vector2;

access_vb_1 :
LEMMA
 FORALL (x:Vector[m], y:Vector[n]):
  Block2V1(V2Block(m,n)(x,y)) = x;
access_vb_2 :
LEMMA
 FORALL (x:Vector[m], y:Vector[n]):
  Block2V2(V2Block(m,n)(x,y)) = y;

*(V, (W: Block_Vect(V`comp1,V`comp2))):
   real = Block2V1(V)*Block2V1(W) + Block2V2(V)*Block2V2(W)

conversion Block2V

%-----------------------------------------------------------
% Operations with Block Matrices
%-----------------------------------------------------------

 Btranspose(M): Block_Matrix =  (# rows1 := M`cols1,rows2 := M`cols2,
        cols1 := M`rows1,cols2 := M`rows2,
        matrix :=
 LAMBDA (i: below(M`cols1 + M`cols2), j: below(M`rows1 + M`rows2)):
        M`matrix(j,i) #)

 conv_transp:
LEMMA
 FORALL (M: Block_Matrix):
Btranspose(M) = M2Block(M`cols1,M`cols2,M`rows1,M`rows2)(transpose(
   Block2M1(M)),
 transpose(Block2M3(M)),transpose(Block2M2(M)),transpose(Block2M4(M)))

 Bsquare?(M): bool = square?(M)

 Bdiag_square?(M): bool = square?(Block2M1(M)) AND square?(Block2M4(M))

 block_square:
LEMMA
 FORALL (M: Block_Matrix):
 Bdiag_square?(M) IMPLIES Bsquare?(M)

 trans_conv:
LEMMA
 FORALL (M: Block_Matrix):
 transpose(Block2M(M)) = (Block2M(Btranspose(M)));

 Bsymmetric?(M): bool = symmetric?(M)

 block_symmetric:
LEMMA
```

```
 FORALL (M: Block_Matrix):
 Bsymmetric?(M) AND M`cols1 = M`rows1 AND M`cols2 = M`rows2
 IFF
 symmetric?(Block2M1(M)) AND symmetric?(Block2M4(M)) AND
 transpose(Block2M2(M)) = Block2M3(M);

 transpose_eq:
LEMMA
 FORALL (M:Block_Matrix):
 Btranspose(M) =transpose(Block2M(M));

 *(M: Block_Matrix,V: Block_Vect(M`cols1,M`cols2)):
 Block_Vector2 = (# comp1 := M`rows1, comp2 := M`rows2,
      vector1 := Block2M1(M)*Block2V1(V) + Block2M3(M)*Block2V2(V),
      vector2 := Block2M2(M)*Block2V1(V) + Block2M4(M)*Block2V2(V)#)

 same_Bdim?(M)(N): bool = M`rows1 = N`rows1 AND M`rows2 = N`rows2 AND
       M`cols1 = N`cols1 AND M`cols2 = N`cols2;

 +(M: Block_Matrix,N: (same_Bdim?(M))):
 Block_Matrix =
 (# rows1 := M`rows1,rows2 := M`rows2,
    cols1 := M`cols1 ,cols2 := M`cols2,
    matrix := LAMBDA (i: below(M`rows1 + M`rows2),j: below(M`cols1 + M`
       cols2)):
      M`matrix(i,j) + N`matrix(i,j) #);

 *(r: real, M: Block_Matrix):
 Block_Matrix = M WITH
 [`matrix := LAMBDA (i: below(M`rows1 + M`rows2), j: below(M`cols1 + M`
    cols2)):
     r * M`matrix(i, j)];

 *(M: Block_Matrix, (v: Vector[M`cols1 + M`cols2])):
 Vector[M`rows1 + M`rows2] = LAMBDA (i: below(M`rows1 + M`rows2)):
   sigma(0, M`cols1 + M`cols2 - 1,
     LAMBDA (j: below(M`cols1 + M`cols2)): M`matrix(i, j) * v(j));

 *(M:Block_Matrix,N:{N:Block_Matrix|N`rows1=M`cols1 AND N`rows2=M`cols2
    }):
 Block_Matrix =
 M2Block(M`rows1,M`rows2,N`cols1,N`cols2)(
   Block2M1(M)*Block2M1(N) + Block2M3(M)*Block2M2(N),
   Block2M2(M)*Block2M1(N) +Block2M4(M)*Block2M2(N),
   Block2M1(M)*Block2M3(N) + Block2M3(M)*Block2M4(N),
   Block2M2(M)*Block2M3(N) +Block2M4(M)*Block2M4(N));

conv_mult:
LEMMA
 FORALL (m,n,p,q,r,s:posnat, A:Mat(m,p), B:Mat(n,p), C:Mat(m,q),
   D:Mat(n,q), E:Mat(p,r), F:Mat(q,r), G:Mat(p,s), H:Mat(q,s)):
   Block2M(M2Block(m,n,p,q)(A,B,C,D))*Block2M(M2Block(p,q,r,s)(E,F,G,H))
        =
   Block2M(M2Block(m,n,r,s)(A*E+C*F, B*E+D*F, A*G+C*H,B*G+D*H));
```

```
conv_sum:
LEMMA
 FORALL (m,n,p,q:posnat, A:Mat(m,p), B:Mat(n,p), C:Mat(m,q),
   D:Mat(n,q), E:Mat(m,p), F:Mat(n,p), G:Mat(m,q), H:Mat(n,q)):
   Block2M(M2Block(m,n,p,q)(A,B,C,D))+Block2M(M2Block(m,n,p,q)(E,F,G,H))
        =
   Block2M(M2Block(m,n,p,q)(A+E, B+F, C+G,D+H));

conv_scal :
LEMMA
 FORALL (m,n,p,q:posnat, A:Mat(m,p), B:Mat(n,p), C:Mat(m,q),
    D:Mat(n,q), a:real):
    a*Block2M(M2Block(m,n,p,q)(A,B,C,D)) =
    Block2M(M2Block(m,n,p,q)(a*A,a*B,a*C,a*D))

eq_block:
LEMMA
 FORALL (m,n,p,q:posnat, A:Mat(m,p), B:Mat(n,p), C:Mat(m,q),
    D:Mat(n,q), E:Mat(m,p), F:Mat(n,p), G:Mat(m,q), H:Mat(n,q)):
    A=E AND B=F AND C=G AND D=H IMPLIES
      Block2M(M2Block(m,n,p,q)(A,B,C,D)) =
        Block2M(M2Block(m,n,p,q)(E,F,G,H));

block_mult_comm:

