CREDIBLE AUTOCODING OF CONTROL SOFTWARE

A Thesis
Presented to
The Academic Faculty

by

Timothy E. Wang

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

Georgia Institute of Technology
August 2015

CREDIBLE AUTOCODING OF CONTROL SOFTWARE

Approved by:

Dr. Eric Feron, Advisor
Daniel Guggenheim School of
Aerospace Engineering
Georgia Institute of Technology

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

Dr. Panagiotis Tsiotras
Daniel Guggenheim School of
Aerospace Engineering
Georgia Institute of Technology

Dr. Pierre-Loïc Garoche
DTIM
ONERA, Toulouse, France

Dr. Marc Pantel
IRIT/ACADIE
École Nationale Supérieure d'Électronique, d'Électrotechnique, d'Informatique, d'Hydraulique, et de Télécommunicatio, Toulouse, France

Date Approved: July 24th, 2015

To grandpa and nanna. I am saddened that you guys could not witness any of my graduations.

PREFACE

'A good scientist is a person with original ideas. A good engineer is a person who makes a design that works with as few original ideas as possible."

Freeman Dyson

Institute for Advanced Study

"...Future planes will have one pilot and one dog in the cockpit. The pilot's job will be to feed the dog. The dog's job will be to make sure the pilot stays away from the instruments. "

MBA Professor

University of Southern California

ACKNOWLEDGEMENTS

I feel grateful to many people, who have contributed in some ways to the successful completion of this thesis. It has been a long, winding road with many U-turns but it was also very interesting at times and I have learned a lot from this experience and I don't regret anything.

Foremost, I am indebted to Professor Eric Feron, who has been one of the greatest influences in my life. Eric always had an unique and out of box streak in his thoughts, which kept any discussions with him, no matter what subject, entertaining. He has taught me much about what is a good research and what we should look for in a good research. He might be very tough but he is also fair. He might be very angry but he is also forgiving. I have personally enjoyed his stream of unfiltered politically incorrect humor even though it might be out of bound for some people. Without his patience and continuous support over the years, I would have been lost like a rudderless boat on a stormy sea. All the priceless experiences that I had, such as my extended stays in France, would not have been possible without Eric.

I like to thank Professor Marc Pantel for his guidance over the years. It was a pleasure to work on his open source tool-chain Gene-Auto. Marc's encyclopedic knowledge on the practice of formal methods and model-based engineering in the industry has always come in handy for exploring research directions with industrial impact. I am forever enriched by all the heated conversations that he and I had in Batiment F on topics ranging from American politics to Airbus vs. Boeing. Of course I am very grateful to him for inviting me to Toulouse on two separate occasions. It was the ideal environment to complete this thesis.

Alwyn, thank you for introducing me to the world of formal methods and PVS.

While my interest in formal methods was piqued at a young age, however I had forgotten about it for many years. The summer I spent at National Institute of Aerospace was certainly an eye opening one. You provided the inspiration for me to march on as a stranger in a strange land. Your soliloquy about history was very much appreciated even if I could not absorb it all.

I am grateful to Dr. Pierre-Loic Garoche, who is one of the finest computer scientists that I have met and, by some stroke of luck, is also very much interested in control theory. He always offered clear, detailed and yet concise suggestions about writings, solutions to problems, and publishing papers. I feel privileged to have him on my committee.

I like to thank my committee members Dr. Holzinger and Dr. Tsiotras for being open minded. I realize the subject of my thesis is very much disconnected from the traditional areas of research in this department.

I am grateful to Arnaud D., who is the first person in Toulouse that I got to know well. He welcomed me to his home on more than one occasion and he was always patient in helping me with my troubles with the French language. His generosity and hospitality made my stay in Toulouse a much more pleasant one.

To Yohan, Romain, Vivek, Aude, Manu, Florian, and the rest of the "balibump" posse, you guys already know how I feel about you even though I have problems expressing my love, my thanks and my gratitudes. Thank you for the unforgettable years. Hopefully, our paths will cross again in the future.

A special shutout to Vlad for indulging in my afternoon whims with meandering conversations over many hypothetical subjects.

Of course, I should not forget the sponsors. Without their continuous support, I would have been hungry and homeless. Cheers to the Air Force Research Laboratory (AFRL) for their grant CertaAMoR, to National Aeronautics and Space Administration (NASA) for their grant NNX08AE37A, to the Army Research Office for the

grant MURI, and to the National Science Foundation (NSF) for the grants CrAVES and SORTIES.

Now to all of my friends who are busy in real life, I am about to join you! I am grateful for all of your unwavering friendships that have lasted through time and space. We had some bloody good ol' times imbibing and merry-making all over the world from the beautiful coastal hideaways of Capetown to beneath the northern lights in Jonkoping.

Finally, I like to thank all the nice people at ENSEEIHT. Sylvie E., your smile always brightened my day and your impeccable secretarial abilities made my life in Toulouse easy. I hope you are enjoying your retirement. Godspeed. Charlie and Simone, I enjoyed all of our lunches together. Charlie, I especially thank you for introducing me to the wonderful world of Raclette.

Contents

List of Tables

List of Figures

SUMMARY

Formal methods is a discipline of using a collection of mathematical techniques and formalisms to model and analyze software systems. Motivated by the new formal methods-based certification recommendations for safety-critical embedded software and the significant increase in the cost of verification and validation (V&V), this research is about creating a software development process for control systems that can provide mathematical guarantees of high-level functional properties on the code. The process, dubbed credible autocoding, leverages control theory in the automatic generation of control software documented with proofs of their stability and performance. The main output of this research is an automated, credible autocoding prototype that transforms the Simulink model of the controller into C code documented with a code-level proof of the stability of the controller. The code-level proof, expressed using a formal specification language, are embedded into the code as annotations. The annotations guarantee that the auto-generated code conforms to the input model to the extent that key properties are satisfied. They also provide sufficient information to enable an independent, automatic, formal verification of the auto-generated controller software.

Chapter I

INTRODUCTION

1.1 Verification of Embedded Control Software

A wide variety of real-time embedded reactive systems, especially their most critical parts, relies on a decision and control computational core. The decision and control functions of an aircraft, a satellite, a ground vehicle, a turbine engine or a medical device are typically processed by a computational loop that is repeated during the active period of the controlled device. This computational loop also models the acquisition of new input values via sensors, from environment measures (wind speed, acceleration, engine RPM, ...) and actuations, for example, the brakes, the accelerator, the stick or wheel control.

For safety-critical applications i.e. the real-time control system of a civilian aircraft, due to the significant costs of failure [67, 18], the avionics industry has had to devote significant time and money towards convincing the regulatory authorities that their on-board products are safe and sound. Part of this rigorous certification process is Verification & Validiation (V&V). Verification is about determining if the output software satisfies the input specifications i.e. is the produced software correct? In contrast, validation is about determining if the specifications are complete and correct i.e. satisfies the end customer's needs.

The specifications may include a description of the software and/or the properties

that the correctness of the software is dependent on. The description of the software can vary in its level of abstraction or its level of details. For example, a description can be some or all of below:

1. A mathematical equation. For example, one can specify the ordinary differential equation that governs the behavior of the controller.

2. An informal natural language description of the tasks that the software need to accomplish.

3. Pseudo-code implementation of the software. This description of the software is very "close" to the actual software.

4. A model of the software in a synchronous language such as lustre, Simulink or State-flow.

The properties or *invariants*, which also can be provided as part of the specifications, are sets of states that holds for all possible executions of the software. The invariants can range from a low level of abstraction, i.e. division by zero or buffer overflow, to the satisfaction of high-level functional properties, such as the controller should achieve a closed-loop bandwidth of 25hz.

The distinction between validation and verification may be blurred, for example, in cases when high-level functional properties such as the performance of the system is included in the specifications. For example, in the context of control software, the control engineer can specify the exact differential equations of the controller along with the expected robustness margins. In that scenario, formally verifying the software

also validates the software against its robustness performance measures up to some inaccuracies in the plant model. In this thesis, however, any activities regarding proving the correctness of software is referred to as formal verification.

1.1.1 Old Certification Guidelines and the Cost Explosion

Currently, certification of safety-critical embedded control software employs tests or simulation based methods. For example, the old FAA certification guideline, DO-178B [78] recommends a process for certifying real-time embedded software but not any specific goals or methods. This process boils down to testing or simulating the software system for as many possible inputs as one can within a period of time. As John Rushby once said: "Because we cannot demonstrate how well we've done, we'll show how hard we've tried [16]."

The extensive simulations have reduced the frequency of on-board software failures in the commercial air transport sector to almost zero [26], but at a great time and cost disadvantage. Already, in the case of safety-critical computer controlled systems such as those found on a modern commercial aircraft, the cost of developing the on-board software approaches one half of the total project development budget [66]. Furthermore, in the software development budget itself, nearly one half is spent for certification. The geometric explosion in the size and complexity of modern avionics software has arguably made this process increasingly untenable. For example, if we just consider code coverage analysis [60], which has a runtime that grows linearly with the complexity of the code; if the current piece of code is one hundred times more complex than its predecessor, what took a month of tests before, now will take

ten years.

1.1.2 Next Generation Certification Guidelines

In any case, extensive simulations, unless exhaustive, do not guarantee that the software is sound for all possible inputs. Given the cost and time constraints, exhaustive simulations is rarely if ever possible. The new FAA certification guideline, the DO-178C [79], has three technology-specific supplements [80]. Two of the technological supplements are relevant to the research in this thesis. The first one describes formal methods [80]. Formal methods, from the field of computer science, is a collection of formalisms and mathematical techniques for modeling and analyzing software. Instead of testing a program for bugs, the practitioners of formal methods seek to prove the absence of bugs in programs.

The benefits of using formal methods in the certification process derives from the potential replacement of tedious simulations with an automatic tool that proves the correctness of the code [61]. It has been suggested, but not without controversy, that the usage of formal methods can lead to a reduction in the cost of the safety-critical software development process [102]. The reasoning for that argument is as follows: any increase in cost due to integrating formal methods into the software development process is dwarfed by the savings due to the reduction of tests in the software certification process.

The second technology is model-based development (MBD). MBD is a software development process, where the software is first written in a high-level modeling language such as Simulink [82] or SCADE [2]. The software is then tested (simulated)

and debugged at this level of abstraction. The source code is eventually generated automatically from the high-level description using a program called the autocoder. One benefit of this approach, according to [84], is that it allows more bugs to be eliminated early in the software development process e.g. at the design level. The bugs that are found at the design level are a couple of orders of magnitude less costly to fix compared to bugs that are discovered after code has gone through the last stages of certification process [70]. The other potential benefit of MBD is that it enables a more rapid and efficient prototyping of software because generally speaking, the higher the level of abstraction, the easier and less mistake-prone it is for the domain experts to write the software [85]. For example, at the level of differential equations, a linear controller can be specified, by the control engineers, using only a few matrices. The same linear controller expressed in Simulink [25] could result in a cluttered collection of signals and blocks.

1.1.3 Contributions and Literature Review

This dissertation is motivated by the rising cost of V&V and seeks to leverage concepts from formal methods, model-based development and control theory to make improvements upon the current state of embedded software development process.

In formal methods, the behaviors of processes are modeled with mathematical objects, such as state machines, transition systems, Petri nets, hybrid automaton, process algebra, etc. With a few exceptions, these formal structures are well-suited for analyzing systems with discrete behaviors and finite sets of states. Their power of

description covers many types of processes including control systems, hardware circuits, mechanical machines, arbitrary computer programs, etc. Model checking [20], which is a major technique in formal methods, automatically checks a finite-state model of a program for their *safety* and *liveness* properties. A safety property is a set of undesirable states, that the software system must not reach. An example of safety property is the lack of division by zero. A liveness property is a property expressed over the traces of the program i.e. sequences of instructions which are executed by the program. An example of liveness property is the program will eventually terminate. Abstraction interpretation [23], another major technique from formal methods, has been applied to the flight control code of Airbus A380 [24]. In abstract interpretation, properties are computed directly from the code to prove the absence of low-level runtime errors such as buffer overflow or division by zero.

Control theory is a field developed to analyze dynamical systems with inputs. Complex computational cores in domain specific software such as control software make their automatic analysis using traditional formal methods difficult in the absence of inputs from the domain experts i.e. the control engineers. The notions of closed-loop and open-loop stability while trivial to control theorists but yet, as properties, are never expressed and proved on the code implementations of controllers. In fact, any knowledge about the control properties of the system tend to be lost once the development process has moved beyond the model level. At the level of the code, control properties are difficult for analysis tools, based on either abstract interpretation or model-checking, to recover. This difficulty is due to a state-space explosion problem since difference equations are state machines with an infinite state-space and

because control systems typically yield quadratic invariants which makes it hard for abstract interpretation tools to analyze.

In general, proof-checking a code documented with its proof is simpler than deducing the proof automatically from the undocumented code. For control algorithms, the engineers, who designed the controllers, are capable of producing proofs that can greatly facilitate this analysis. However there are differences between the languages of control theory and the ones of formal methods, which prevents meaningful communications between the two disciplines. From the languages used to express the *semantic* or the mathematical meaning of the system to the languages used in the proof of correctness of the system, this semantic gap makes it difficult for either side to use techniques from the other side. The main motivation for this thesis is the closing of this semantic gap.

This thesis is about the translation of domain-specific knowledge from control theory into a language suitable for program verification. We show how basic concepts from the fields of formal method and control theory are intertwined. We argue that information from control theory can be translated down to the level of code and applied towards the verification of control programs. A proof of concept is built which demonstrates this approach. To summarize, the main contributions of this thesis are as follows.

1. Developing a novel translational framework that enables an efficient flow of information from control theory to be applied towards formal verification of control software.

2. Creating a new annotation language to express control properties and proofs inside of an autocoding environment.

3. Building the first prototype autocoder that is capable of documenting control properties and proofs on the level of the code.

4. Demonstrating the prototype on an example from the industry.

5. Accounting for implementation artifacts such as the effect of floating-point computations on the stability proofs based on theory of real numbers.

The structure of this thesis and the publications produced are as follows. Chapter 2 describe the link established between Lyapunov-based methods and formal analysis of software. Chapter 3 gives the development of the translational framework [47, 99]. Chapter 4 presents the realization of the translational framework i.e. the prototype translator [96]. Chapter 5 describes methods to make the prototype translator sound with regards to floating-point computation errors. Chapter 6 describes the application of the prototype tool-chain to an example from industry [97]. Chapter 7 concludes the thesis and discusses some future topics of explorations including the work which extended credible autocoding to convex optimization algorithms [98].

1.1.4 Literature Review

This section provides an overview of past and concurrent research works that can be divided into the following categories.

1. Background to the current research.

2. Techniques used in the current research.

3. Works that inspired the current research.

4. Works that are directly related to the current research.

The intersection of two separate fields, control theory and formal methods, is not empty. For example, a link between formal methods and control theory was first established in the work of Jerome Feret [33], who analyzed a second order digital filter using ellipsoidal templates within the abstraction interpretation framework. Ursula Martin in [5] attempted at expressing control-theoretic concepts formally within a Simulink environment. However that work did not result in producing proofs of high-level control properties on the code. Feron in [34], which directly led to this thesis, was the first to demonstrate that information provided by a control analysis can potentially be documented on a controller code to support its verification.

The first person to conceptualize the process of proving programs was Alan Turing in [93]. Turing's work was remarkable as it provided a proof of a program using a method that resembled the much later flowchart system of Robert Floyd [35]. Fast forward to the 1960s, McCarthy was generally attributed as the first person to write about mathematically proving programs [57], and Naur in [65] were among the first along with Floyd to describe a working method for doing so. From that point on, more formal systems of program verification were done by Charles Hoare with his Hoare logic in [42] and Edsger Dijstra in [31]. In the credible autocoding framework, classic concepts from program verification such as Hoare logic is used for the annotations of the code.

The model checking technique [19, 72] came in the early 1980s as the result of simultaneous works of Clarke and Sifakis. Since then, the model checking approach has been successfully applied to hardware systems [21]. For software systems, model checking algorithms suffered with the issue of scalability as the state-space is significantly larger in computer programs. Recent advances in model checking techniques such as efficient Boolean Satisfiability (SAT) solvers [64], and eventually Satisfiability Modulo Theory (SMT) solvers [27] has improved greatly the scalability and scope of model checking. Other advances in model checking have been in incorporating abstractions into the construction of the program model [4, 90]. However, control systems remain difficult for model checking techniques to analyze.

Abstract interpretation was first introduced by the Cousots in [23]. This technique has since been used in practice to check for low-level errors such as buffer overflow or divide by zero, of commercial aircraft software [24]. Practical advances in abstract interpretation have been either in more efficient abstract domains [94]. or in more efficient widening and narrowing algorithms [6]. Other recent research activities in abstract interpretation include new relational abstract domains such as the zonotope in [38] or new extensions for special properties such as floating-point errors [37].

A related work in [75], presented a framework to use a Lyapunov-based method to check a control software for runtime errors. Unlike that work, which searches for Lyapunov functions to verify low-level properties of the software, the current research is focused on translating known high-level properties of the system down to the level of the code.

The following concurrent works are related to the current research. The first one, by Garoche et al. in [77], uses ellipsoidal templates, computed from stability analysis, for the static analysis of linear control programs. Another work, by Herencia-Zapana et al. in [40] incorporated the mathematical theories, that are useful in the verification of the annotated code produced by the credible autocoding framework, into a NASA theorem prover.

The idea of closing the semantic gap is not new. For example, the prototype described in [22] is a translational framework from Simulink to the various model checking languages. There are many other works in the formal methods literature on translational prototypes for transforming Simulink into another language that is more suitable for verification. For example, the work in [17] presents a tool that translates from a discrete-time subset of Simulink to Lustre or the work in [3] that translates Simulink into hybrid automata. The key difference between that body of work and this thesis is that we also provide formal guarantees of the functional properties of the system. Other more direct approaches in the literature regarding the verification of Simulink models, include creating formal semantics for the Simulink/Stateflow models [89, 13].

On the matter of floating-point computation errors in control systems, the recent thesis of Maisonneuve [55] produced similar results. Another recent work by Roux [76] computes the over-approximation of ellipsoidal sets due to floating-point errors with an application towards soundly checking if a matrix is positive-definite. Unlike that work, this thesis is about accounting for the floating-point errors in the context of the translation framework.

Chapter II

AXIOMATIC SEMANTICS FOR CONTROL SYSTEMS

This chapter reviews some fundamental concepts from control theory and formal methods. We first give an introduction to Lyapunov-based methods. The techniques to compute quadratic invariants for linear and nonlinear systems are described. This is followed by an introduction to concepts from program verification including axiomatic semantics, Hoare logic and Dijkstra's predicate transformer semantics. We show how domain-specific knowledge from control theory can be applied towards the deductive verification of a control program.

2.1 Lyapunov-based Methods

One of the main difficulties in software verification is being able to compute a *fix-point* of a function. Consider a program `while(true)f(x);end`, where $f(x)$ is an abstraction of the loop body. A fix-point of the function $f(x)$ is a set of program states X such that $f(X) = X$. For the example `while` program, computing the fix-point X of f leads to a loop invariant of the program. As per Rice's Theorem [73], no general algorithm exists that can decide the fix-point of arbitrary functions $f(x)$. An analogous concept from control theory is the Lyapunov function.

Lyapunov formalized the notion of stability for continuous dynamical systems in 1892 [54]. Here we give the definition of the discrete-time version [39]. Let $\mathcal{G}$ be the

discrete-time dynamical system

$$x(k+1) = f(x(k)), \quad x(0) = x_0, \quad k \in \mathbb{N} \tag{1}$$

where $x(k) \in \mathcal{D} \subseteq \mathbb{R}^n$ is the system state vector, $\mathcal{D} \subseteq \mathbb{R}^n$ and $0 \in \mathcal{D}$, $f(0) = 0$ and $f : \mathcal{D} \to \mathbb{R}^n$ is continuous on $\mathcal{D}$.

Definition 1.1 The zero solution to $\mathcal{G}$ is Lyapunov stable if, $\forall \epsilon > 0$, there exists $\delta(\epsilon) > 0$, such that if $\|x(0)\| < \delta$, then $\|x(k)\| \leq \epsilon$ for all $k > 0$.

Lyapunov's second method is the more commonly used technique for demonstrating the stability of dynamical systems.

Theorem 1.2 If there exists a function $V(x) : \mathcal{D} \to \mathbb{R}$ such that

1. $V(0) = 0$,
2. $V(x(k))$ is positive on $x \in \mathcal{D} / \{0\}$ and $V(0) = 0$,
3. $V(x(k+1)) - V(x(k)) \leq 0 \forall k \in N$,

then (1) is stable (locally).

The function $V(x)$ is a Lyapunov function candidate and if it satisfies the conditions in Theorem 1.2 then it is a Lyapunov function. Let f in (1) be the linear function $f(x) = Ax$, $A \in \mathbb{R}^{n \times n}$, then (1) becomes a linear discrete-time system

$$x(k+1) = Ax(k), \quad x(0) = x_0, \quad k \in \mathbb{N}. \tag{2}$$

For a linear discrete-time system, a sufficient condition for Lyapunov stability is the existence of a quadratic Lyapunov function $V(x) = x^{\mathsf{T}} P x$ where P is positive-definite.

Theorem 1.3 If there exists a matrix $P \in \mathbb{R}^{n\times n}$ such that $P \succ 0$, and $A^\intercal PA - P \prec 0$, then the system in (2) is Lyapunov stable.

Proof. Assume that $V(x) = x^\intercal Px$. We have $V(0) = 0$, $V(x) > 0, x \neq 0$ by the definition of $P \succ 0$, and $A^\intercal PA - P \prec 0 \implies x^\intercal(k)A^\intercal PAx(k) - x^\intercal(k)Px(k) < 0 \implies V(x(k+1)) - V(x(k)) \leq 0$ for all $x \in \mathcal{D}$.

The inequality $A^\intercal PA - P \prec 0$, linear in the variable P, is a special case of linear matrix inequality or LMI.

2.1.1 Quadratic Invariants

The sub-level sets of a quadratic Lyapunov function $V(x) = x^\intercal Px$ form a family of ellipsoids $\mathcal{E}_{P,c} \triangleq \{x \in \mathbb{R}^n \mid x^\intercal Px \leq c\}$. In this thesis, a predicate notation is used to denote ellipsoidal sets. A predicate is a function $f(x)$ that takes a variable x, which belongs to some domain $\mathcal{S}$, and returns true or false. For example, the predicate $f(x) = x < 0$ returns false for any $x \in \mathbb{N}$. The predicate notation is useful for expressing ellipsoidal sets on variables in a program. For $P \in \mathbb{R}^{n\times n}$, and $c > 0$, we have a family of quadratic predicates

$$p(P, c)(x) = x^\intercal Px \leq c. \tag{3}$$

The notation $p(P, c)(x)$ is overloaded to also express the set $\{x \in \mathbb{R}^n \mid p(P, c)(x)\}$. If $P \succ 0$ then $p(P, 1)(x)$ is an ellipsoidal set.

Another way of expressing ellipsoidal sets is to use the Schur-form. For a positive-semidefinite matrix $Q \in \mathbb{R}^{n\times n}$, and a scalar $c > 0$, an ellipsoid in the Schur-form $\mathcal{G}_{Q,c}$

is the set

$$\left\{ x \in \mathbb{R}^n \begin{bmatrix} & x^{\mathsf{T}} \\ x & Q \end{bmatrix} \succ 0 \right\}. \tag{4}$$

As in (3), we also use a predicate notation to express ellipsoids in the Schur-form. We have for $Q \succeq 0$, $x \in \mathbb{R}^n$, and $c > 0$, the family of quadratic predicates

$$q(Q, c)(x) = \begin{bmatrix} c & x^{\mathsf{T}} \\ x & Q \end{bmatrix} \succ 0. \tag{5}$$

Remark 1 If the matrix parameter Q in $q(Q, c)(x)$ is singular, then the ellipsoidal set $q(Q, c)(x)$ is degenerate. An example of a degenerate ellipsoid in $\mathbb{R}^3$ is an ellipse. If P in $p(P, c)(x)$ is singular, then the set $p(P, c)(x)$ becomes an elliptic cylinder. If the matrix parameter Q in $q(Q, c)(x)$ is not singular, and $Q = P^{-1}$, then the two ellipsoids $q(Q, c)(x)$ and $p(P, c)(x)$ are equivalent.

Computing a Lyapunov function $V(x) = x^{\mathsf{T}}Px$ for (2) leads to a quadratic invariant $p(P, c)(x)$. The quadratic invariant, including its sum of squares polynomial extensions [69], exists for a wide range of control systems of interest. Classical design methods such as the tuning of the proportional, integral and derivative (PID) gains, while not relying on the computation of quadratic Lyapunov functions, yields control systems that have quadratic invariants.

Theorem 1.4 For the linear system in (2), if there exists $P \succ 0$ such that $V(x) = x^{\mathsf{T}}Px$ is a Lyapunov function, then $p(P, c)(x)$ is an invariant set of (2).

Proof. If $V(x) = x^\intercal Px$ is a Lyapunov function, then $x^\intercal(k+1)Px(k+1) - x(k)Px(k) \leq 0$, $\forall x \in \mathcal{D}$. This implies that for $c = x_0^\intercal Px_0$, $x^\intercal(k)Px(k) \leq c$ for all $k \in \mathbb{N}$.

Remark 2 Without a loss of generality, we can assume $c = 1$, since we can always scale the matrix P by c^{-1}.

We can also compute quadratic invariants for nonlinear systems consist of linear systems in feedback interconnections with bounded nonlinearities. A unifying framework from robust control, the integral quadratic constraints (IQCs) [59], can be used to analyze the stability of many of such systems. The technique generates quadratic invariants as by-products.

2.1.2 Examples of Stability Analysis of Control Systems

We give two examples of stability analysis of control systems. Both analysis reduces the stability problem into a linear matrix inequality problem. The first problem is a linear system with bounded input. The second problem is the problem of absolute stability i.e. the stability of a linear system with a nonlinearity in the actuation, considered by Lur'e, Postnikov and others in the Soviet Union in the 1940s [53].

We first introduce the S-Procedure relaxation technique by Yakubovich [106]. Consider the quadratic forms $q(x), q_1(x), \ldots, q_m(x)$ defined on $x \in \mathbb{R}^n$. The S-Procedure is used to relax the problem of determining $q_1(x) \geq 0 \wedge \ldots \wedge q_m(x) \geq 0 \rightarrow q(x) \leq 0$ into a single matrix inequality.

Lemma 1.5 For the quadratic forms $q_(x), q_1(x), \ldots, q_m(x)$, if there exist scalar multipliers $\tau_i > 0$ such that for all $x \in \mathbb{R}^n$, $q(x) + \sum_i \tau_i q_i(x) \leq 0$, then $q_1(x) \geq$

$0 \wedge \ldots \wedge q_m(x) \geq 0 \rightarrow q(x) \leq 0.$

The S-Procedure for quadratic inequalities was generalized to sum of squares polynomials in [69].

Example 2.1.1 Consider a discrete-time linear system

$$\begin{aligned} x_+ &= Ax + By_{ref}, \quad x(0) = x_0 \\ y &= Cx, \end{aligned} \tag{6}$$

where $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{k\times n}$, $D \in \mathbb{R}^{k\times m}$ and the input y_{ref} is bounded.

Lemma 1.6 Assume $\|y_{ref}\| \leq 1$. If there exists $P \succ 0$ and a multiplier $\alpha > 0$ such that

$$\begin{bmatrix} A^\mathsf{T}PA - P + \alpha P & A^\mathsf{T}PB \\ B^\mathsf{T}PA & B^\mathsf{T}PB - \alpha I_{m\times m} \end{bmatrix} \preceq 0, \tag{7}$$

then the ellipsoidal set $p(P, c)(x)$ is invariant with respect to (6).

Proof. The proof is from Boyd et al. [14, p. 83]. Let $V(x) = x^\mathsf{T}Px$ for some $P \succ 0$. $V(x_+) - V(x) \leq 0$ implies that, for any x and y_{ref} satisfying $x^\mathsf{T}Px \geq 1$ and $y_{ref}^\mathsf{T}y_{ref} \leq 1$, $(Ax + By_{ref})^\mathsf{T} P (Ax + By_{ref}) - x^\mathsf{T}Px \leq 0$. Apply S-Procedure, we get if there exist multipliers $\alpha > 0$, $\beta > 0$ such that, for any x and y_{ref},

$$(Ax + By_{ref})^\mathsf{T} P (Ax + By_{ref}) - x^\mathsf{T}Px < 0 + \alpha (x^\mathsf{T}Px - 1) + \beta \left(1 - y_{ref}^\mathsf{T}y_{ref}\right) \leq 0, \tag{8}$$

then $\mathcal{E}_{P,1}$ is an invariant set. Rewrite (8) as a quadratic form on $\begin{bmatrix} x^\mathsf{T} & y_{ref}^\mathsf{T} \end{bmatrix}^\mathsf{T}$ with $\beta = \alpha$, we get the linear matrix inequality in (7).

Example 2.1.2 Consider a discrete-time Lur'e system [48], with matrices $A \in \mathbb{R}^{n\times n}$, $B \in \mathbb{R}^{n\times m}$, $C \in \mathbb{R}^{k\times n}$, $D \in \mathbb{R}^{k\times m}$, and a nonlinearity Δ.

$$\begin{aligned} x_+ &= Ax + B\left(y_{ref} - \Delta(y)\right), \quad x(0) = x_0 \\ y &= Cx. \end{aligned} \tag{9}$$

The second example contains a nonlinearity term $\Delta(y)$. The term $\Delta(y)$ can be used to model many nonlinear and uncertain behaviors that are present in a realistic control system. Examples of these nonlinear and uncertain behaviors include saturations, noise, high-frequency dynamics, hysteresis, time-varying parameters in the system matrices, etc. Consider (9) and assume Δ is the output of a saturation function on y. The saturation function has an upper and lower saturation level of $\Delta_{max} > 0$ and $\Delta_{min} = -\Delta_{max}$. Assume $|y| \leq y_{max} > \Delta_{max}$, we can capture the semantics of the saturation function with the quadratic constraint

$$\left(\Delta\left(y\right) - m_1 Cx\right)\left(\Delta\left(y\right) - m_2 Cx\right) \leq 0 \tag{10}$$

for $m_1 = \dfrac{\Delta_{max}}{y_{max}}$, and $m_2 = 1$. The quadratic constraint in (10) is a *sector-bound* inequality and it is illustrated in Figure 1.

Lemma 1.7 Given that $\Delta(y)$ satisfies the sector-bound inequality in (10), $\|y_{ref}\| \leq 1$, $\sigma = m_1 m_2$, and $\nu = \dfrac{1}{2}\left(m_1 + m_2\right)$, if there exists $P \succ 0$, multipliers $\alpha > 0$ and $\beta > 0$ such that

$$\begin{bmatrix} A^\intercal PA - P + \alpha P - \beta\sigma C^\intercal C & A^\intercal PB & -A^\intercal PB + \beta\nu C \\ B^\intercal PA & B^\intercal PB - \alpha I_{m\times m} & -B^\intercal PB \\ -B^\intercal PA + \beta\nu C^\intercal & -B^\intercal PB & B^\intercal PB - \beta I_{m\times m} \end{bmatrix} \preceq 0, \tag{11}$$

and $\sqrt{CP^{-1}C^\intercal} \leq y_{max}$, then the ellipsoid $p(P, c)(x)$ is invariant with respect to (9).