LEMMA


FORALL (M:(Bdiag_square?), V:Block_Vect(M`cols1,M`cols2)):
  Block2M(M)*Block2V(V) = Block2V(M*V);

block_v_mult_comm:
LEMMA
 FORALL (V1,V2:Block_Vect(m,n)): Block2V(V1)*Block2V(V2) = V1*V2;

block_v_mult_comm2:
LEMMA
 FORALL (M:Block_Matrix, VV:Block_Vect(M`cols1,M`cols2)):
 Block2M(M)*Block2V(VV) = Block2V(M*VV)

 split_vect:
LEMMA
 FORALL (n:{n:posnat|n>=2},z:Vector[n],m:{m:posnat|m<n}):
   EXISTS (x:Vector[m],y:Vector[n-m]):
     z = Block2V(BV2toBV1((# comp1:=m ,comp2:=n-m,vector1:=x, vector2:=y
         #)));

id_block:
LEMMA
 FORALL (n,m:posnat):
   I(n+m) = Block2M(M2Block(n,m,n,m)(I(n),Zero_mat(m,n),Zero_mat(n,m),I(
      m)))

block_induct_1:
```

```
LEMMA
 FORALL (p:pred[Square]):
   (FORALL (MM:Mat(1,1)): p(MM)) AND
   (FORALL (n:posnat):
     (FORALL (MM:Mat(n,n)): p(MM))
     IMPLIES
       (FORALL (NN:Mat(n,n), a:real,b,c:Vector[n]):
  p(Block2M(M2Block(1,n,1,n)(a*I(1),transpose(V2Ml(n,b)),V2Ml(n,c),NN)))
     ))
     IMPLIES FORALL(MM:Square): p(MM)

 AUTO_REWRITE+ access_m_1
 AUTO_REWRITE+ access_m_2
 AUTO_REWRITE+ access_m_3
 AUTO_REWRITE+ access_m_4
 AUTO_REWRITE+ access_v_1
 AUTO_REWRITE+ access_v_2
 AUTO_REWRITE+ block_mult_comm
 AUTO_REWRITE+ conv_mult
 AUTO_REWRITE+ block_v_mult_comm

 block_invertible :
LEMMA
 FORALL (NN:Mat(n,n), c:real,b:Vector[n]):
  invertible?(NN) IMPLIES
    (c- (b*(inverse(NN)*b))) /=0 IMPLIES
      invertible?(Block2M(M2Block(1,n,1,n)(
  c*I(1),transpose(V2Ml(n,b)),
  V2Ml(n,b),NN)))

END block_matrices
```

C.3 *Quadratic forms and positive definite matrices*

The following theory introduces the concept of positive definiteness and semidefiniteness for matrices. Useful results are proven. Of particular interest, the Schur complement formula, and the Cholesky factorization existence are proven in this theory.

```
% ---------------------------------------------------------------
%  File Name: posdef
%  Author Names: Romain Jobredeaux
%
% ----------------------------------------------------------------
% This theory proves results on quadratic forms defined by a
% matrix, and results on positive-definite, and semi positive-
% definite matrices. Particular results of interest include
% the existence of a Cholesky decomposition for semi positive-
% definite matrices, as well as results involving Schur's
% complement.
```

```
posdef:THEORY

BEGIN

IMPORTING matrices,reals+sigma_below_sub, reals+sigma_swap,
matrix_lemmas, block_matrices

% Some results on quadratic forms

symetric_qua_trans:
LEMMA
  FORALL(A:(square?),M:{M:Matrix|M`cols=A`rows}):
    symmetric?(A)IMPLIES symmetric?(M*A*transpose(M))

skewsym_quad_zero :
LEMMA
  FORALL(A:(skew_symmetric?), x:Vector[A`cols]):
    x*(A*x) = 0

quad_scal:
LEMMA
  FORALL(A:(square?), a:real,x:Vector[A`cols]): x*((a*A)*x) =a*(x*(A*x))

quad_sum:
LEMMA
  FORALL (A:(square?),B:(same_dim?(A)), x:Vector[A`cols]):
    x*((A+B)*x) = x*(A*x)+x*(B*x)

sym_block_quad_expr :
LEMMA
  FORALL (m,n:posnat, A:Mat(m,m), B:Mat(n,m), D:Mat(n,n),
    x:Vector[m], y:Vector[n], G: Mat(m+n,m+n), z:Vector[n+m]):
        z = Block2V(V2Block(m,n)(x,y)) AND
        G = Block2M(M2Block(m,n,m,n)(A,B,transpose(B),D)) IMPLIES
        z*(G*z) = x*(A*x) + 2*(y*(B*x)) + y*(D*y)

%------------
% Definitions
%------------

semidef_pos?(A: (square?)): bool = FORALL (x: Vector[A`rows]): x*(A*x)
   >= 0

def_pos?(A: (square?)):
 bool = FORALL (x: Vector[A`rows]): (NOT x=zero[A`rows]) IMPLIES x*(A*x)
     > 0

def_pos_id: LEMMA FORALL (n:posnat) : def_pos?(I(n))

semidef_pos_sym :
LEMMA
  FORALL (A:(square?)): semidef_pos?(A) iff semidef_pos?(symmetric_part(
     A))
```

```
def_pos_semidef_pos :
LEMMA FORALL (A:(square?)): def_pos?(A) IMPLIES semidef_pos?(A)

posdef_inv :
THEOREM
  FORALL (n:posnat,P:SquareMat(n)):
    invertible?(P) AND symmetric?(P)
    IMPLIES (def_pos?(P) IMPLIES def_pos?(inverse(P)))

posdef_inveq:
THEOREM
  FORALL (n:posnat,P:SquareMat(n)):
    invertible?(P) AND symmetric?(P)
    IMPLIES (def_pos?(P) IFF def_pos?(inverse(P)))

semidef_qua_trans:
LEMMA
  FORALL(A:(square?),M:{M:Matrix|M`cols=A`rows}):
    semidef_pos?(A) IMPLIES semidef_pos?(M*A*transpose(M))

semidef_sum:
LEMMA
  FORALL (A:(square?), B:(same_dim?(A))):
  semidef_pos?(A) AND semidef_pos?(B) IMPLIES semidef_pos?(A+B);

semidef_scal: LEMMA FORALL (a:nnreal, A:(semidef_pos?)): semidef_pos?(a*
   A);
defpos_scal : LEMMA FORALL (a:posreal, A:(def_pos?)): def_pos?(a*A);

block_semidef:
LEMMA
  FORALL (M:(Bdiag_square?)):
    semidef_pos?(Block2M(M)) IMPLIES
      semidef_pos?(Block2M1(M)) AND semidef_pos?(Block2M4(M))