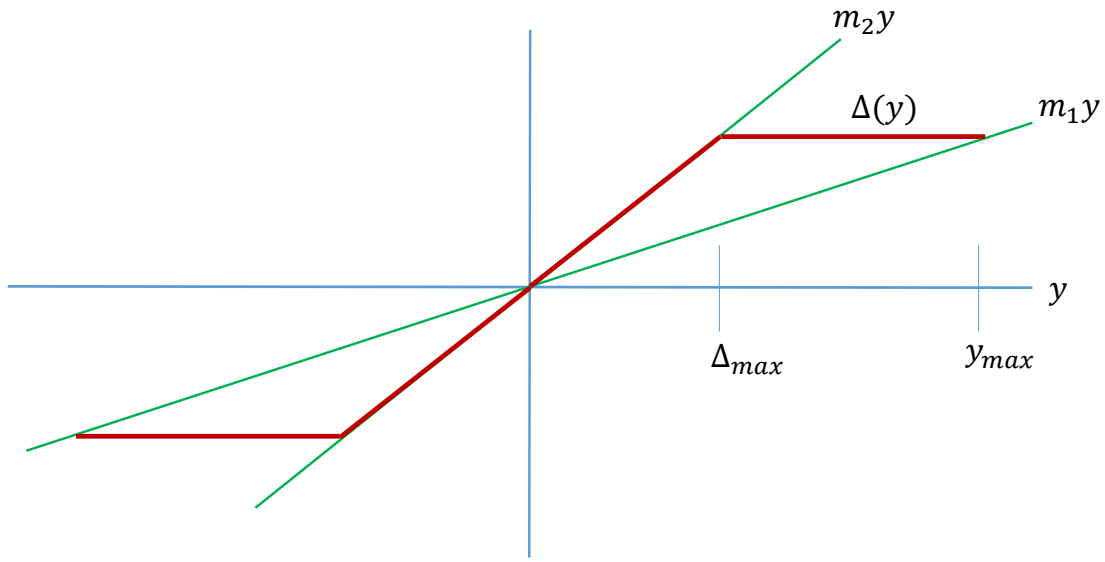


Figure 1: Approximating saturation function with a sector-bound inequality

Proof. The proof is similar to the one for lemma 1.6 but with the additional quadratic inequality

$$\begin{bmatrix} x \\ \Delta(y) \end{bmatrix}^{\mathsf{T}} \begin{bmatrix} m_1 m_2 C^{\mathsf{T}} C & -\frac{1}{2}(m_1 + m_2) C \\ -\frac{1}{2}(m_1 + m_2) C^{\mathsf{T}} & I_{m \times m} \end{bmatrix} \begin{bmatrix} x \\ \Delta(y) \end{bmatrix} \leq 0 \tag{12}$$

obtained from the sector-bound condition in (10).

The inequalities in (7) and (11) are more examples of LMIs. The first explicit mention of linear matrix inequalities in systems and control was by Yakubovich [105] in the 1960s. In the late 1980s, researchers realized that many system and control problems reduces to computationally efficient problems in the form of LMIs [14]. With modern digital computers, LMIs can be solved in practice [15]. If feasible, a solution $P \succ 0$ for either LMI in (7) or (11) can be computed using a semi-definite programming (SDP) solver such as SeDuMi [88].

2.2 Program Verification

We can show the stability of linear control systems and certain nonlinear control systems by computing a quadratic Lyapunov function. Now we want to extend this analysis down to the code level, in the form of a translated proof for code. A framework from formal program verification is introduced here, as it enables us to formulate an analogous proof of stability on the level of the code.

Axiomatic semantics is a method from computer science to assign mathematical meanings to programs through predicates about the program state that hold before the execution of the code and predicates that hold after the execution of the code [83]. A predicate that is expected to hold at a point in a program is called an assertion or an invariant. In axiomatic semantics, there is a language to express assertions about the program and followed by formal rules to prove the assertions. An example of language for expressing program assertions is first-order logic which is used extensively in this thesis. Here we introduce *Hoare logic* [42], which led to the notion of axiomatic semantics. The main structure in Hoare logic is the *Hoare triple*.

Definition 2.1 A Hoare triple is the 3-tuple $(\{P\}, C, \{Q\})$, in which $\{P\}$ is a predicate or a set defined by a logic formula, and $\{Q\}$ is also another predicate, and C denotes a block of code.

The symbol P denotes a pre-condition and the symbol Q denotes a post-condition.

Definition 2.2 A Hoare triple $(\{P\}, C, \{Q\})$ is interpreted to be *partially correct*, if $\{P\}$ holds before the execution of C, and $\{Q\}$ holds after the execution of C.

```
1 // abs(x)<=1;
2 while (x*x>0.5)  {
3   x=0.9*x;
4 }
5 // abs(x)<=1;
```

Figure 2: A `while` program in C

Remark 3 Program correctness requires a proof of termination. In the rest of this thesis, correctness only refers to the notion of partial correctness.

The pre and post-conditions are expressed on the code as comments before and after the block of code. For example, given the simple `while` program in Figure 2, If the set $|x| \leq 1$ holds before the execution of the loop, then it should hold for all executions of the loop.

Definition 2.3 An *loop invariant* is a predicate that holds before and for all executions of the loop.

The set $|x| \leq 1$ is a loop invariant. It can be inserted into the code as both the pre-condition and the post-condition, see the C comments in Figure 2. The Hoare triple in Figure 2, therefore is $\{|x| \leq 1\}$ `while` a do C end $\{|x| \leq 1\}$.

For a controller program implementing $x_+ = Ax + By$, such as the Matlab code in Figure 3, the loop invariant can be obtained from a Lyapunov stability analysis. Assuming bounded input u, we can compute an ellipsoidal set $\mathcal{E}_{P,1}$ that is invariant with respect to $x_+ = Ax + Bu$ by using lemma 1.6. The ellipsoidal set $\mathcal{E}_{P,1}$ is defined by the logic predicate $p(P, 1)(x)$. Note for the rest of this thesis, the predicate notation $p(P, 1)(x)$ will also be used to denote the set $\mathcal{E}_{P,1}$. The invariance of the set $p(P, 1)(x)$ enables us to express the logic formula $p(P, 1)(x)$ as a loop invariant for the Matlab

code. The result is the annotated Matlab program in Figure 3. Next, the basics of

```
% x'*P*x<=1;
while t<5000
  u=C*x+D*y;
  x=A*x+B*y;
end
% x'*P*x<=1;
```

Figure 3: Annotated Lead/Lag Compensator in Matlab

deductive program verification are described.

2.2.1 Hoare logic and Deductive Verification

Hoare logic is a formal proof system that comes with a set of axioms and inference rules for reasoning about the correctness of Hoare triples on various structures of an imperative programming language i.e. `if-else` statements, `assignment` statements, `while` statements, `for` statements, *empty* statements, etc.

For example, an axiom in Hoare logic for the `while` program construct is

$$\frac{\{P \wedge a\}\, C\, \{P\}}{\{P\}\ \texttt{while}\ a\ \text{do}\ C\ \text{end}\ \{\neg a \wedge P\}}. \tag{13}$$

Syntactically speaking, the axioms and inference rules can be parsed as follows.

1. The formula above the horizontal line implies the formula below that line.

2. The correctness of the formula below the horizontal line can be proved by showing the correctness of the formula above the line.

In the `while` axiom in (13), note that pre and post-conditions of the loop are necessarily the same. This requirement for a loop invariant is the key reason why program

Table 1: Hoare logic Inference Rules for a Imperative Language

$$\frac{\{P_1 \to P_2\}\, C\, \{Q_1 \to Q_2\}}{\{P_1\}\, C\, \{Q_2\}} \quad (14) \qquad \frac{\{P\}\, C_1\, \{R\}\,;\{R\}\, C_2\, \{Q\}}{\{P\}\, C_1; C_2\, \{Q\}} \quad (15)$$

$$\frac{}{\{P\}\, SKIP\, \{P\}} \quad (16) \qquad \frac{}{\{P[e/x]\}\, x := expr\, \{P\}} \quad (17)$$

verification is difficult. Some of the basic inferences rules for reasoning about imperative programs using Hoare logic are listed in Table 1. The consequence rule in (14) is used when a *stronger* pre-condition or *weaker* post-condition is needed. The set defined by the stronger condition is a subset of the set defined by the weaker condition. For example, $x > 0$ is a stronger pre-condition than $x \geq 0$. The substitution rule in (17) is used when the code is an *assignment* statement. The weakest pre-condition expression $P[x/expr]$ in (17) means P with all free occurrences of the expression *expr* replaced by x. For example, given a post-condition `y<=1` for the line of code `y=x+1`, one can deduct that `x+1<=1` is a weakest pre-condition using the backward substitution rule in (17). The skip rule in (16) can be used when the executing piece of code does not change any variables in the pre and post-conditions.

To verify the Hoare triple in Figure 3, use the inference rules from Table 1 on the code, starting from the post-condition `x*P*x<=1`. The process generates an alternate pre-condition $p(P, 1)(A * x + B * y)$ for the loop body. By the consequent rule, the correctness of the initial Hoare triple can be checked by checking if $p(P, 1)(x1) \rightarrow p(P, 1)(A * x + B * y)$. The process in Figure 4 is deductive. An algorithmic reformulation of it is Dijkstra's work on Predicate transformers [30]. By using the Predicate transformers, the deductive process of Figure 4 is reduced to a computational process of checking formulas expressed in prepositional logic.

1. $\{p(P,1)(x)\}$ `while` a do C end $\{p(P,1)(x)\}$.
2. $\{p(P,1)(x)\}\, C\, \{p(P,1)(x)\}$ by the `while` axiom in (13).
3. $\{p(P,1)(A*x+B*y)\}\, x = A*x+B*y\, \{p(P,1)(x)\}$ by the backward substitution rule in (17).
4. $\{p(P,1)(A*x+B*y)\}\, u = C*x+B*y\, \{p(P,1)(A*x+B*y)\}$ by the skip rule in (16).
5. $\{p(P,1)(x), p(P,1)(A*x+B*y)\}\, C\, \{p(P,1)(x)\}$ by the composition rule in (15).
6. if $p(P,1)(x) \to p(P,1)(A*x+B*y)$, then $\{p(P,1)(x)\}\, C\, \{p(P,1)(x)\}$ by the consequent rule in (14).

Figure 4: Correctness of the program Using Hoare logic

2.2.2 Predicate Transformers

In the semantics of Predicate Transformers, the *weakest pre-condition* of C is a function wp that maps any post-condition Q to a pre-condition. The output of the weakest pre-condition function $wp(C,Q)$ is the largest set such that, after the execution of C, Q holds. For example, the correctness of a Hoare triple, for a set of variables x in the code C, is determined by checking if the logic formula $\forall x, P \to wp(C,Q)$ holds. The wp function can be applied to various constructs in an imperative programming language. Some examples are given in Table 2. The sequence of I_i in (21) can be replaced by a single I if I is an invariant of the loop. Denote the `while` program as $\mathcal{P}$, in the case of partial correctness, $wp(\mathcal{P},Q) = I$ is the *weakest literal pre-condition* if $I \to wp(C,I)$. In the case of total correctness, $wp(\mathcal{P},I) = I$ is the weakest pre-condition, if $I \to Q$ and the loop terminates. Recall the control program in Matlab

Table 2: Weakest Pre-condition Calculus

$$wp(C_1;,...,C_N,Q) = wp(C_1, wp(C_2, wp(C_3, ..., wp(C_N, Q)) \tag{18}$$

$$wp(\textbf{skip}, Q) = Q \tag{19}$$

$$wp(x := e, Q) = Q[e/x] \tag{20}$$

$$\begin{aligned} wp(\texttt{while}\ a\ \text{do}\ C\ \text{end}, Q) = \forall i \in \mathbb{N}, I_i \\ I_0 = true \\ I_{i+1} = (\neg a \to Q) \wedge (a \to wp(C, I_i)) \end{aligned} \tag{21}$$

from 3, which consisted of a `while` loop and satisfies the invariant $p(P,1)(x)$, Apply wp-calculus to that program i.e $wp(\mathcal{P}, p(P,1)(x)) = p(P,1)(x)$ leads to two logic formulas:

1. $I \to Q$ and the loop terminates. The loop in 3 terminates after a finite number of iterations and clearly $p(P,1)(x) \to p(P,1)(x)$ is true.

2. $I \to wp(C, I)$ i.e. $p(P,1)(x) \to wp(C, p(P,1)(x))$.

The second condition is harder to verify since $p(P,1)(x) \to wp(C, p(P,1)(x))$ involves checking if one quadratic inequality implies another. For programs that are purely linear transformations, checking $p(P,1)(x) \to wp(C, p(P,1)(x))$ might be automatic using state of art SMT solvers, but there are nonlinearities in the example control systems under consideration. Notice the formula $p(P,1)(x) \to wp(C, p(P,1)(x))$ is equivalent to the Hoare triple

$$\{p(P,1)(x), wp(C, p(P,1)(x))\, C\, \{p(P,1)(x)\}\,, \tag{22}$$

which means $p(P,1)(x)$ can be inserted as the pre and post-conditions of the loop body C in Figure 3. Applied further *wp*-calculus on the loop body results in the annotated code in Figure 5. The set of pre-conditions generated by *wp*-calculus i.e.

```
% x'*P*x<=1;
while (t<5000)
% x'*P*x<=1, wp(u=C*x+D*y,wp(x=A*x+B*y,x'*P*x<=1))=(A*x+B*y)'*
    ↪ P*(A*x+B*y)<=1
  u=C*x+D*y;
% wp(x=A*x+B*y,x'*P*x<=1)=(A*x+B*y)'*P*(A*x+B*y)<=1
  x=A*x+B*y;
% x'*P*x<=1
end
% x'*P*x<=1;
```

Figure 5: Annotated Lead/Lag Compensator in Matlab

the displayed Matlab comments inside the loop in Figure 5, along with the Matlab code itself, form a translated proof for the code. Verifying this translated proof implies that $p(P,1)(x)$ is a loop invariant of the `while` program, which is an evidence that the controller implementation is stable.

2.2.3 Strongest Post-condition

The dual of weakest pre-condition is the *strongest post-condition*. The strongest post-condition function *sp* on C maps pre-condition P to the strongest post-condition that holds after execution of C. The strongest post-condition $sp(C,P)$ is the smallest set that holds after the execution of C, given that the set defined by P holds before the execution of C. To check a Hoare triple $\{P\}\,C\,\{Q\}$ using *sp*-calculus, first compute $sp(C,P)$ or some over-approximation of $sp(C,P)$ and then check that $sp(C,P) \rightarrow Q$. In this thesis, *sp*-calculus is used to generate the proof on the code. Traditionally, the weakest pre-condition calculus is used to verify program since the automatic

computation of strongest post-condition is rarely feasible. For the control systems under consideration, we can apply ellipsoidal transformation rules, which allow us to perform *sp*-calculus automatically.

2.2.4 Verifying the Proof on the Code Using a Theorem Prover

A theorem prover or a proof assistant is a computer program that provides an environment where mathematical theories can be expressed and then proved using an interactive procedure. The soundness of a theorem prover is based on the collection of accepted axioms on which the theories are built upon. A theorem prover is interactive, whereby a human user has to input a proof step and the theorem prover checks the correctness of the proof step. A theorem prover can be automatic in the sense that, if provided with a right set of theories and strategies, it can check automatically the correctness of a formula. A theorem prover, however cannot generate a proof of a theory automatically for all but the most trivial ones. In this thesis, the translated proof on the code is verified by a proof-checking program, based on the theorem prover Prototype Verification System (PVS) [68]. The proof-checking program is provided by the thesis of Romain Jobredeaux [46]. It uses the theories and definitions from the NASA PVS linear algebra library [40].

2.2.5 Summary

Here we summarize the process of extending Lyapunov stability analysis down to the code-level as illustrated by the Matlab example in Figure 5.

1. Lyapunov stability analysis is performed to compute an ellipsoidal invariant.

2. The ellipsoidal invariant set $\mathcal{E}_{P,1}$ is translated into the first-order logic formula $p(P,1)(x)$, and inserted as a pre-condition and a post-condition of the loop body.

3. The rest of the pre- and post-conditions for each line of code in Figure 5 are auto-generated using *sp*-calculus.

4. The translated proof is comprised of all the pre- and post-conditions, displayed as Matlab comments in Figure 5 and the program $\mathcal{P}$ itself.

5. The correctness of the Hoare triples in Figure 5 implies that $p(P,1)(x)$ is a loop invariant of $\mathcal{P}$, and in turn this means $\mathcal{P}$ is stable.

6. Checking Hoare triples with quadratic invariants can be automated in a theorem prover [46], by first proving the ellipsoid transformation theories used in *sp*-calculus are correct, and then applying the ellipsoid transformation theories on the Hoare triples extracted from the annotated code.

Remark 4 From the last summary statement above, the need for an independent verification of all the theories used in the proof translation process before using the same theories to check the translated proofs, is very important. This ensures a degree of separation between the proof providers i.e. the producers of safety-critical software and the proof checkers i.e. the certification authorities.

```
1 /*@
2     requires x*x<=1;
3     ensures x*x<=1;
4 */
5 while (x*x>0.5) {
6   x=0.9*x;
7 }
```

Figure 6: ACSL annotations for a `while` loop Program

2.2.6 Annotation Language for Expressing Axiomatic Semantics of C Programs

A prototype is developed in this thesis to automate the translation process, starting from a high-level modeling language and ending with the C language. The C language is chosen as the output language of the prototype because of its industrial popularity and because there is a relatively popular formal annotation language developed for C. The ANSI/ISO C Specification Language (ACSL [8]), which is a formal specification language for C programs, is supported by the static analyzer frama-C [7]. The ACSL annotations are expressed in special C comments denoted by the symbol `/*@`. The main structure of ACSL is the *function contract*. A *function contract* consisted of a set of requirements i.e. pre-conditions on the arguments to a function and/or another set of properties that are ensured after the execution of the function i.e. post-conditions. A *function contract* is inserted before a function C to form a Hoare triple $\{P\}\, C\, \{Q\}$. The pre and post-conditions in an ACSL function contract are denoted respectively using the ACSL keywords *requires* and *ensures*.

For example, inserting the invariant `x*x<=1` as a ACSL function contract into the C program from Figure 2 results in the ACSL-annotated program in Figure 6. The ACSL comments are referred to as code specifications or annotations.

```
1  /*@
2      behavior one:
3      assumes x<=2;
4      ensures x*x<=4;
5
6      behavior two:
7      assumes x<=1;
8      ensures x*x<=1;
9  */
10 while (x*x>0.5) {
11   x=0.9*x;
12 }
```

Figure 7: ACSL Behaviors

One can have multiple *behaviors* on the code, denoted by the ACSL keyword *behavior*.

Definition 2.4 A behavior is defined as a set of invariants that the program satisfies for some given set of assumptions.

For example, if one take the same code from Figure 6, but assumes two different initial values of x. The result is two sets of invariants for the code (see Figure 7).

In ACSL, one can also express annotations on ghost code, which are annotative code. Ghost code is denoted using the ACSL keyword *ghost*. For example, in Figure 8 the `while` loop from the previous examples can be expressed as a ghost code, and the same invariant `x*x<=1` can be inserted as a property of the ghost code. The ghost code construct is useful for expressing properties that depend on the environment. For example, some properties in control, such as the stability of the closed-loop system, depends on the model of the plant. Using the ACSL *ghost* keyword, the plant model can be embedded into the annotations and used as part of the analysis to prove closed-loop stability on the code. More descriptions of ACSL are provided in the

```
/*@
  requires x*x<=1;
  ensures  x*x<=1;
*/
{
  /*@
    ghost  while (x*x>0.5) { x=0.9*x;}
  */
}
```

Figure 8: ACSL Ghost Code

next chapter.

Chapter III

CREDIBLE AUTOCODING

Autocoding is an automated programming process that transforms a system expressed in a high-level modeling language such as Simulink or SCADE into a low-level implementation language such as C. In *credible autocoding*, the code is generated along with mathematically verifiable guarantees of functional correctness. The concept of credible autocoding is analogous to Rinard's credible compilation in [74]. Both processes generate formally verifiable evidences that the output correctly preserves certain semantics of the input. Unlike credible compilation of Rinard's, the formally verifiable evidences of interest in this research are the high-level functional properties of control systems which include stability, robustness and performance. While high-level functional properties allows for a in-depth understanding of the underlying behavior of the software, they can also be used to prove the absence of runtime errors such as divide by zero [75].

3.1 Credible Autocoding Framework

Data-flow modeling languages such as Simulink or SCADE are the default industry choice for Model-based development of safety-critical control systems. In a data-flow language environment such as Simulink, there are two major elements: "blocks", and "lines." The blocks are functions that perform some operations on its input(s) and then output the result(s). The lines are directed edges that flow from an origin block's

output to a destination block's input.

Within this framework of software development, systems are built using a language of high-level abstraction in order to facilitate rapid design and prototyping. The source code is then generated automatically from the input model using an automated code generation tool or an autocoder. The trustworthiness of the autocoder has often been questioned in the industry [28]. A related work [29], which is complementary to this research used a model-based approach (meta-model approach since its applied towards a model-based development tool) to assign provably correct semantics to a set of Simulink blocks. The result of that research is a library of trustworthy blocks i.e. the *BlockLibrary* language, with precise semantics, that can be reasoned about formally.

In the framework of credible autocoding, instead of proving that individual block transformations are correct i.e. building a collection of trustworthy blocks, the goal is to be able to show that the output code also satisfies the high-level functional properties of the input model. The functional properties of the input model are dependent on the domain of the input model. In the domain of control systems, a strong functional property is the stability of the closed-loop system and a weaker property is the boundedness of the state of the system. The verification of the code against high-level functional properties gives an additional layer of guarantee on the correctness of the code. For example, if a gain in a Simulink model was inverted accidentally before autocoding, the output code, while correct in the sense of each individual block transformations, is not likely to satisfy a pre-computed property such as the Lyapunov stability of the system.

The current safety-critical software development process is based on the traditional V-cycle in Figure 9. The V&V activities on the right is generating simulation results which have to satisfy the testing requirements written by the developers on the left. For example, at the model level description of the software, individual functions of the program are specified and units tests are developed for them.

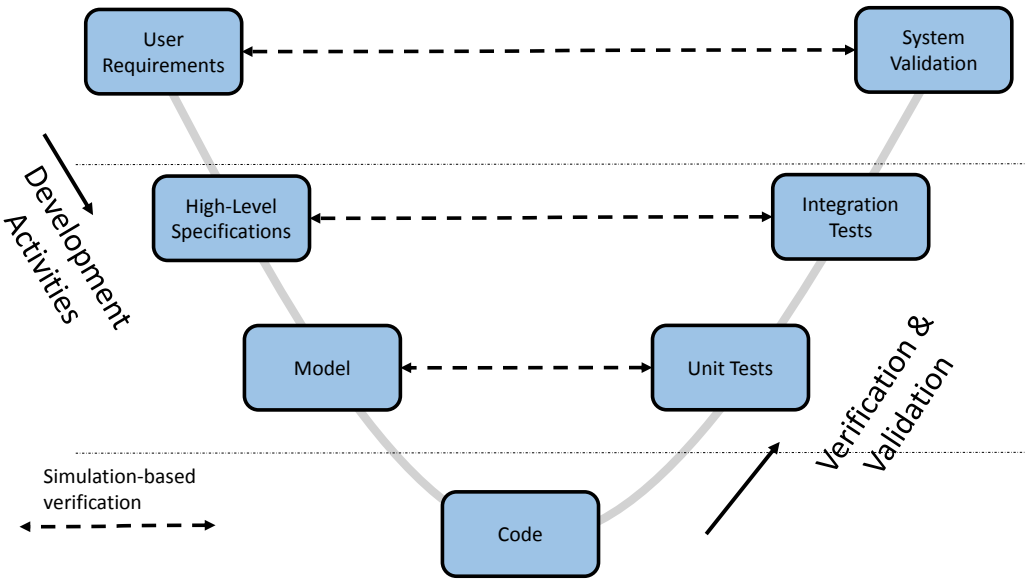


Figure 9: Safety-critical software development process

A problem with the traditional V-cycle is that V&V do not occur until well after the system is translated either manually or automatically into code. Furthermore, the people performing the V&V are a different group of specialists than the people doing the development. They have little if any domain-specific knowledge about the system being developed. These knowledge could provide useful information about the correctness of the produced code but they are often lost in the code generation process. This time and communication gap between V&V and development activities worsens as the process move into the last phases of the V-cycle. For example, critical problems

discovered in the integration testing phase could result in the process starting over at the high-level specifications phase. This slow and expensive feedback loop is one of the reason for the explosion in the cost of safety-critical software development.

In this dissertation, a framework of credible autocoding is developed, which aims at reducing the time and communication gap between V&V and software development. An idealized vision of this framework is shown in Figure 10, where the high-level and low-level specifications, along with their requirements and properties are translated into code and proofs on the code, which can then be independently and automatically checked by a proof-checker. In this idealized scenario, the time horizon between the

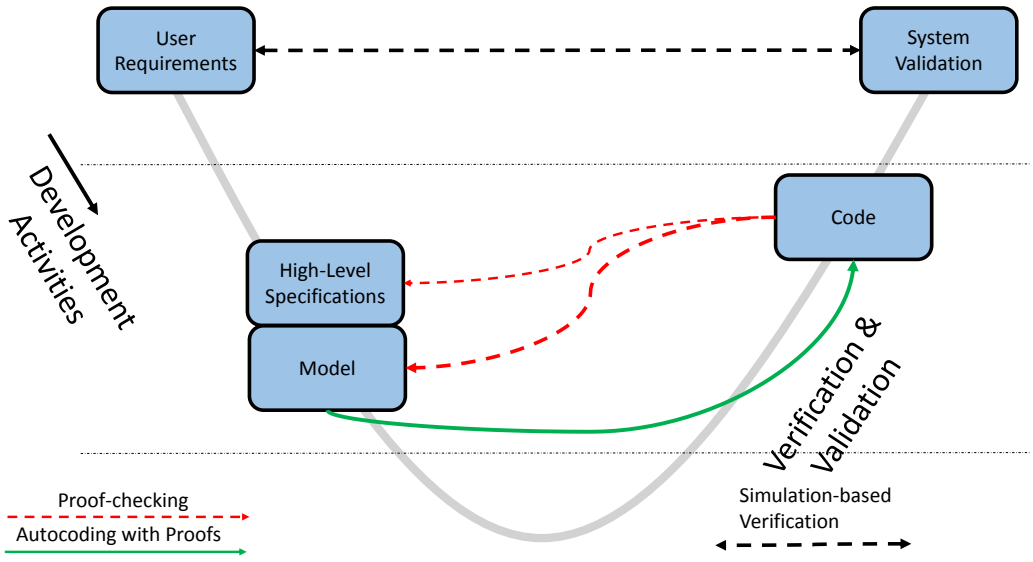


Figure 10: Safety-critical software development process with credible autocoding

user requirements phase and the system validation phase is significantly reduced as the unit and integration testing phases are supplanted by an automatic, proof based process.

In this thesis, credible autocoding is demonstrated on the domain of control systems. This process is summarized in Figure 11. The *control semantics* include the stability property of the system and the plant models used in the stability analysis for the closed-loop cases. The scope of this thesis is concerned with the left half of the diagram in Figure 11, which is the credible autocoding portion. The framework adds, on top of a classic model-based development cycle, another translation process, that converts quadratic invariant sets, computed using Lyapunov-based methods, into code annotations on the code, which form a proof of the correctness of the output code. The work in this thesis is a first proof of concept and presumably it can be

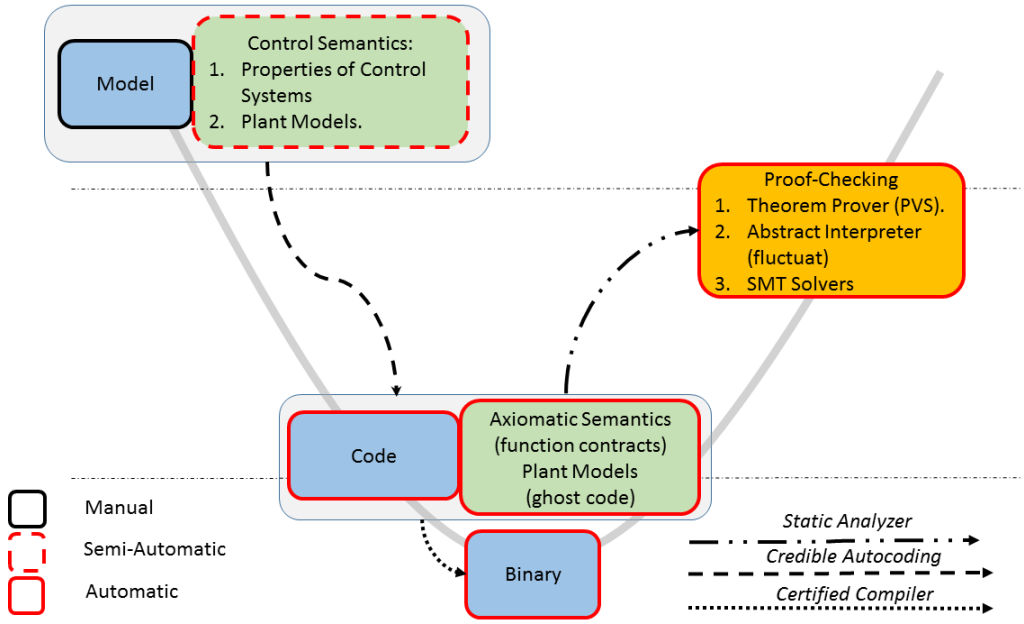


Figure 11: Automated Credible Autocoding/Compilation Chain for Control Systems

further extended to other domains such as convex optimization [98]. The code annotations include both the axiomatic semantics described in Chapter 2, and *ghost code*, which are non-executed code. The ghost code is useful for representing the models of

the plant, which are not part of the semantics of the program. For credible autocoding of control software, compared against the traditional model-based development, the only additional requirement on control engineers is that they provide the Lyapunov function. The procedure for generating Lyapunov-type certificate of stability and performance property of control systems can be mostly automated using LMIs and the IQC framework. Each stability and performance property generated can be encoded using an ellipsoid, which can then be transformed into a quadratic invariant for the code.

The advantages of the framework developed in this research can be summarized as follows.

1. All the advantages of model-based development are inherited.

2. The correctness of the autocoder is guaranteed by the correctness of its output code. This is based on the idea of credible compilation in [74]. This replaces the need to formally verify the autocoder.

3. Under credible autocoding, an independent verification of the produced code is possible. This in contrast to the certified autocoder approach.

4. The properties carried in credible autocoding can be used to evince both safety and liveness.

5. Credible autocoding provides guarantees of high-level functional properties, which is a more useful characterization of the correctness of the whole system for the certification authorities.

6. The framework could generate feedback information to the domain expert so errors in the construction of the model could be diagnosed more rapidly.

7. Credible autocoding could reduce the number of tests required for certification of the control software. In traditional unit testing, a piece of code, such as a controller function, is tested repeatedly for many possible different inputs and scenarios. The credible autocoding framework enables a meta-testing procedures, in which the function, is tested for all possible inputs and scenarios, in one iteration.

3.1.1 Prototype Tool-chain

In this research, a prototype tool-chain has been developed for the demonstration of credible autocoding. The prototype tool-chain is split into a credible autocoder frontend and a proof-checker backend. The credible autocoder frontend translates the model into annotated code. The proof-checking backend analyzes the annotated code produce by the frontend and decides whether or not the proof is coherent. The scope of this thesis falls on the frontend. The backend, which is developed in thesis of Romain Jobredeaux [46] is also briefly mentioned in the context of proof-checking the generated code.

3.1.1.1 Input Language

The input language of the framework should be a graphical data-flow modeling language such as Simulink, since it is familiar to control engineers. The exact choice of the input language is up to the domain users' preference and does not affect the utility of the framework as it can be eventually adapted to other modeling languages such

as SCADE [11]. For the prototype tool-chain developed in this research, the choice of the input language is a discrete-time subset of Simulink blocks, which includes basic blocks such as unit delays, gain, input, output, plus, minus, multiplication, divide, min, and max. This subset is sufficient to express any control systems of interest.

3.1.1.2 Language Extensions in Gene-Auto

The autocoding prototype is based on Gene-Auto [92, 45, 91, 12], which is an existing, open-source, autocoding prototype for embedded systems. The prototype required language extensions in the Simulink environment, the Gene-Auto environment and ACSL. The language extensions are summarized as follows.

1. A library of Annotation blocks in Simulink/Gene-Auto.
2. An ACSL-like language within Gene-Auto.
3. Abstract types and their operators in ACSL: matrix, vector and quadratic predicates.

The language extensions in Simulink/Gene-Auto and the ACSL abstract types are described in Section 3.2. The syntax of the ACSL abstract types are briefly described. The ACSL-like extensions within Gene-Auto is called *GAVAModel*. For more details of *GAVAModel*, including its meta-model, please see `block-library.enseeiht.fr/html/Progress/geneautoAnnot.html`. The *GAVAModel* language enables common ASCL constructs such as: *behavior*, *assumes*-statement, *function contract*, *require*-statement, *ensure*-statement, and *ghost code* to be expressed within an *intermediate representation* in Gene-Auto.

3.2 Language Extensions

An observer (see [101]) in Simulink takes an input signal and returns an output of 1 if the input satisfies a specific property, and 0 if otherwise. Both boundedness and stability can be expressed, for example, using an observer with inputs $x(i), i = 1, \ldots, n$, $P \succ 0$,

$$x \rightarrow \sum_{i,j=1,\ldots,n} x(i)P(i,j)x(j) \leq 1. \tag{23}$$

To express observers as annotations on the Simulink model, we extended the Simulink language and the Gene-Auto environment with a set of annotation blocks.

3.2.1 Annotation Blocks for Simulink

The Simulink language and Gene-Auto are extended by an annotation block library. Annotation blocks have the same structure as the regular blocks but they are ignored during code generation. The annotation block library is sufficient to express the stability of linear systems and nonlinear systems. They can also express the semantics of observer-based fault-detection systems.