diag_block_semidef: LEMMA FORALL (m,n:posnat, A:Mat(m,m), B:Mat(n,n)):
        semidef_pos?(A) AND semidef_pos?(B) IMPLIES
        semidef_pos?(Block2M(M2Block(m,n,m,n)
      (A,Zero_mat(n,m),Zero_mat(m,n),
       B)))

chol_step1 :
LEMMA
  FORALL(n:posnat,alpha:real, v:Vector[n], B:Mat(n,n)) :
    alpha=0 AND
    semidef_pos?(Block2M(M2Block(1,n,1,n)(
      alpha*I(1), transpose(V2Ml(n,v)),V2Ml(n,v),B)))
    IMPLIES v= zero[n]

chol_step2 :
LEMMA
  FORALL (n:posnat, alpha:nzreal, v:Vector[n], B:Mat(n,n)):
    LET C = B - (1/alpha)*(transpose(V2Ml(n,v))*V2Ml(n,v)) in
    semidef_pos?(Block2M(M2Block(1,n,1,n)(
```

```
      alpha*I(1), transpose(V2Ml(n,v)),V2Ml(n,v),B)))
    IMPLIES semidef_pos?(C)

cholesky_semidef:
LEMMA
  FORALL (MM:(square?)):
    symmetric?(MM) AND semidef_pos?(MM)
    IMPLIES
    EXISTS (R:Mat(MM`rows,MM`cols)): MM = transpose(R)*R

cholesky_rev :
LEMMA
  FORALL (MM:(square?)):
    (EXISTS (R:Mat(MM`rows,MM`cols)): MM = transpose(R)*R)
    IMPLIES
    symmetric?(MM) AND semidef_pos?(MM)

semidefpos_inv_defpos:
LEMMA
  FORALL (A:(invertible?)):
    symmetric?(A) AND semidef_pos?(A) IMPLIES def_pos?(A)

def_pos_sym :
LEMMA
  FORALL (A:(square?)): def_pos?(A) iff def_pos?(symmetric_part(A))

schur_semidef_1:
LEMMA
  FORALL (n,m:posnat, A:Mat(n,n),B:Mat(n,m),D:Mat(m,m)):
    invertible?(A) AND symmetric?(A) AND symmetric?(D)
    IMPLIES
      (semidef_pos?(Block2M(M2Block(n,m,n,m)(A,transpose(B),B,D)))
      IFF
      def_pos?(A) AND semidef_pos?(D-transpose(B)*inverse(A)*B))

schur_semidef_2 :
LEMMA
  FORALL (n,m:posnat, A:Mat(n,n),B:Mat(n,m),D:Mat(m,m)):
    invertible?(D) AND symmetric?(A) AND symmetric?(D)
    IMPLIES
      (semidef_pos?(Block2M(M2Block(n,m,n,m)(A,transpose(B),B,D)))
      IFF
      def_pos?(D) AND semidef_pos?(A-B*inverse(D)*transpose(B)))

IMPORTING reals+quadratic

posdef22 : LEMMA FORALL (p11,p22:posreal, p12:real, MM:Mat(2,2)):
   MM = (# cols := 2, rows :=2,
 matrix:= LAMBDA(i,j:below(2)) : COND
(i=0 AND j=0) -> p11, (i=0 AND j=1) -> p12,
(i=1 AND j=0) -> p12, (i=1 AND j=1) -> p22
ENDCOND #) AND p11*p22-p12^2 > 0 IMPLIES
   def_pos?(MM);
```

```
END posdef
```

C.4 Ellipsoids and control theory results

This theory contains the fundamental ellipsoid combination results discussed in chapter 4

```
% -------------------------------------------------------------
%  File Name: ellipsoid
%  Author Names: Romain Jobredeaux & Heber Herencia-Zapana
%
% --------------------------------------------------------------
% This PVS theory defines the notion of ellipsoid (centered at the
   origin),
% both in the classic way (x^T P x <= 1) and the "Schur complement way":
% (   1 x^T   )
% (           ) >= 0, referred to with the predicate "in_ellipsoid_Q?"
% (   x P^-1 )
%
% Theorems are then introduced that describe how a linear transformation
% affects an ellipsoid
ellipsoid: theory

begin

importing matrix_operator, matrices, block_matrices, posdef

N:var Matrix
n:var posnat


Vector_no_param: TYPE = [# length: posnat, vect: vectors[length].Vector
   #]

-(x: Vector_no_param,y: Vector_no_param | x`length=y`length) :
  Vector_no_param = (# length:=x`length, vect:=x`vect-y`vect #);

*(P: Matrix, c: Vector_no_param | P`cols=c`length) :
  Vector_no_param = (# length:= P`rows, vect:= P*c`vect #);

in_ellipsoid_P?(n:posnat, P:SquareMat(n), x:Vector[n]): bool =
  def_pos?(P) AND symmetric?(P) AND x*(P*x)<=1

in_ellipsoid_Q?(n:posnat, Q:SquareMat(n), x:Vector[n]): bool =
  semidef_pos?(Q) AND symmetric?(Q) AND
  semidef_pos?(Block2M(M2Block(1,n,1,n)(I(1),transpose(V2Ml(n,x)),V2Ml(n
     ,x),Q)))

ellipsoid: LEMMA
forall (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

ellipsoid_simp: LEMMA
forall (n:posnat, Q, M: SquareMat(n), x, y: Vector[n]):
bijective?(n)(T(n,n)(Q)) AND bijective?(n)(T(n,n)(M))
AND x*(inv(n)(Q)*x)<=1
AND y=M*x
IMPLIES
y*(inv(n)(M*(Q*transpose(M)))*(y))<=1

ellipsoid_general: THEOREM
FORALL (n:posnat,m:posnat,Q:SquareMat(n),M:Mat(m,n),x:Vector[n],y:Vector
   [m]):
in_ellipsoid_Q?(n,Q,x)
AND y = M*x
IMPLIES
in_ellipsoid_Q?(m,M*Q*transpose(M),y)