The prototype annotation block library contains four symbols: *vamux*, *constant*, *quadratic*, and *system*. Each annotation symbol denotes an annotation block type, To illustrate the annotations blocks, we have Figure 12, which shows a Simulink model of an engine controller, along with 6 annotation blocks. The annotation blocks are highlighted in red for the purpose of clarity.

In Simulink, the *vamux* block type takes n scalar or vector inputs x_i, and outputs a concatenated signal $y = \begin{bmatrix} x_1^\mathsf{T}, & \ldots, & x_n^\mathsf{T} \end{bmatrix}^\mathsf{T}$. In Figure 12, there are three *vamux* blocks, labeled as `nh`, `xc` and `ybar`. The *vamux* block type only accepts one parameter, which

determines the number of inputs to the block type. The *vamux* block does not express any property of the system. In Gene-Auto+, the main functionality of the *vamux* block is to establish equivalence relations between its inputs and the ith entry of its output. i.e. $x_i == y_i$. This enables the prototype to replace the pseudo-variables in the templates within the other annotation blocks with the actual variables from the code.

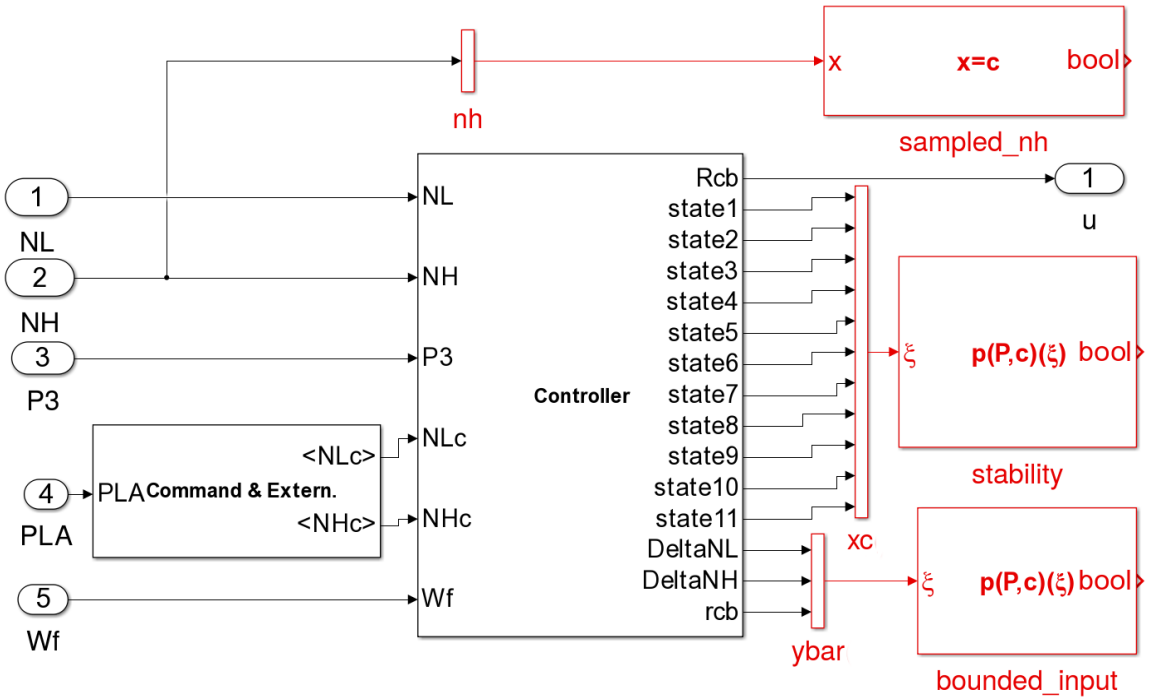


Figure 12: Simulink Model with annotation blocks

The *constant* block type accepts one scalar, vector, or matrix input x, and a constant matrix parameter $[c_1]$ or $[c_1, \ldots, c_n]$ for $n \in \mathbb{N}$. The type of the constants c_i are constrained to be the same type as the input x. The output of the block is the boolean value $x == c_1$ or $\bigvee_{i=1}^{n} (x == c_i)$, which implies n sets of *behaviors* for the code.

Definition 2.1 A behavior is a set of unique assumptions on the parameters and

input, and output of the model.

This block type is useful for expressing the semantics of parameter varying systems such as a gain-scheduled controller. For example, the scheduling parameter of the controller in Figure 12 is the input NH, which is annotated with a *constant* block labeled `sampled_nh`. In Gene-Auto+, the *constant* block type generates a set of assumption(s) $\{x == c_i\}$, $i = 1, \ldots, n$.

The *quadratic* block type accepts a vector input $\xi \in \mathbb{R}^n$, a matrix parameter $P \in \mathbb{S}^{n \times n}$, a logic connective symbol $\diamond \in \{<=, <, >, ==\}$, a level-set constant $c \in \mathbb{R}$, and outputs the boolean value of $\xi^\intercal P \xi \diamond c$. The *quadratic* block type can be used, for example, to express ellipsoidal invariant sets, sector-bound inequalities, 2-norm squared, sum of squares polynomial sets, etc. The *quadratic* block also accepts a positive scalar parameter *mu*. This is used to indicate the multiplier computed in stability analysis. The *quadratic* block type behaves like the observer from (23) in Simulink. In Gene-Auto+, the *quadratic* block type represents a quadratic predicate on its inputs: $\forall \xi . \xi^\intercal P \xi \leq c$. An example of *quadratic* block can be found in Figure 12, where the block `stability` express a quadratic invariant on the input signal *xc*. The other *quadratic* block `bounded_input` is used to express a bound on the input *ybar* to the controller.

The *system* block type is parameterized by 4 matrices A, B, C, and D. An example of a Simulink model annotated with the *system* block can be found in Section 3.2.3. The *system* block type accepts two vector inputs u and y. The output of the *system*

block type is the state x of the dynamical system

$$\begin{aligned} x_+ &= Ax + Bu, \quad x(0) = x_0 \\ y &= Cx + Du. \end{aligned} \tag{24}$$

The semantics of the *system* block in Gene-Auto include the semantics of the discrete-time linear state-space system in (24), and a set of relations $\{\tilde{y}_i == y_i, u_i = \tilde{u}_i\}$ that establish equivalence between the annotation variables y and u and their corresponding variables $\tilde{y}$ and $\tilde{u}$ from the controller model. The *system* block type can be used, for example, to express a model of the plant the controller is expected to interact with. The same controller model can be annotated with multiple *system* blocks, which results in multiple sets of predicates for the code, which can be annotated using the *behavior* keyword from ACSL.

3.2.2 Annotation Blocks and Behaviors in the Model

In a model, multiple *system* blocks $s_1, \ldots, s_n$ can be connected to the same set of *vamux* blocks. This results in a set of n behaviors expressed by the formula $\bigvee_i^n s_i$. If there are n *system* blocks connected to the controller model, then there are n behaviors in the model.

If there are also k *constant* blocks in the model, each connected to a different *vamux* block, and each with m behaviors, then we have a total of m^k behaviors resulting from the *constant* blocks: $\bigwedge_i^k \left(\bigvee_i^m c_i \right)$. The complete set of behaviors in the model resulting from both the *system* and *constant* blocks is described by the formula

$$\left(\bigwedge_i^k \left(\bigvee_i^m c_i \right) \right) \wedge \left(\bigvee_i^n s_i \right) \tag{25}$$

or a total of nm^k possible behaviors.

Lastly, if there are w *quadratic* blocks in the model as well, and they are all connected to the same set of *vamux* block, then we have w number of behaviors $\vee_i^w q_i$ due to the *quadratic* blocks. Combining this set of behaviors conjunctively with the set of behaviors generated by the *system* and *constant* blocks results in

$$\left(\bigwedge_i^k \left(\bigvee_i^m c_i\right)\right) \wedge \left(\bigvee_i^n s_i\right) \wedge \left(\bigvee_i^w q_i\right) \tag{26}$$

for a possible total of wnm^k behaviors in the model. However, each of the *quadratic* blocks that encode an inductive property such as stability are assigned a behavior produced by a *system* block. This is true for any examples, in which the quadratic invariant is computed based on some plant model. For example, if there are n quadratic invariants and each is assigned a behavior from a *system* block, then there are only n behaviors in the model:

$$\bigvee_i^n \left(s_i \wedge q_i\right). \tag{27}$$

Next, some annotated examples are given. Each example contains a different possible set of control semantics.

3.2.3 Closed-Loop Stability of a Linear System with Bounded Input

The closed-loop stability of a control system with bounded input can be expressed with a *system* block and a pair of *quadratic* blocks. An example of such is displayed in Figure 13.

1. The *quadratic* block `stability` is used to express the ellipsoidal invariant that encodes the closed-loop stability of the system.

2. The *quadratic* block `bounded_input` is used to express a 2-norm bound on the input.

3. The *system* block `plant` is used to express the discrete-time linear state-space system used in the closed-loop stability analysis.

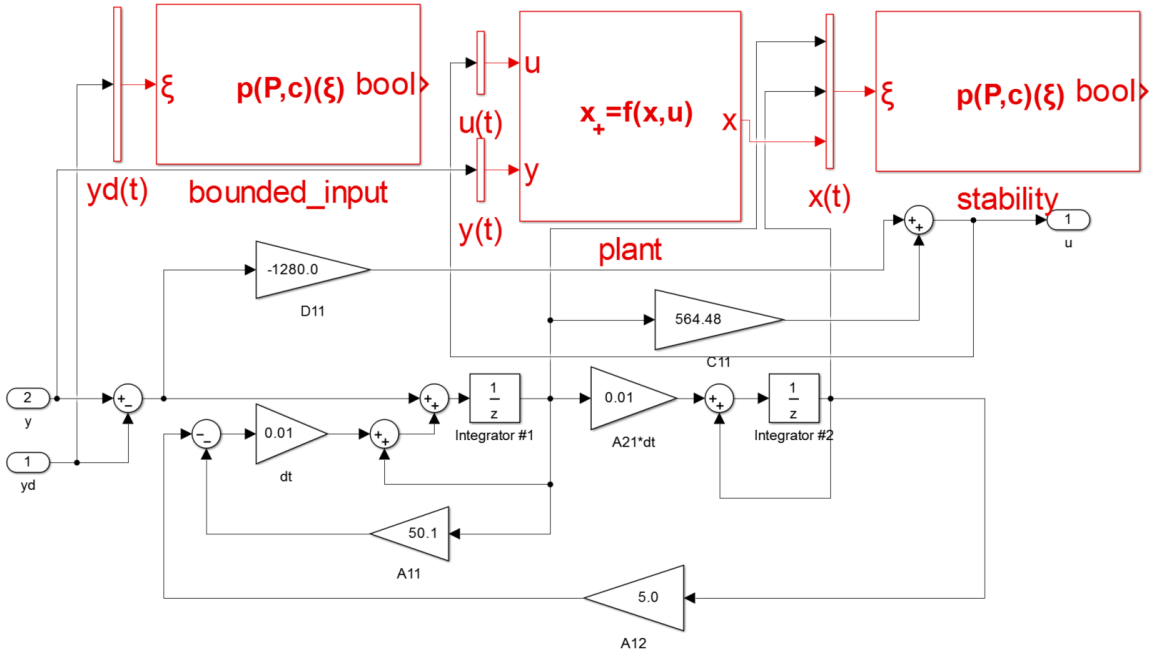


Figure 13: Control system model annotated with control semantics

3.2.4 Open-loop Stability of a Control System with Saturations

Saturations are present in many realistic control systems. As described in Section 2.1.2, their semantics can be over-approximated using a sector-bound inequality. The sector-bound inequality, being quadratic, can also be expressed with a *quadratic* block. For the altitude controller in Figure 14, obtained from NASA's transport class model [43], the relations between the inputs and outputs of the saturations are captured using a single sector-bound inequality. This sector-bound inequality is expressed by the *quadratic* block `sector` in Figure 14.

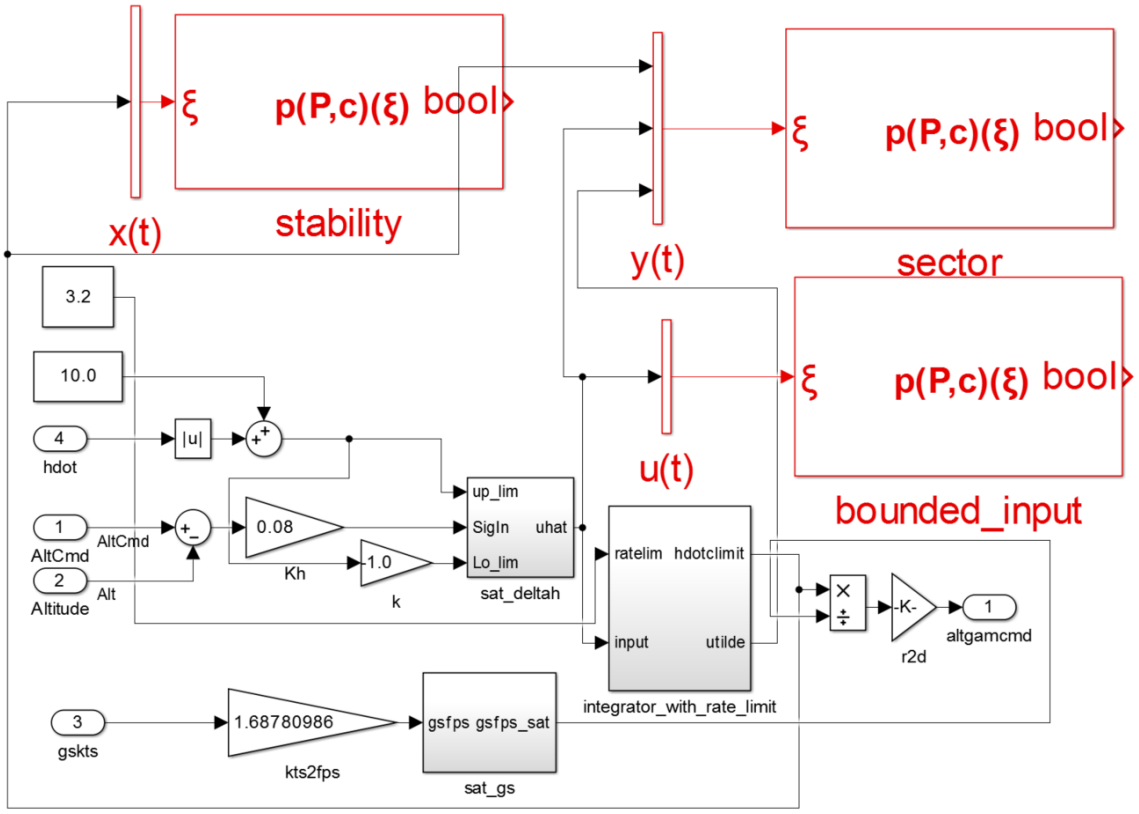


Figure 14: Sector-bound condition for saturation operators in an altitude controller

3.2.5 Expressing Semantics of Observer-based Fault-detection Systems in Simulink

In an observer-based fault-detection system, the dynamics of the observer are designed such that the output of the observer changes if the plant model changes or is subject to a malicious attack. Once the change exceeds a certain pre-defined threshold, the system is said to be in the faulty mode. To express the faulty and nominal behavior of a fault-detection system, one can use two different *system* blocks. One *system*

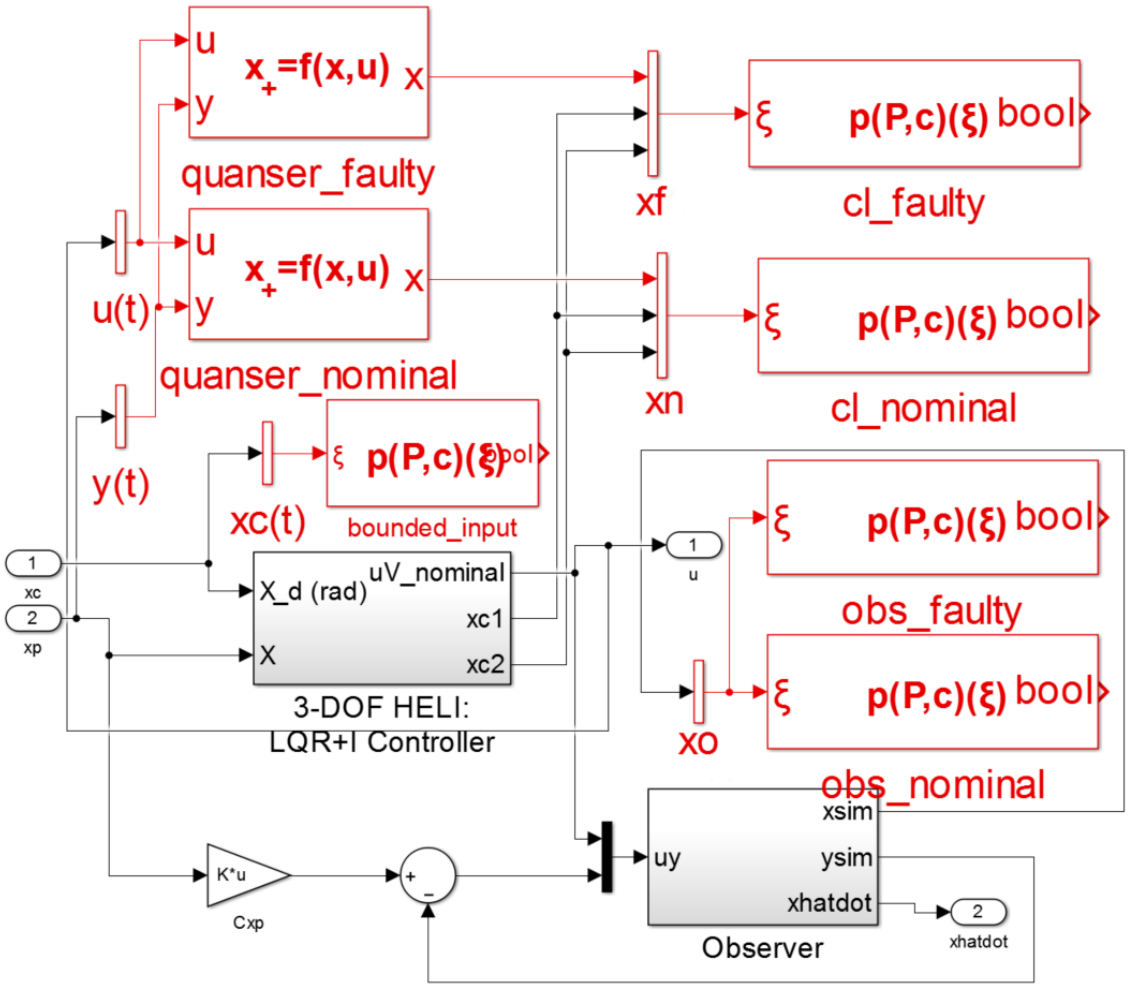


Figure 15: Expressing multiple behaviors: observer-based fault-detection system

block is the model of the faulty plant that is predicted to trigger the faulty mode

and the other is the nominal plant. This is displayed in Figure 15. The *quadratic* blocks connected to the *vamux* blocks `xf` and `xn` express the closed-loop stability of the system. They are assigned behaviors based on their physical connections to the *system* block. For example, as displayed in Figure 15, the block `cl_faulty` is connected to the *system* block `quanser_faulty` using the *vamux* block `xf`. The two *quadratic* blocks connected to the *vamux* block `xo` are used to express the stability of the observer dynamics. They are assigned the behaviors $faulty$ and *nominal* based on the labels in their names.

3.3 Credible Autocoding of Control Software

This section describes the credible autocoding process in a nutshell for a simple dynamical system, using a mixture of math, C and pseudo-ACSL. We make the following assumptions on the credible autocoding of control semantics. The C code is executed on a machine $\mathcal{M}$ capable of infinite precision arithmetic. This assumption is discarded in Chapter 5 of this thesis, where floating-point computation errors are accounted for.

The process starts with computing a quadratic invariant set for the system. Given a dynamical system $\mathcal{G}$ defined by $x_+ = Ax$, the ellipsoid set $p(P,1)(x)$, constructed by solving the LMI $A^\intercal PA - P \prec 0$ for $P \succ 0$, is also invariant with respect to $\mathcal{G}$. The invariance of $p(P,1)(x)$ enable us to know a priori that the Hoare Triple $\{p(P,1)(x)\}\,\mathcal{P}\,\{p(P,1)(x)\}$, in which $\mathcal{P}$ is a code implementation of $\mathcal{G}$ in Figure 16, is correct. Since $P \succ 0$ is invertible, then $q(Q,1)(x)$ with $Q = P^{-1}$ is equivalent to $p(P,1)(x)$. The credible autocoder inserts $q(Q,1)(x)$ as the pre- and post-condition of the program.

Using the weakest literal pre-condition function from (21) on $q(Q,1)(x)$, one obtains $q(Q,1)(x)$ as the pre- and post-condition of the loop body in $\mathcal{P}$. The quadratic invariant $q(Q,1)(x)$ is inserted into the code in Figure 16 as pre- and post-condition of the loop body. This is displayed in lines 7 and 8 of Figure 16, with the loop body enclosed in curly braces.

```
1  /*@
2    requires q(Q,1)(x);
3    ensures  q(Q,1)(x);
4  */
5  while (true) {
6  /*@
7    requires q(Q,1)(x);
8    ensures  q(Q,1)(x);
9  */
10  {
11   y1=0.4990*x1+0.1*x2;
12   y2=0.01*x1+1.0*x2;
13   x1=y1;
14   x2=y2;
15  }
16 }
```

Figure 16: $\mathcal{P}$: code implementation of $\mathcal{G}$

Next, given the pre-condition $q(Q,1)(x)$ on the loop body, the strongest post-condition computations i.e. *sp*-calculus is performed on the code. In most cases, computing the strongest post-condition is not feasible. However for ellipsoidal invariants, there are transformation rules, which can be exploited to automate this process. Denote the body of the while loop in $\mathcal{P}$ as B, the credible autocoding process computes $sp(B, q(Q,1)(x))$ and then checks that $sp(B, q(Q,1)(x)) \rightarrow q(Q,1)(x)$ to ensure the correctness of $\{q(Q,1)(x)\}\, B\, \{q(Q,1)(x)\}$.

For a piece of code that is linear, as is the case in $\mathcal{P}$, *sp*-calculus use the following result regarding linear transformation of an ellipsoidal set.

Lemma 3.1 Given a set $q(Q,1)(x)$, and given a linear transformation $\mathcal{T}(x) = Tx$ for some matrix T, the image $\mathcal{T}(q(Q,1)(x))$ is the set $q(TQT^{\intercal},1)(x)$ [51].

Using the formula $TQT^{\intercal}$, we can compute a strongest post-condition for every line of code in B. Define C_i as the ith line of code in B. Denote x_i as the state vector after the execution of C_i. For example, the state vector starts with $x = \begin{bmatrix} x1 \\ \\ x2 \end{bmatrix}$ before the execution of C_1. The lines of code C_1 and C_2 respectively assigns some values to the variables $y1$ and $y2$. Hence the state vector increases in dimension and becomes $x_2 = \begin{bmatrix} x1 & x2 & y1 & y2 \end{bmatrix}^{\intercal}$ after the execution of C_2. The state vector is x again after the execution of C_4. because the variables $y1$ and $y2$ are discarded from the state vector when they are not used in the code again. Next, from the state vectors x_{i-1}, x_i, and the affine semantics of C_i, a linear transformation T_i from x_{i-1} to x_i is deduced. For example, C_1 computes the expression $0.4990 * x1 + 0.1 * x2$ and assigns it to the variable $y2$. The affine semantics of C_1 is $y1 := Lx$, in which $L = \begin{bmatrix} 0.4990 & 0.1 \end{bmatrix}$. The state vector x_0 is x and the state vector x_1 is $x_1 = \begin{bmatrix} x1 & x2 & y1 \end{bmatrix}^{\intercal}$. Hence $T_1 = \begin{bmatrix} I \\ \\ L \end{bmatrix}$. Applying lemma 3.1, the strongest post-condition for C_m is

$$q(\prod_{i=m}^{1} T_i Q \prod_{i=1}^{m} T_i^{\intercal}, x_m, 1). \tag{28}$$

By (28), the strongest-post condition of B i.e. $sp(B, q(Q,1)(x))$ is $q(Q_4, x, 1)$, where

$$Q_4 = T_4 T_3 T_2 T_1 Q T_1^{\intercal} T_2^{\intercal} T_3^{\intercal} T_4^{\intercal}. \tag{29}$$

The computed post-conditions are inserted into $\mathcal{P}$ as pseudo-ACSL annotations (see

```
while (1) {
/*@
  requires q(Q,1)(x);
  ensures  q(T1*Q*transpose(T1),1)(x1);
 */
 {
  y1=0.4990*x1+0.1*x2;
 }
/*@
  requires q(Q,1)(x);
  ensures  q(T2*T1*Q*transpose(T1)*transpose(T2),1)(x2);
 */
 {
  y2=0.01*x1+1.0*x2;
 }
/*@
  requires q(Q,1)(x);
  ensures  q(T3*T2*T1*Q*transpose(T1)*transpose(T2)*transpose(T3),1)
     ↪ (x3);
 */
 {
  x1=y1;
 }
/*@
  requires q(Q,1)(x);
  ensures  q(T4*T3*T2*T1*Q*transpose(T1)*transpose(T2)*transpose(T3)
     ↪ *transpose(T4),1)(x);
 */
 {
  x2=y2;
 }
}
```

Figure 17: $\mathcal{P}$ from Figure 16 annotated with pseudo-ACSL

Figure 17). Together with the loop invariant $q(Q,1)(x)$, they form a proof of correctness for $\mathcal{P}$ with respect to the invariance of $q(Q,1)(x)$. To check the inductive invariance of $q(Q,1)(x)$. we still need to verify that $sp(B, q(Q,1)(x)) \rightarrow q(Q,1)(x)$. This can done by verifying either $Q - Q_4 \succ 0$ or $Q_4^{-1} - P \succ 0$ using a Cholesky decomposition algorithm [56].

3.3.1 Abstract Types in ACSL

In Gene-Auto and Simulink, the control semantics are expressed using linear algebra types such as matrix and vector. To express the same expressions in ACSL, a library of linear algebra symbols and axioms in ACSL was defined. The ACSL matrix and vector types are displayed in Figure 18. A matrix $P \in \mathbb{R}^{n \times n}$ in ACSL is defined using a function template `mat_of_nxn_scalar` that takes in n^2 arguments that corresponds to the entries of the matrix. A vector $x \in \mathbb{R}^n$ is defined similarly.

ACSL
```
logic matrix P = mat_of_nxn_scalar(a_1,.....,a_n*n)
logic vector x = vect_of_n_scalar(a_1,....,a_n)
```

Figure 18: matrix and vector types in ACSL

To express the quadratic predicates $p(P,x)(1)$ and $q(Q,1)(x))$, two ACSL functions, displayed in Figure 19, were defined.

ACSL
```
in_ellipsoidP(P,vect_of_n_scalar(...));
in_ellipsoidQ(Q,vect_of_n_scalar(...));
```

Figure 19: Predicate Types in ACSL

To express operations on the matrix and vector types, a set of additional ACSL

functions were defined. Some of them are displayed in Figure 20. Note that the `block` function takes in four matrices A, B, C, D and returns $\begin{bmatrix} A & B \\ C & D \end{bmatrix}$.

ACSL

```
mat_mult(matrix,matrix)
mat_add(matrix,matrix)
block(matrix,matrix,matrix,matrix)
transpose(matrix)
.
.
.
```

Figure 20: matrix and vector arithmetic in ACSL

Chapter IV

TRANSLATION PROTOTYPE

This chapter describes, in more details, the translation process in the credible autocoding prototype. The running examples include a lead/lag compensator system and a plant model for expressing closed-loop stability. From the input model to the verified output, the property of open-loop and closed-loop stability is translated and the formally verified using the backend to the prototype. The example lead/lag compensator system is described by the state-space difference equation in (30). This example is chosen because it has enough complexity to be representative of many controllers used in the industry, and is simple enough such that its output annotations can be displayed in this thesis.

Example 4.0.1 The compensator system consists of states $x \in \mathbb{R}^2$, input $y \in \mathbb{R}$, output $u \in \mathbb{R}$, the state-transition and output functions in (30).

$$\begin{aligned} x_+ &= \begin{bmatrix} 0.4990 & -0.05 \\ 0.01 & 1 \end{bmatrix} x + \begin{bmatrix} 0 \\ 0.01 \end{bmatrix} y \\ u &= \begin{bmatrix} 564.48 & 0 \end{bmatrix} x + \begin{bmatrix} 1280 \end{bmatrix} y. \end{aligned} \tag{30}$$

The plant model used in the closed-loop case is

$$\begin{aligned} x_+ &= \begin{bmatrix} 1.000 & 0.01 \\ -0.01 & 1.000 \end{bmatrix} x + \begin{bmatrix} 0.00005 \\ 0.01 \end{bmatrix} y \\ u &= \begin{bmatrix} 1.0 & 0.0 \end{bmatrix} x + \begin{bmatrix} 0.0 \end{bmatrix} y. \end{aligned} \tag{31}$$

4.1 Constructing the Models for Credible Autocoding

The model annotated with the property of closed-loop stability is displayed in Figure 21. The reference input y_d in Figure 21 is assumed to be bounded. This assump-

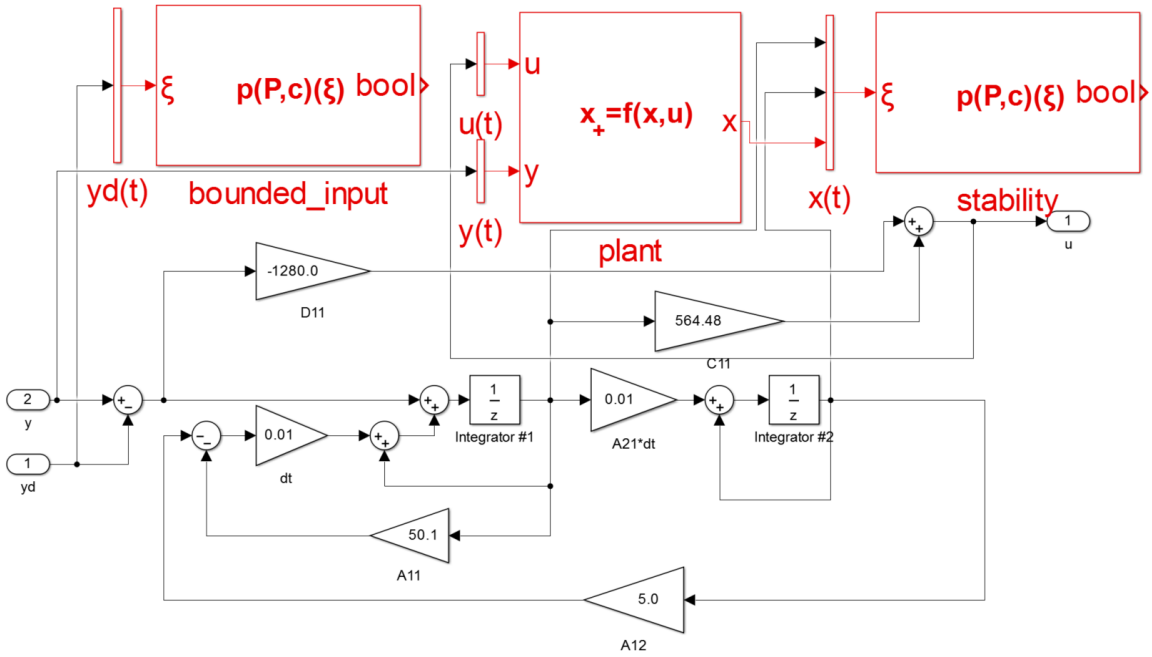


Figure 21: Control system annotated with closed-loop stability

tion is expressed by the *quadratic* block `bounded_input`. A stability analysis for the closed-loop yields a quadratic invariant defined by $P \succ 0$ such that

$$P = \begin{bmatrix} 0.1878 & 0.1258 & -0.0813 & 0.0149 \\ 0.1258 & 0.3757 & -0.0220 & 0.0100 \\ -0.0813 & -0.0220 & 0.0660 & -0.0063 \\ 0.0149 & 0.0100 & -0.0063 & 0.0012 \end{bmatrix} \tag{32}$$

and a multiplier $\alpha = 0.991$. The ellipsoidal set $p(P, 1)(x)$, which encodes the proof of stability, is inserted into the model in Figure 21. It is expressed by the *quadratic* block `stability`. The *system* block `plant` in Figure 21 expresses the dynamics of

the plant model used in the closed-loop analysis. The first input port of the block `plant` accepts the input from the controller to the plant. The second input port of `plant` accepts the output from the plant to the controller. The output of `plant` is the internal state of the plant model.