 AUTO_REWRITE+ v2ml_row
 AUTO_REWRITE+ v2ml_col

ellipsoid_equivalence: THEOREM FORALL (n:posnat, P:SquareMat(n), x:
   Vector[n]):
  invertible?(P) IMPLIES
  (in_ellipsoid_Q?(n,inverse(P),x) IFF in_ellipsoid_P?(n,P,x))

ellipsoid_combination: THEOREM
 forall (n,m:posnat, lambda_1, lambda_2: posreal,
 Q_1: Mat(n,n), Q_2: Mat(m,m), x:Vector[n],
 y:Vector[m], z:Vector[m+n]):
   in_ellipsoid_Q?(n,Q_1,x)
   AND in_ellipsoid_Q?(m,Q_2,y)
   AND lambda_1+ lambda_2 = 1
   AND z = Block2V(V2Block(n,m)(x,y))
   IMPLIES in_ellipsoid_Q?(n+m,
   Block2M(M2Block(n,m,n,m)(1/lambda_1*Q_1,Zero_mat(m,n),
   Zero_mat(n,m), 1/lambda_2*Q_2)),z)


step_ellipsoid_combination_plus: THEOREM
 forall (n:posnat, Q: Mat(n,n), x: Vector[n], mu: posreal):
     mu>=1 AND in_ellipsoid_Q?(n,Q,x) IMPLIES in_ellipsoid_Q?(n,mu*Q,x)

ellipsoid_combination_plus: THEOREM
 forall (n,m:posnat, lambda_1, lambda_2: posreal,
  Q_1: Mat(n,n), Q_2: Mat(m,m), x:Vector[n],
  y:Vector[m], z:Vector[m+n]):
   in_ellipsoid_Q?(n,Q_1,x)
   AND in_ellipsoid_Q?(m,Q_2,y)
   AND lambda_1+ lambda_2 <= 1
   AND z = Block2V(V2Block(n,m)(x,y))
   IMPLIES in_ellipsoid_Q?(n+m,
   Block2M(M2Block(n,m,n,m)(1/lambda_1*Q_1,Zero_mat(m,n),
```

```
   Zero_mat(n,m),1/lambda_2*Q_2)),z)

ellipsoid_general_2 : THEOREM
 forall (n:posnat,m:posnat, Q:SquareMat(n), M: Mat(m,n),
  x:Vector[n], y:Vector[m],c:Vector[n]):
   in_ellipsoid_Q?(n,Q,x-c)
   AND y = M*x
   IMPLIES in_ellipsoid_Q?(m,M*Q*transpose(M),y-M*c)

in_ellipsoid_Q2?(n:posnat, Q:SquareMat(n), x:Vector[n],lam:nnreal): bool
    =
   semidef_pos?(Q) AND symmetric?(Q) AND
   semidef_pos?(Block2M(M2Block(1,n,1,n)(lam*I(1),transpose(V2Ml(n,x)),
           V2Ml(n,x),Q)))

in_ellipsoid_P2?(n:posnat, P:SquareMat(n), x:Vector[n],lam:nnreal): bool
    =
    def_pos?(P) AND symmetric?(P) AND x*(P*x)<=lam

eq_ellq_ellq2 : lemma
  forall (n:posnat, Q:SquareMat(n), x:Vector[n],lam:posreal):
   in_ellipsoid_Q2?(n, Q, x,lam) IFF in_ellipsoid_Q?(n, lam*Q, x)

eq_ellp_ellp2 : lemma
  forall (n:posnat, P:SquareMat(n), x:Vector[n],lam:posreal):
   in_ellipsoid_P2?(n, P, x,lam) IFF in_ellipsoid_P?(n, (1/lam)*P, x)

ellP2_zero: LEMMA FORALL (n:posnat, P:SquareMat(n),x:Vector[n]):
  in_ellipsoid_P2?(n,P,x,0) IFF (x=zero[n] AND symmetric?(P) AND
         def_pos?(P))

ellQ2_zero: LEMMA FORALL(n:posnat, Q:SquareMat(n),x:Vector[n]):
      in_ellipsoid_Q2?(n,Q,x,0) IFF (x=zero[n] AND symmetric?(Q) AND
             semidef_pos?(Q))

ellipsoid_equivalence_2: THEOREM FORALL (n:posnat, P:SquareMat(n),
  x:Vector[n], lam:nnreal):
    invertible?(P)
    IMPLIES
    (in_ellipsoid_Q2?(n,inverse(P),x,lam) IFF in_ellipsoid_P2?(n,P,x,lam
       ))


ellipsoid_general_3 : THEOREM
 forall (n:posnat,m:posnat, Q:SquareMat(n), M: Mat(m,n),
         x:Vector[n], y:Vector[m],c:Vector[n],lam:nnreal):
    in_ellipsoid_Q2?(n,Q,x-c,lam)
    AND y = M*x
    IMPLIES in_ellipsoid_Q2?(m,M*Q*transpose(M),y-M*c,lam)

ellipsoid_general_4 : THEOREM
 forall (n:posnat,m:posnat, Q:SquareMat(n), M: Mat(m,n),
   x:Vector[n], y,b:Vector[m],c:Vector[n],lam:nnreal):
    in_ellipsoid_Q2?(n,Q,x-c,lam)
```

```
    AND y = M*x+b
    IMPLIES in_ellipsoid_Q2?(m,M*Q*transpose(M),y-(M*c+b),lam)

ellipsoid_bound : THEOREM
 forall(n:posnat, Q:SquareMat(n), c,x:Vector[n]):
   in_ellipsoid_Q?(n,Q,x) IMPLIES abs(c*x)<= sqrt(c*(Q*c))

sat(x:real): real = TABLE
  %------+----++
  | x < -1| -1 ||
  %------+----++
  | x > 1 | 1  ||
  %------+----++
  | ELSE  | x  ||
  %------+----++
ENDTABLE

sat_id_comp : LEMMA FORALL (x:real) : (x<=0 IMPLIES sat(x)>=x) AND
              (x>=0 IMPLIES sat(x)<=x)

sat_lin_com : LEMMA FORALL (x:real, d:posreal) : (abs(x)<= d) AND d>=1
    IMPLIES
       (x<=0 IMPLIES sat(x)<=(1/d) * x) AND
       (x>=0 IMPLIES sat(x)>=(1/d) * x)

sat_sect_bound_step: LEMMA FORALL (n:posnat, Q:SquareMat(n),
  c,x: Vector[n], d:posreal):
    in_ellipsoid_Q?(n,Q,x) AND d >= sqrt(c*(Q*c))
    IMPLIES
    (sat(c*x)-(c*x))*(sat(c*x)-(1/d)*(c*x))<=0


block_rank_1: LEMMA FORALL (m,n:posnat, x,z:Vector[m],y,t:Vector[n]):
        LET  a = Block2V(V2Block(m,n)(x,y)) IN
        LET  b = Block2V(V2Block(m,n)(z,t)) IN
        transpose(V2Ml(n+m,a))*V2Ml(n+m,b) =
        Block2M(M2Block(m,n,m,n)(transpose(V2Ml(m,x))*V2Ml(m,z),
               transpose(V2Ml(n,y))*V2Ml(m,z),
               transpose(V2Ml(m,x))*V2Ml(n,t),
               transpose(V2Ml(n,y))*V2Ml(n,t)))


block_sym_rank_1 : LEMMA FORALL (n:posnat, c:Vector[n], d:posreal):
         LET a = Block2V(V2Block(n,1)(-c,LAMBDA (i:below(1)): 1)) IN
         LET b = Block2V(V2Block(n,1)((-1/d)*c,LAMBDA (i:below(1)): 1))
            IN
         LET G = 1/2*(transpose(V2Ml(n+1,a))*V2Ml(n+1,b) +
          transpose(V2Ml(n+1,b))*V2Ml(n+1,a))
         IN
         G = Block2M(M2Block(n,1,n,1)(
      (1/d)*transpose(V2Ml(n,c))*V2Ml(n,c), 1/2*(-1-1/d)*V2Ml(n,c),
      1/2*(-1-1/d)*transpose(V2Ml(n,c)), I(1)))
```

```
sat_sect_bound : THEOREM FORALL (n:posnat, Q:SquareMat(n),
       c,x:Vector[n], d:posreal):
         LET z = Block2V(V2Block(n,1)(x,LAMBDA (i:below(1)): sat(c*x)))
            IN
         LET a = Block2V(V2Block(n,1)(-c,LAMBDA (i:below(1)): 1)) IN
         LET b = Block2V(V2Block(n,1)((-1/d)*c,LAMBDA (i:below(1)): 1))
            IN
         LET G = 1/2*(transpose(V2Ml(n+1,a))*V2Ml(n+1,b) +
          transpose(V2Ml(n+1,b))*V2Ml(n+1,a)) IN
           in_ellipsoid_Q?(n,Q,x) AND
         d >= sqrt(c*(Q*c))
         IMPLIES z*(G*z) <= 0

sect_bound_quad: THEOREM FORALL (n:posnat, d1,d2:posreal,f:[real->real],
    x:real):
     LET v = Block2V(V2Block(1,1)(LAMBDA (i:below(1)): x,
                LAMBDA (i:below(1)): f(x))) IN
     LET g = (# cols :=2, rows:=2,
       matrix:= LAMBDA(i,j:below(2)):
    COND
    (i=0 AND j=0) -> 1/(d1*d2), (i=0 AND j=1) -> -(d1+d2)/(2*d1*d2),
    (i=1 AND j=0) -> -(d1+d2)/(2*d1*d2), (i=1 AND j=1) -> 1
    ENDCOND #) IN
       (f(x)-(1/d1)*x)*(f(x)-(1/d2)*x)<=0 IFF
       v*(g*v)<=0

sect_bound_ell : THEOREM FORALL (n:posnat, Q:SquareMat(n), c,x:Vector[n
    ],
  d1,d2,mu:posreal, f:[real->real]):
  LET d = c*(Q*c) IN
  LET g = (# cols :=2, rows:=2,
  matrix:= LAMBDA(i,j:below(2)) :
    COND
    (i=0 AND j=0) -> 1/(d1*d2), (i=0 AND j=1) -> -(d1+d2)/(2*d1*d2),
    (i=1 AND j=0) -> -(d1+d2)/(2*d1*d2), (i=1 AND j=1) -> 1
    ENDCOND #) IN
  LET v = Block2V(V2Block(1,1)(LAMBDA (i:below(1)): c*x,
              LAMBDA (i:below(1)): f(c*x))) IN
  LET D = 4*(d1*d2)^2-mu*d*(d1-d2)^2 IN
  in_ellipsoid_Q?(n,Q,x) AND
  D>0 AND
  (f(c*x)-(1/d1)*(c*x))*(f(c*x)-(1/d2)*(c*x))<=0 AND
  d>0
  IMPLIES
  LET q =(# cols :=2, rows:=2,
    matrix:= LAMBDA(i,j:below(2)) :
    COND
    (i=0 AND j=0) -> 1/d, (i=0 AND j=1) -> 0,
    (i=1 AND j=0) -> 0, (i=1 AND j=1) -> 0
    ENDCOND #) IN
  in_ellipsoid_P2?(2,q+mu*g,v,(c*x)^2/(c*(Q*c)))

sect_bound_invertible:THEOREM FORALL(d,d1,d2,mu:posreal):
```

```
  LET g = (# cols :=2, rows:=2,
    matrix:= LAMBDA(i,j:below(2)) :
    COND
    (i=0 AND j=0) -> 1/(d1*d2), (i=0 AND j=1) -> -(d1+d2)/(2*d1*d2),
    (i=1 AND j=0) -> -(d1+d2)/(2*d1*d2), (i=1 AND j=1) -> 1
    ENDCOND #) IN
  LET q =(# cols :=2, rows:=2,
    matrix:= LAMBDA(i,j:below(2)) :
    COND
    (i=0 AND j=0) -> 1/d, (i=0 AND j=1) -> 0,
    (i=1 AND j=0) -> 0, (i=1 AND j=1) -> 0
    ENDCOND #) IN
  LET D = 4*(d1*d2)^2-mu*d*(d1-d2)^2 IN
  D>0 IMPLIES invertible?(q+mu*g)

sect_bound_inv: THEOREM FORALL(d,d1,d2,mu:posreal):
  LET g = (# cols :=2, rows:=2,
    matrix:= LAMBDA(i,j:below(2)) :
    COND
    (i=0 AND j=0) -> 1/(d1*d2), (i=0 AND j=1) -> -(d1+d2)/(2*d1*d2),
    (i=1 AND j=0) -> -(d1+d2)/(2*d1*d2), (i=1 AND j=1) -> 1
    ENDCOND #) IN
  LET q =(# cols :=2, rows:=2,
    matrix:= LAMBDA(i,j:below(2)) :
    COND
    (i=0 AND j=0) -> 1/d, (i=0 AND j=1) -> 0,
    (i=1 AND j=0) -> 0, (i=1 AND j=1) -> 0
    ENDCOND #) IN
  LET D = 4*(d1*d2)^2-mu*d*(d1-d2)^2 IN
  D>0 IMPLIES
  LET h = 1/D *(# cols :=2, rows:=2,
    matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> sq(2*d1*d2)*d,       (i=0 AND j=1) -> (2*d*d1*d2)*(d1+
    d2),
(i=1 AND j=0) -> (2*d*d1*d2)*(d1+d2), (i=1 AND j=1) -> (4*d1*d2)/mu*(mu*
    d+d1*d2)
ENDCOND #) IN
  inverse(q+mu*g) = h

sect_bound_P2Q: THEOREM FORALL(d,d1,d2,mu:posreal, lam:nnreal,v:Vector
    [2]):
  LET g = (# cols :=2, rows:=2,
  matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> 1/(d1*d2), (i=0 AND j=1) -> -(d1+d2)/(2*d1*d2),
(i=1 AND j=0) -> -(d1+d2)/(2*d1*d2), (i=1 AND j=1) -> 1
ENDCOND #) IN
  LET q =(# cols :=2, rows:=2,
  matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> 1/d, (i=0 AND j=1) -> 0,
(i=1 AND j=0) -> 0, (i=1 AND j=1) -> 0
ENDCOND #) IN
```