The model expressing open-loop stability is displayed in Figure 22. In the open-

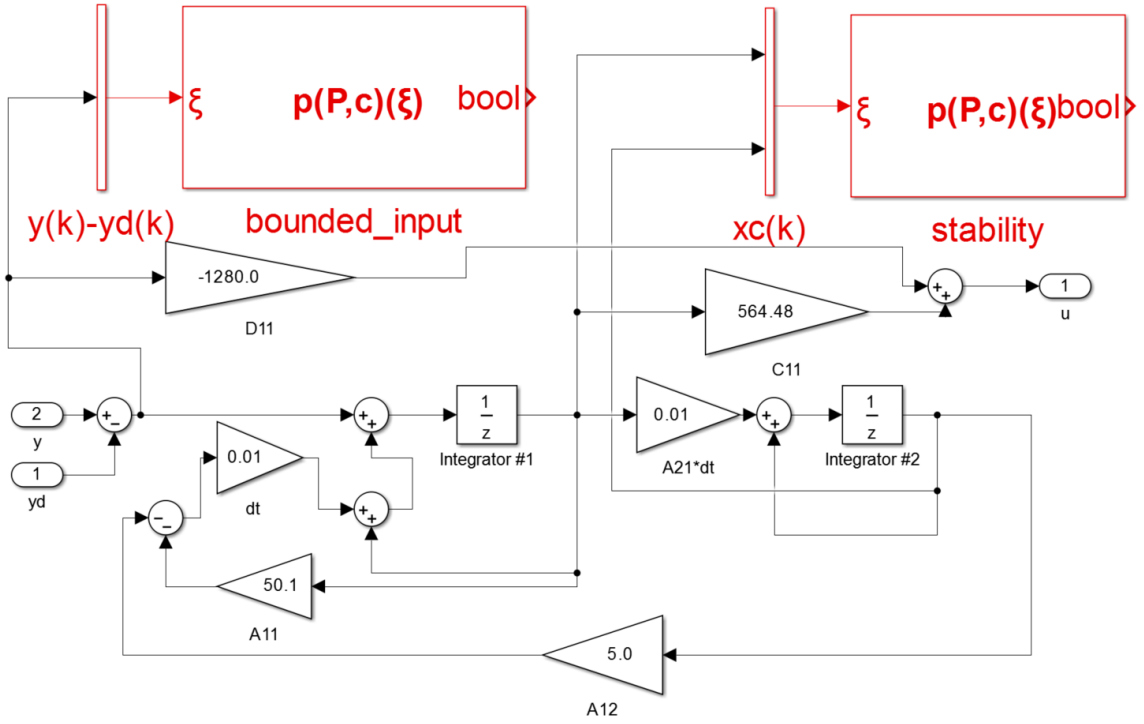


Figure 22: Control system annotated with open-loop stability

loop case, the bounded input assumption is on the signal $y - y_d$. This is expressed by the *quadratic* block `bounded_input` in Figure 22. A stability analysis of the open-loop case produced the quadratic invariant $p(P, 1)(x)$ where

$$P = \begin{bmatrix} 0.0005859 & 4.8246 \times 10^{-5} \\ 4.8246 \times 10^{-5} & 0.002007 \end{bmatrix}, \tag{33}$$

and a multiplier of $\alpha = 0.9991$. This property is expressed by the block `stability` in Figure 22.

4.2 Gene-Auto+: A prototype credible autocoder

This section gives some details on the prototype credible autocoder. The current prototype is capable of automated translation of control semantics, described in Section 3.2, into verifiable ACSL annotations on the code.

4.2.1 Gene-Auto: Translation

Gene-Auto's translation architecture consists of a sequences of independent model transformation stages. This modular design allowed the insertion of additional analysis and verification stages, such as the annotation generation stage in the prototype, without heavy modifications to the rest of the autocoder. The main translation modules within Gene-Auto, are the importer, the block sequencer and typer, the code model generator, and the C printer. We re-use all the translation modules and added additional modules to handle the translation of control semantics.

The Gene-Auto translation process goes through two layers of intermediate languages. The first one, called the *GASystemModel*, is a data-flow language that is similar to Simulink. The input Simulink model, after being imported, is first transformed into the system model. The system model, which is expressed in the *GASystemModel* language, is then transformed into the code model. The code model is expressed in the *GACodeModel* language representation, which is a generic imperative programming language. It shares many similarities with C and Ada. For the translation of the control semantics, we added the sub-module annotations generator, to the code model generator module. The annotations are expressed using *GAVAModel*, the ACSL-like language extension in Gene-Auto+. For more details about

GAVAModel, including its meta-model, please see [1]. The *GAVAModel* language is used to express ASCL statements such as *behavior*, *assumes*-statement, *function contract*, *require*-statement, *ensure*-statement, and *ghost code*, within the code model representation in Gene-Auto+.

Figure 23 summarizes the main differences between the translation process of Gene-Auto and Gene-Auto+. The top half of the figure shows the process in terms of languages and intermediate representations while the bottom half of the figure shows the translation modules. Of the four language representations used in the translation process, only the *GASystemModel* representation remains unchanged. This is because in term of syntax, the annotation blocks are identical to the regular blocks.

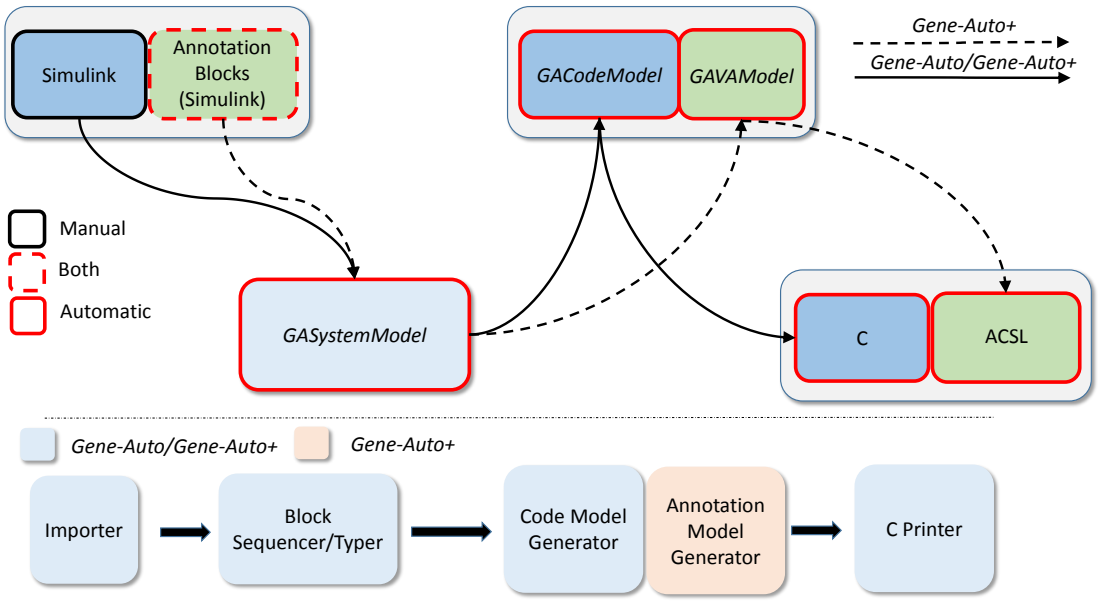


Figure 23: Translation in Gene-Auto+ vs Gene-Auto

For an input controller model, Gene-Auto+ generates two C functions. One is the initialization function and the other is the update function. The initialization function is used to assign initial values to the controller states. This function is typically only

called once before the execution of the controller. The update function is called once per sample period by a loop. It computes the control outputs $u = Cx + Dy$ and updates the controller states $x_+ = Ax + By$. The ASCL annotations described in this chapter are for the update function. By proving the ellipsoidal set obtained from stability analysis is a fix-point of the update function, we also prove that it is an inductive invariant of the loop calling the update function.

4.2.2 Translation of Annotative Blocks

The annotation blocks are also first transformed into a *GASystemModel* representation. This translation stage is not modified from Gene-Auto. In the code model generation stage, the blocks that express the control semantics are skipped since they are categorized as annotations. Instead, they are imported into the annotations generation sub-module. The annotations generation sub-module is initiated after the system model has been translated into the code model. This sub-module translates the annotation blocks into either invariant or ghost code objects, and inserts them into appropriate locations on the code model. Based on these inserted objects, an invariant propagation process is executed on the code model, which generates additional invariants that are also inserted into their appropriate locations on the code model. Finally, all the inserted objects on the code model are translated into *GAVAModel* representations and inserted as annotations on the code model. The code model with the annotations expressed in *GAVAModel* becomes the output of the annotations generation sub-module. This new code model with axiomatic semantics is dubbed the GAVA model.

A high-level overview of all the translations, insertions, and proof generations performed by the annotations generator is displayed in Figure 24. The important

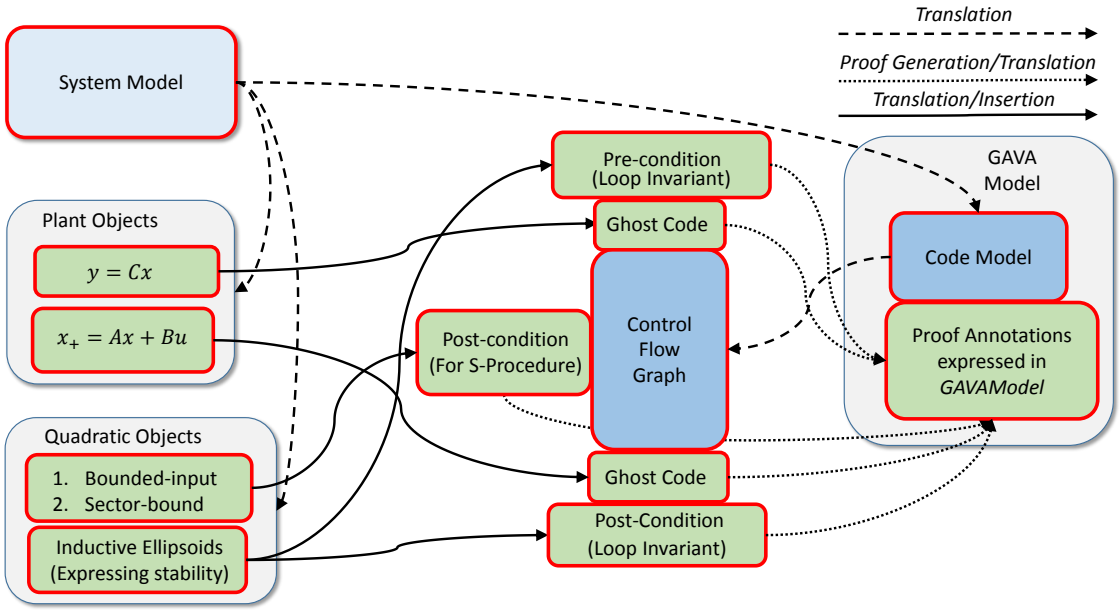


Figure 24: Transformation of control semantics from *GASystemModel* to *GAVAModel*

steps in transforming the annotation blocks into a GAVA model are as follows.

1. The code model is generated as in Gene-Auto.

2. The code model is analyzed and transformed into a control-flow graph structure $\mathcal{C}$.

3. The *constant* blocks are translated into invariant objects and inserted into $\mathcal{C}$.

4. Constant propagation is executed on $\mathcal{C}$ with the definitions provided by the *constant* blocks.

5. The *system* block is translated into two ghost code objects. The first ghost code object corresponds to the output function of $y = Cx$, and is inserted into the

beginning of ⌋. The second ghost code object corresponds to the state-transition function $x_+ = Ax + Bu$, and is inserted into the end of $\mathcal{C}$.

6. The *quadratic* blocks are typed based on their inputs as either inductive, bounded-input, or sector-bound. They are translated into quadratic invariants and inserted into appropriate locations on $\mathcal{C}$.

7. Ellipsoid propagation is executed i.e. either *sp*-calculus or *wp*-calculus. During this process, quadratic invariants are generated for nearly every line of code. and then inserted into $\mathcal{C}$.

8. The resulting collection of invariants and ghost code objects from $\mathcal{C}$ are translated into annotations expressed in *GAVAModel*, and inserted into their corresponding locations in the code model.

4.3 Translation and insertion of the system block

The system block, which represents the model of the plant, is split into two ghost code objects representing respectively the output function $y = Cx$ and the state-transition function $x_+ = Ax + Bu$. Each ghost code object contains a collection of code templates and an affine transformation. The affine transformations are used in the invariant propagation process to be described later. The code templates, parameterized by the state-space matrices, are used to generate C code representation of the state-space system of the plant. They are instantiated with the data $\{A, B, C\}$ from the *system* block and become ghost code statements expressed in *GAVAModel*. The *GAVAModel* ghost code statements are printed as ACSL ghost code statements

```
/*@
        ghost REAL Plant_0_1[2];
*/
/*@
        ghost REAL Plant_xp_0_tmp;
*/
/*@
        ghost REAL Plant_xp_1_tmp;
*/

/*@
        requires \valid(_io_) && \valid(_state_);
        requires _io_->y == 1.0 * Plant_0_1[0];
  .
  .
*/
void cl_result_compute(t_cl_result_io *_io_, t_cl_result_state *
   ↪ _state_) {
    REAL A11;
    REAL A12;
    .
    .

/*@
    .
  .
  requires in_ellipsoidQ(QMat_31,vect_of_6_scalar(
     ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
     ↪ Plant_0_1[0],Plant_0_1[1],_io_->u,Sum5));
  ensures in_ellipsoidQ(QMat_32,vect_of_4_scalar(
     ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
     ↪ Plant_0_1[0],Plant_0_1[1]));
  .
  .
*/
 {
    /*@
            ghost Plant_xp_0_tmp = Plant_0_1[0];
    */
    /*@
            ghost Plant_xp_1_tmp = Plant_0_1[1];
    */
    /*@
            ghost Plant_0_1[0] = 1.0 * Plant_xp_0_tmp + 0.01 *
               ↪ Plant_xp_1_tmp + 5.0E-5 * _io_->u;
    */
    /*@
            ghost Plant_0_1[1] = -0.01 * Plant_xp_0_tmp + 1.0 *
               ↪ Plant_xp_1_tmp + 0.01 * _io_->u;
    */

 }
}
```

Figure 25: Ghost code representation of the plant dynamics

during the C printer stage. The set of ACSL ghost code statements, generated from the closed-loop example, is displayed in Figure 25. The pre-condition `\valid(_io_) && \valid(_state_);` in line 12 requires that memories are allocated correctly for the pointers `_io_` and `_state_`. The pre-condition `_io_->y == 1.0 * Plant_0_1[0]` in line 13 establishes the equivalence between the output $y = Cx$ of the plant and the input y to the controller. This pre-condition is used to establish one half of the feedback interconnection between the plant model and the controller program. The state-transition function of the plant is expressed by the block of ghost code statements in lines 33 to 42. The output variable from the controller program `_io_->u` is the input to the plant. This establishes the other half of the feedback interconnection. Although ghost code statements are not executable i.e. it does not change the semantics of the program, it can be used to change the semantics expressed in the annotations. For example, there is a ACSL contract in lines 26 and 27 of Figure 25 for the block of ghost code from lines 33 to 42.

4.4 Translation of the quadratic blocks

A short description of the typing of the quadratic blocks and their translations is given here. The semantics of stability are structured in such way that there is one inductive quadratic invariant obtained from the stability analysis and one or more assertive quadratic invariants which are assumptions or properties of the inputs. The assertive quadratic blocks can express either a simple bounded-input type condition or a more complex sector-bound type condition.

4.4.1 Typing of Quadratic Blocks

The quadratic blocks are separated into two groups. The first group contains the inductive blocks that encode the stability property of the system. To compute if a quadratic block is inductive, the following conditions are checked.

1. Every input port of the quadratic block is connected to either an input port of an unit delay block or to an output port of a system block.

2. Let set $\mathcal{U}$ be the set of all unit delay blocks connected to the quadratic block. For any unit delay blocks in $\mathcal{U}$, there must exist a path from its output node to its input node on the system model.