```
  LET D = 4*(d1*d2)^2-mu*d*(d1-d2)^2 IN
  D>0 AND
  in_ellipsoid_P2?(2,q+mu*g,v,lam) IMPLIES LET h = 1/D *(# cols :=2,
     rows:=2,
  matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> sq(2*d1*d2)*d,       (i=0 AND j=1) -> (2*d*d1*d2)*(d1+
   d2),
(i=1 AND j=0) -> (2*d*d1*d2)*(d1+d2), (i=1 AND j=1) -> (4*d1*d2)/mu*(mu*
   d+d1*d2)
ENDCOND #) IN
  in_ellipsoid_Q2?(2,h,v,lam)


sect_bound_ell_Q: THEOREM FORALL (n:posnat, Q:SquareMat(n),
      c,x:Vector[n], d1,d2:posreal,mu:posreal, f:[real->real]):
  LET d = c*(Q*c) IN
  LET D = 4*(d1*d2)^2-mu*d*(d1-d2)^2 IN
  LET v = Block2V(V2Block(1,1)(LAMBDA (i:below(1)): c*x,
             LAMBDA (i:below(1)): f(c*x))) IN
  in_ellipsoid_Q?(n,Q,x) AND
  D>0 AND
  (f(c*x)-(1/d1)*(c*x))*(f(c*x)-(1/d2)*(c*x))<=0 AND
  d>0
  IMPLIES
  LET h = 1/D *(# cols :=2, rows:=2,
  matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> sq(2*d1*d2)*d,        (i=0 AND j=1) -> (2*d*d1*d2)*(d1+
   d2),
(i=1 AND j=0) -> (2*d*d1*d2)*(d1+d2), (i=1 AND j=1) -> (4*d1*d2)/mu*(mu*
   d+d1*d2)
ENDCOND #) IN
  in_ellipsoid_Q2?(2,h,v,(c*x)^2/d)

sat_sect_bound_ell_Q: THEOREM FORALL (n:posnat, Q:SquareMat(n),
          c,x:Vector[n],mu:posreal):
  LET v = Block2V(V2Block(1,1)(LAMBDA (i:below(1)): c*x,
             LAMBDA (i:below(1)): sat(c*x))) IN
  in_ellipsoid_Q?(n,Q,x)
  IMPLIES
  LET d1 = sqrt(c*(Q*c)) IN
  LET D = 4-mu*(d1-1)^2 IN
  D>0 AND
  d1>0 IMPLIES
  LET h = 1/D *(# cols :=2, rows:=2,
matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> sq(2*d1),        (i=0 AND j=1) -> (2*d1)*(d1+1),
(i=1 AND j=0) -> (2*d1)*(d1+1),  (i=1 AND j=1) -> 4/mu*(mu*d1+1)
ENDCOND #) IN
  in_ellipsoid_Q2?(2,h,v,(c*x)^2/(c*(Q*c)))

ell_proj_lem_1: LEMMA FORALL (n:posnat, Q:SquareMat(n),
```

```
    c,x:Vector[n], d1,d2:posreal,mu:posreal, f:[real->real]):
  LET d = c*(Q*c) IN
  LET D = 4*(d1*d2)^2-mu*d*(d1-d2)^2 IN
  in_ellipsoid_Q?(n,Q,x) AND
  D>0 AND
  (f(c*x)-(1/d1)*(c*x))*(f(c*x)-(1/d2)*(c*x))<=0 AND
  d>0
  IMPLIES
  LET v = Block2V(V2Block(n,1)( ((c*x)/d)*(Q*c),LAMBDA (i:below(1)): f(c
      *x))) IN
  LET M = Block2M(M2Block(n,1,1,1)(transpose(V2Ml(n,(1/d)*(Q*c))),
           Zero_mat(1,1),Zero_mat(n,1),I(1))) IN
  LET h = 1/D *(# cols :=2, rows:=2,
  matrix:= LAMBDA(i,j:below(2)) :
COND
(i=0 AND j=0) -> sq(2*d1*d2)*d,        (i=0 AND j=1) -> (2*d*d1*d2)*(d1+
   d2),
(i=1 AND j=0) -> (2*d*d1*d2)*(d1+d2), (i=1 AND j=1) -> (4*d1*d2)/mu*(mu*
   d+d1*d2)
ENDCOND #) IN
  in_ellipsoid_Q2?(n+1,M*h*(transpose(M)),v,(c*x)^2/d)

ell_proj_lem_2: LEMMA FORALL (n:posnat, Q:SquareMat(n), c,x:Vector[n]):
  c*(Q*c)>0 AND
  in_ellipsoid_Q?(n,Q,x) IMPLIES
  in_ellipsoid_Q2?(n,Q,x-((c*x)/(c*(Q*c)))*(Q*c),1-(c*x)^2/(c*(Q*c)))

ell_proj_combination: THEOREM FORALL (n:posnat, Q:SquareMat(n),
          c,x:Vector[n], h:Mat(2,2), y:real):
  LET v = Block2V(V2Block(1,1)(LAMBDA (i:below(1)): c*x,
             LAMBDA (i:below(1)):y)) IN
  LET outV = Block2V(V2Block(n,1)(x,LAMBDA (i:below(1)): y)) IN
  in_ellipsoid_Q?(n,Q,x) AND
  in_ellipsoid_Q?(2,h,v) AND
  c*(Q*c) >0 IMPLIES
  in_ellipsoid_Q?(n+1,Block2M(M2Block(n,1,n,1)(
      Q-1/(c*(Q*c))*(transpose(V2Ml(n,Q*c))*V2Ml(n,Q*c)),
      Zero_mat(1,n),Zero_mat(n,1),Zero_mat(1,1))) +
          Block2M(M2Block(n,1,1,1)(
      1/(c*(Q*c))*transpose(V2Ml(n,Q*c)),
      Zero_mat(1,1),Zero_mat(n,1),I(1)))*
      h *
      transpose(Block2M(M2Block(n,1,1,1)(
            1/(c*(Q*c))*transpose(V2Ml(n,Q*c)),
            Zero_mat(1,1),Zero_mat(n,1),I(1)))),
      outV)


convex_ellipsoid: THEOREM FORALL (n:posnat, Q:SquareMat(n),
      x,y:Vector[n], lam:nnreal):
in_ellipsoid_Q?(n,Q,x) AND
in_ellipsoid_Q?(n,Q,y) AND
lam<=1 IMPLIES
in_ellipsoid_Q?(n,Q,lam*x+(1-lam)*y)
```

```
end ellipsoid
```

REFERENCES

[1] Leveson, N. and Turner, C., "An investigation of the therac-25 accidents," *Computer*, vol. 26, pp. 18–41, July 1993.

[2] "Medical Device Recall Report, FY2003 to FY2012," tech. rep., Food and Drug Administration, 2012.

[3] Hatcliff, J., Wassyng, A., Kelly, T., Comar, C., and Jones, P., "Certifiably safe software-dependent systems: Challenges and directions," in *Proceedings of the on Future of Software Engineering*, FOSE 2014, (New York, NY, USA), pp. 182–200, ACM, 2014.

[4] Hrgarek, N., "Certification and regulatory challenges in medical device software development," in *Software Engineering in Health Care (SEHC), 2012 4th International Workshop on*, pp. 40–43, June 2012.

[5] Burghardt, J., Gerlach, J., and Hartig, K., "ACSL by example: Towards a verified C standard library, version 4.2.0 for Frama-C Beryllium 2," 2010.

[6] Izerrouken, N., Thirioux, X., Pantel, M., and Strecker, M., "Certifying an automated code generator using formal tools : Preliminary experiments in the GeneAuto project," in *ERTS*, 2008.

[7] Moy, Y., "Union and cast in deductive verification."

[8] Council, N. R., Jackson, D., and Thomas, M., *Software for Dependable Systems: Sufficient Evidence?* Washington, DC, USA: National Academy Press, 2007.

[9] Boulton, R. J., Hardy, R., and Martin, U., "A Hoare logic for single-input single-output continuous-time control systems," in *Proceedings 6th International Workshop on Hybrid Systems, Computation and Control*, pp. 113–125, Springer, 2003.

[10] Clarke, Jr., E. M., Grumberg, O., and Peled, D. A., *Model Checking*. Cambridge, MA, USA: MIT Press, 1999.

[11] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S., "Chaff: Engineering an efficient SAT solver," in *Proceedings of the 38th Annual Design Automation Conference*, DAC '01, (New York, NY, USA), pp. 530–535, ACM, 2001.

[12] De Moura, L. and 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*, TACAS'08/ETAPS'08, (Berlin, Heidelberg), pp. 337–340, Springer-Verlag, 2008.

[13] Gao, S., Kong, S., and Clarke, E., "dReal: An SMT solver for nonlinear theories over the reals," in *Automated Deduction CADE-24* (Bonacina, M., ed.), vol. 7898 of *Lecture Notes in Computer Science*, pp. 208–214, Springer Berlin Heidelberg, 2013.

[14] Souyris, J., "Industrial experience of abstract interpretation-based static analyzers," in *Building the Information Society* (Jacquart, R., ed.), vol. 156 of *IFIP International Federation for Information Processing*, pp. 393–400, Springer US, 2004.

[15] Feret, J., "Static analysis of digital filters," in *ESOP*, no. 2986, 2004.

[16] Feret, J., "Numerical abstract domains for digital filters," in *International workshop on Numerical and Symbolic Abstract Domains (NSAD)*, 2005.

[17] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "The ASTRÉE analyzer," in *Proceedings of the 14th European Symposium on Programming*, vol. 3444 of *Lecture Notes in Computer Science*, 2005.

[18] Monniaux, D., "Compositional analysis of floating-point linear numerical filters," in *CAV*, 2005.

[19] Roozbehani, M., Féron, E., and Megretski, A., "Modeling, optimization and computation for software verification," in *HSCC*, 2005.

[20] Cousot, P., "Proving program invariance and termination by parametric abstraction, Lagrangian relaxation and semidefinite programming," in *VMCAI*, 2005.

[21] Adjé, A., Gaubert, S., and Goubault, E., "Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis," in *ESOP*, pp. 23–42, 2010.

[22] Gawlitza, T. M. and Seidl, H., "Computing relaxed abstract semantics w.r.t. quadratic zones precisely," in *SAS*, 2010.

[23] Alegre, F., Feron, e., and Pande, S., "Using ellipsoidal domains to analyze control systems software," *CoRR*, vol. abs/0909.1977, 2009.

[24] Feron, E., "From control systems to control software," *Control Systems, IEEE*, vol. 30, pp. 50 –71, dec. 2010.

[25] Gonthier, G., "Point-free, set-free concrete linear algebra," in *ITP*, pp. 103–118, 2011.

[26] Garillot, F., Gonthier, G., Mahboubi, A., and Rideau, L., "Packaging Mathematical Structures," in *Theorem Proving in Higher Order Logics*, vol. 5674 of *LNCS*, Springer, 2009.

[27] Harrison, J., "The HOL light formalization of Euclidean space," in *AMS Special Session on Formal Mathematics for Mathematicians*, 2011.

[28] de Dinechin, F., Quirin Lauter, C., and Melquiond, G., "Certifying the floating-point implementation of an elementary function using Gappa," *IEEE Trans. Computers*, vol. 60, no. 2, pp. 242–253, 2011.

[29] Muñoz, c. and Narkawicz, A., "Formalization of an efficient representation of Bernstein polynomials and applications to global optimization," *J. of Automated Reasoning*, 2011.

[30] Alur, R., Henzinger, T., and Ho, P.-H., "Automatic symbolic verification of embedded systems," *Software Engineering, IEEE Transactions on*, vol. 22, pp. 181–201, Mar 1996.

[31] Frehse, G., Le Guernic, C., Donzé, A., Cotton, S., Ray, R., Lebeltel, O., Ripado, R., Girard, A., Dang, T., and Maler, O., "SpaceEx: Scalable verification of hybrid systems," in *Computer Aided Verification* (Gopalakrishnan, G. and Qadeer, S., eds.), vol. 6806 of *Lecture Notes in Computer Science*, pp. 379–395, Springer Berlin Heidelberg, 2011.

[32] Kloetzer, M. and Belta, C., "A fully automated framework for control of linear systems from temporal logic specifications," *Automatic Control, IEEE Transactions on*, vol. 53, pp. 287–297, Feb 2008.

[33] Platzer, A., *Logical Analysis of Hybrid Systems: Proving Theorems for Complex Dynamics*. Heidelberg: Springer, 2010.

[34] Boyd, S., El Ghaoui, L., Éric Féron, and Balakrishnan, V., *Linear Matrix Inequalities in System and Control Theory*, vol. 15 of *SIAM*. June 1994.

[35] Boyd, S. and Vandenberghe, L., *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[36] Lyapunov, A. M., *The General Problem of the Stability of Motion*. PhD thesis, Moscow University, 1892.

[37] Haddad, W. and Chellaboina, V., *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2011.

[38] Sturm, J. F., "Using sedumi 1.02, a matlab toolbox for optimization over symmetric cones," *Optimization Methods and Software*, vol. 11, no. 1-4, pp. 625–653, 1999.

[39] Pólik, I. and Terlaky, T., "A survey of the s-lemma," *SIAM Rev.*, vol. 49, pp. 371–418, July 2007.

[40] Luré, A. and Postnikov, V., "On the Theory of Stability of Control Systems," *Prikl. Mat. Mehk.*, vol. 8, 1944.

[41] Roux, P., Delmas, R., and Garoche, P.-L., "Smt-ai: An abstract interpreter as oracle for k-induction," *Electron. Notes Theor. Comput. Sci.*, vol. 267, pp. 55–68, Oct. 2010.

[42] Cousot, P. and Cousot, R., "Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints," in *Proceedings of the 4th ACM SIGACT-SIGPLAN Symposium on Principles of Programming Languages*, POPL '77, (New York, NY, USA), pp. 238–252, ACM, 1977.

[43] Roux, P. and Garoche, P.-L., "Practical policy iterations," *Formal Methods in System Design*, vol. 46, no. 2, pp. 163–196, 2015.

[44] Hoare, C. A. R., "An axiomatic basis for computer programming," *Comm. ACM*, vol. 12, pp. 576–580, October 1969.

[45] Floyd, R. W., "Assigning meanings to programs," *Mathematical aspects of computer science*, vol. 19, no. 19-32, p. 1, 1967.

[46] Dijkstra, E., *A Discipline of Programming*. Prentice-Hall, 1976.

[47] Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., and Prevosto, V., *ACSL: ANSI/ISO C Specification Language*, 2008. `http://frama-c.cea.fr/acsl.html`.

[48] Cuoq, P., Kirchner, F., Kosmatov, N., Prevosto, V., Signoles, J., and Yakobowski, B., "Frama-C: A software analysis perspective," in *Proceedings of the 10th International Conference on Software Engineering and Formal Methods*, SEFM'12, (Berlin, Heidelberg), pp. 233–247, Springer-Verlag, 2012.

[49] Leavens, G. T. and Cheon, Y., "Design by contract with JML," 2006.

[50] Peano, G., *Arithmetices Principia Novo Methodo Exposita*. Bocca, 1889.

[51] Boldo, S. and Filliatre, J.-C., "Formal verification of floating-point programs," in *Computer Arithmetic, 2007. ARITH '07. 18th IEEE Symposium on*, pp. 187–194, June 2007.

[52] Owre, S., Shankar, N., Rushby, J. M., and J.Stringer-Calvert, D. W., *PVS Language Reference*. Computer Science Laboratory, SRI International, Sept. 1999.

[53] Owre, S., Rushby, J. M., , and Shankar, N., "PVS: A prototype verification system," in *CADE*, vol. 607 of *LNAI*, pp. 748–752, Springer, Jun 1992.

[54] Barendregt, H., *The Lambda Calculus: Its Syntax and Semantics*. Studies in logic and the foundations of mathematics, North-Holland, 1984.

[55] Rugina, A.-E., Thomas, D., Olive, X., and Veran, G., "Gene-Auto: Automatic software generation for real-time embedded systems." 2010.

[56] Wang, T., *Formal verification of control software in the autocoding framework*. PhD thesis, Georgia Institute of Technology, 2015.

[57] Owre, S. and Shankar, N., "Theory Interpretation in PVS," tech. rep., Computer Science Laboratory, SRI International, 2001.

[58] Muñoz, C., "Batch proving and proof scripting in PVS," *NIA-NASA Langley, National Institute of Aerospace, Hampton, VA, Report NIA Report*, no. 2007-03, 2007.

[59] Roux, P., Jobredeaux, R., Garoche, P.-L., and Feron, E., "A generic ellipsoid abstract domain for linear time invariant systems," in *HSCC*, pp. 105–114, 2012.

[60] Delmas, D., Goubault, E., Putot, S., Souyris, J., Tekkal, K., and Védrine, F., "Towards an industrial use of fluctuat on safety-critical avionics software," in *Formal Methods for Industrial Critical Systems* (Alpuente, M., Cook, B., and Joubert, C., eds.), vol. 5825 of *Lecture Notes in Computer Science*, pp. 53–69, Springer Berlin Heidelberg, 2009.

[61] Roux, P., Jobredeaux, R., and Garoche, P.-L., "Closed loop analysis of control command software," in *HSCC*, 2015.

[62] Feron, E., Jobredeaux, R., and Wang, T., "Autocoding control software with proofs i: Annotation translation," in *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pp. 1 –19, oct. 2011.

[63] Wang, T., Ashari, A., Jobredeaux, R., and Feron, E., "Credible autocoding of fault detection observers," in *American Control Conference (ACC), 2014*, pp. 672–677, June 2014.

[64] Herencia-Zapana, H., Jobredeaux, R., Owre, S., Garoche, P.-L., Feron, E., Perez, G., and Ascariz, P., "Pvs linear algebra libraries for verification of control software algorithms in c/acsl," in *NASA Formal Methods*, pp. 147–161, 2012.

[65] Feron, E. (in press), *Advances in Control System Technology for Aerospace Applications*. Springer, 2015.

VITA

Romain Jobredeaux was born in Amiens, France. His research interests center around the application of domain specific knowledge in the formal certification of safety-critical systems. He has previously worked with NASA (Langley and Ames centers), and ONERA (Toulouse, France), as a visiting researcher. He holds a master's degree in Information Security from the Georgia Institute of Technology, as well as a master's degree in Electrical and Computer Engineering from Supélec, a leading engineering school in France.