The second group are the assertive blocks. These blocks encode certain assumptions or properties on inputs. . Any quadratic blocks with one or more input connected to blocks other than the unit delay block or the system block is categorized as an assertive block. The assertive quadratic blocks are further grouped into either a sector-bound type or a bounded-input type. The sector-bound type blocks are determined by checking that its level-set parameter c is set to 0 and its inputs are connected to outputs of saturation functions. The prototype autocoder assumes any saturation function on the Simulink model is implemented with a pair of min and max blocks.

Next, the inductive blocks and the bounded-input type blocks are translated into ellipsoidal invariants in the Schur-form. For example, if an inductive block expresses the quadratic predicate $p(P, x)(1),\ P \succ 0$, then it is translated into the quadratic invariant $q(Q, 1)(x)$ where $Q = P^{-1}$. This translation step is necessary as the subsequent ellipsoid propagation process can produce degenerate ellipsoids where Q in

$q(Q,1)(x)$ is singular. The sector-bound type blocks are not translated into the Schur-form since they do not express ellipsoidal sets.

4.4.2 Insertion of Quadratic Invariants

An assertive quadratic invariant is inserted as a post-condition of a node on the control flow graph. Consider a bounded-input quadratic invariant $q(Q_b,1)(x)$ where x is a vector of variable(s). The location of the node is determined using x.

1. Find all assignment statements $x(i) := \ldots$ for variables $x(i)$ in x.

2. Choose the assignment statement that is executed last as the location of insertion.

If x contains variables that are not arguments of the update function, then the prototype will try to compute a weakest pre-condition $q(Q_b,1)(x') = wp(C, q(Q_b,1)(x))$ where x' contains affine expressions of the arguments of the update function. For example, consider the quadratic block `bounded_input` connected to the signal $y - y_d$ in the open-loop case. The `bounded_input` block is translated into the invariant $q(Q_{16},1)(x)$ where $x = [$`Sum4`$]$, which gets inserted as the post-condition of `Sum4=ol_result_y - ol_result_yd` (see line 35 of Figure 26). Since the variable `Sum4` is not an argument of the update function, the autocoder initiates wp-calculus which results in the weakest pre-condition $q(Q_0,1)(x')$, $Q_0 = Q_{16}$ in line 3 of Figure 26 where $x' = [$`_io_->y - _io_->yd`$]$.

The insertion of an inductive ellipsoid is straightforward. The ellipsoid is duplicated three times and inserted as pre and post-condition respectively at the beginning and end of the update function body. The remaining duplicates are inserted as a pre-

```
/*@
  .
  requires in_ellipsoidQ(QMat_0,vect_of_1_scalar(_io_->y - _io_->yd)
     ↪ );
  .
*/
void simple_olg_compute(t_simple_olg_io *_io_, t_simple_olg_state *
   ↪ _state_) \{
.
.
.
/*@
  .
  requires in_ellipsoidQ(QMat_11,vect_of_1_scalar(_io_->y - _io_->yd
     ↪ ));
  ensures in_ellipsoidQ(QMat_12,vect_of_1_scalar(ol_result_y -
     ↪ _io_->yd));
  @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
 */
 {
    ol_result_y = _io_->y;
 }

/*@
  .
  requires in_ellipsoidQ(QMat_12,vect_of_1_scalar(ol_result_y -
     ↪ _io_->yd));
  ensures in_ellipsoidQ(QMat_14,vect_of_1_scalar(ol_result_y -
     ↪ ol_result_yd));
  @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
  .
 */
 {
    ol_result_yd = _io_->yd;
 }

/*@
  .
  .
  requires in_ellipsoidQ(QMat_14,vect_of_1_scalar(ol_result_y -
     ↪ ol_result_yd));
  ensures in_ellipsoidQ(QMat_16,vect_of_1_scalar(Sum4));
  @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
  .
*/
 {
    Sum4 = ol_result_y - ol_result_yd;
 }
.
.
}
```

Figure 26: *wp*-calculus on quadratic invariants expressed in ACSL

and post-condition on the update function itself. These ellipsoids $q(Q_1, 1)(x)$ are defined by the matrix variable `QMat_1` in Figure 27. After the inductive ellipsoids are inserted, the prototype autocoder generates a line by line proof, showing that $q(Q_1, 1)(x)$ is a fix-point of the update function. The line by line proof is produced automatically by using a set of strategies for transforming quadratic sets. The strategies, based on ellipsoidal calculus, enables the prototype to perform *sp*-calculus on the linear and nonlinear portions of the code. Next, we describe in some details about these strategies.

4.5 Computing Post-conditions

In Gene-Auto+, *sp*-calculus for ellipsoidal invariants has been automated using a set of transformation algorithms for ellipsoids. This set of transformation strategies can be divided into two categories: affine transformations, and S-procedure transformations. The affine transformations compute the strongest ellipsoidal post-condition for code that are linear transformations on the state of the program. The S-procedure strategies compute over-approximations of the strongest post-condition for the nonlinear parts of the code.

4.5.1 Affine Transformation

The affine transformation has been described briefly in Section 3.3. For automating the proof-checking of the affine transformations of ellipsoids, we define a proof tactic denoted *AffineEllipsoid*, which corresponds to a proof strategy of the same name defined in PVS [40]. This rule is applied whenever a linear abstraction of the code can be computed. Recall from Section 3.3, given the pre-condition $q(Q, 1)(x)$ and a

```
/*@
  logic matrix QMat_1 = mat_of_2x2_scalar(1710.0449662492558,-41
     ↪ .101885746811455,-41.101885746811455,499.17657993449376);
*/
.
.
.
/*@
  requires in_ellipsoidQ(QMat_1,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));
 */
void simple_olg_compute(t_simple_olg_io *_io_, t_simple_olg_state *
   ↪ _state_) {
/*@
  .
  .
  requires in_ellipsoidQ(QMat_1,vect_of_2_scalar(
     ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory));
  .
  .
 */
 {
   Integrator_1 = _state_->Integrator_1_memory;
 }
 .
 .
 .
/*@
  .
  .
  ensures in_ellipsoidQ(QMat_1,vect_of_2_scalar(
     ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory));
  .
  .
 */
 {

 }

}
```

Figure 27: Inductive ellipsoids in ACSL

line of code $\hat{z} := Lx$, then the linear transformation from x to $x_+ = \begin{bmatrix} x^\intercal & \hat{z} \end{bmatrix}^\intercal$ is $T = \begin{bmatrix} I & L^\intercal \end{bmatrix}^\intercal$. Hence the strongest post-condition is $q(Q_+, 1)(x_+)$ where $Q_+ = TQT^\intercal$.

In the more general case, let $\hat{z} := Ly$, where y is a vector of m program variables, and $L \in \mathbb{R}^{1\times m}$. Let $\mathcal{Q}_i(x) := q(Q_i, 1)(x)$ where x is a vector of n program variables. Let z denote a vector containing only the variable $\hat{z}$. Let $x \cup z$ denote the union of the variables from x and z i.e. $x \cup z = \begin{bmatrix} x(1) & \dots & x(n) & \hat{z} \end{bmatrix}^\intercal$ if $\hat{z} \notin x$. Note that if $\hat{z} \in x$ then $x \cup z = x$. Let the function $\mathcal{F}$ be

$$\mathcal{F} : (Q_n, \psi(L, y, x), \phi(\hat{z}, x)) \to \mathcal{T}\left(\psi(L, y, x), \phi(\hat{z}, x)\right) Q_n \mathcal{T}^\intercal \left(\psi(L, y, x), \phi(\hat{z}, x)\right), \quad (34)$$

where

$$\mathcal{T}\left(\psi(L, y, x), \phi(\hat{z}, x)\right)(i, j) = \begin{cases} 1, & 0 \le i, j < n \wedge i = j \wedge i \neq \phi(\hat{z}, x) \\ 0, & 0 \le i, j < n \wedge i \neq j \wedge i \neq \phi(\hat{z}, x) \\ \psi(L, y, x)(j), & i = \phi(\hat{z}, x) \wedge 0 \le j < n \end{cases}$$

$$\psi(L, y, x)(j) = \begin{cases} L(0, k), & 0 \le j < n \wedge 0 \le k < m \wedge x(j) = y(k) \\ 0, & 0 \le j < n \wedge 0 \le k < m \wedge x(j) \neq y(k) \end{cases}$$

$$\phi(\hat{z}, x) = \begin{cases} i, & \hat{z} \in x \wedge \hat{z} = x_i \\ n, & \hat{z} \notin x \end{cases} \quad (35)$$

The *AffineEllipsoid* strategy is

$$\overline{\{\mathcal{Q}_n(x)\}\, \hat{z} := Ly\, \{\mathcal{Q}_{n+1}(x \cup z)\}}, Q_{n+1} = \mathcal{F}\left(Q_n, \psi(L, y, x), \phi(\hat{z}, x)\right) . \quad (36)$$

The function $\mathcal{T}$ computes the linear transformation matrix T such that $Q_{n+1} = TQ_nT^\intercal$. To clarify (35), we provide a simple example. Let $x = \begin{bmatrix} x_1 & x_2 & x_3 & x_4 \end{bmatrix}^\intercal$.

Consider the line of code $x_5 := 2x_1 + 3x_4$. We have $\hat{z} = x_5$, $L = \begin{bmatrix} 2 & 3 \end{bmatrix}$ and $y = \begin{bmatrix} x_1 & x_4 \end{bmatrix}^\intercal$. Since $\hat{z} \notin x$ hence $\phi(\hat{z}, x) = 4$. According to the definition of $\mathcal{T}$ in (35), for $i \neq \phi = 4$, $\mathcal{T}$ returns a identity matrix. We have the first 4 rows of the T being a identity matrix and the last row being unknown.

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ ? & ? & ? & ? \end{bmatrix} \tag{37}$$

The function $\psi\,(L, x, y)$ is used to fill in the last row of (37). For example, for the fourth entry in the last row or at index $j = 3$, we have $x(3) = x_4$ which is the same as the variable located at index $k = 1$ in y. Hence we have $x(3) = y(1)$ which means $\psi\,(L, x, y)\,(0)$ returns $L(0, 1)$ which is 3. This is repeated for every column index $j = 0, \ldots, 3$ and we get the complete transformation matrix

$$T = \begin{bmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 2 & 0 & 0 & 3 \end{bmatrix}. \tag{38}$$

The *ReduceEllipsoid* strategy is also an affine transformation strategy. Given an ellipsoid $\mathcal{Q}_n(x)$ and x is a vector of program variables of dimension n. We want to compute an ellipsoid $\mathcal{Q}_{n+1}(x')$ where $x' = x \setminus z$ i.e. x' is a vector of variables obtained

from removing the variable $\hat{z}$ from x. This reduction in the dimension of the ellipsoid invariant is necessary to generate a final post-condition that is not degenerate. Let the function $\mathcal{G}$ be

$$\mathcal{G} : (Q_n, \theta(\hat{z}, x)) \rightarrow \mathcal{T}\left(\theta(\hat{z}, x)\right) Q_n \mathcal{T}\left(\theta(\hat{z}, x)\right)^{\mathsf{T}}, \tag{39}$$

where

$$\mathcal{T}\left(\theta(\hat{z}, x)_{i,j}\right) := \begin{cases} 1, & 0 \leq i, j \leq n-1 \wedge ((i < \theta(\hat{z}, x) \wedge i = j) \vee (i \geq \theta(\hat{z}, x) \wedge j = i+1)) \\ 0, & 0 \leq i, j \leq n-1 \wedge ((i < \theta(\hat{z}, x) \wedge i \neq j) \vee (i \geq \theta(\hat{z}, x) \wedge j \neq i+1)) \end{cases}$$

$$\theta(\hat{z}, x) := \begin{cases} i, & \hat{z} = x(i) \\ error, & \hat{z} \notin x \end{cases} \tag{40}$$

The *ReduceEllipsoid* strategy is

$$\frac{}{\{\mathcal{Q}_n(x)\}\ \mathbf{SKIP}\ \{\mathcal{Q}_{n+1}(\mathrm{x} \setminus \mathrm{z})\}}, Q_{n+1} = \mathcal{G}\left(Q_n, \theta(\hat{z}, x)\right)\ . \tag{41}$$

The reduction rule is applied whenever a variable in x is no longer used in further program execution. The function $\mathcal{T}$ in the *ReduceEllipsoid* strategy is equivalent to a function that deletes a row from a identity matrix $I_{n \times n}$. The row deleted is $I(i)$ where $i = \theta(\hat{z}, x)$ is the index location of $\hat{z}$ in x.

The control flow graph as well as any ghost code objects are analyzed for their affine semantics. For each line of code that is linear, a matrix L is computed and stored in the control flow graph. For example, for the line of code `x = y+ 2*x`, the affine analysis algorithm returns the matrix $L = \begin{bmatrix} 1 & 2 \end{bmatrix}$. For the ghost code objects, their affine semantics are computed by instantiating the existing templates from the system block.

Figure 28 shows an example of the *AffineEllipsoid* usage in the closed-loop case. In this example, the pre-condition is the ellipsoid defined by the matrix variable

```
1  /*@
2    logic matrix QMat_28 = mat_mult(mat_mult(mat_of_10x10_scalar(1.0,0
        ↪ .0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0
        ↪ .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
        ↪ .0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0
        ↪ .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
        ↪ .0,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0
        ↪ .0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,0.0,1.0),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
        ↪ ,1.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0
        ↪ ,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
        ↪ ,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,1.0,0.0,0.0,0.0,0.0,1
        ↪ .0)));
3  */
4  .
5  .
6  .
7  /*@
8    requires \separated(_io_,_state_);
9    ensures \separated(_io_,_state_);
10
11   behavior Plant_17:
12   requires in_ellipsoidQ(QMat_27,vect_of_10_scalar(
        ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
        ↪ Plant_0_1[0],Plant_0_1[1]
        ↪ ,Integrator_1,Sum3,Sum4,_io_->u,Sum1,dt_));
13   ensures in_ellipsoidQ(QMat_28,vect_of_10_scalar(
        ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
        ↪ Plant_0_1[0],Plant_0_1[1]
        ↪ ,Integrator_1,Sum3,Sum4,_io_->u,dt_,Sum2));
14   @ PROOF_TACTIC (use_strategy (AffineEllipsoid));
15  */
16  {
17     Sum2 = dt_ + Integrator_1;
18  }
```

Figure 28: Application of the *AffineEllipsoid* strategy

`QMat_27`, and the line of code assigns the expression `dt_+ Integrator_1` to the variable `Sum2`. The affine semantics of the code is $\hat{z} := Ly$, where $\hat{z}$ = `Sum2`,

$y = \begin{bmatrix} \texttt{dt\_ Integrator\_1} \end{bmatrix}^{\mathsf{T}}$ and $L = \begin{bmatrix} 1 & 1 \end{bmatrix}$. Applying the *AffineEllipsoid* rule in (35), we get the ellipsoid transformation matrix T defined by the ACSL function `mat_of_10x10_scalar` in line 2.

4.5.2 S-procedure

There are two strategies in Gene-Auto+ for computing over-approximation of the strongest post-condition for the nonlinear parts of the code. The first strategy handles simple bounded inputs. The second strategy handles any nonlinearity in the code in which a sector-bound inequality can be used to over-approximate the semantics of the nonlinearity. Both strategies are based on the S-procedure relaxation technique described first in lemma 1.5 and then applied in the stability analysis of the open-loop and close-loop cases. We first consider the strategy for bounded inputs.

4.5.2.1 Bounded Inputs

In the stability analysis of both the closed-loop and open-loop cases, the ellipsoid invariants were computed along with a positive multipliers $\alpha > 0$. This relaxation multiplier $\alpha > 0$ is used in *sp*-calculus when we have two pre-conditions $q(Q, 1)(x)$ and $q(Q_b, 1)(x_i)$ where $x_i \cap x = \varnothing$, and a line of code $\hat{z} := Ly$ where some variables in y belongs to x and others belong to x_i. Let the bounded-input ellipsoid be $q(Q_b, 1)(x_i)$. Let the inductive ellipsoid be $q(Q, 1)(x)$. To ensure further ellipsoid propagation, a strategy is used to combine $q(Q, 1)(x)$ and $q(Q_b, 1)(x_i)$ into a single ellipsoidal post-condition $q(Q_+, 1)(x \cup x_i)$ where

$$Q_+ = \begin{bmatrix} \frac{1+\alpha}{1} Q & 0 \\ 0 & \frac{1+\alpha}{\alpha} Q_b \end{bmatrix}. \tag{42}$$

The multiplier $\alpha > 0$ computed with the S-procedure technique is used in this strategy. This ensures the post-condition remains an inductive ellipsoid invariant.

For the general case of combining an inductive ellipsoid invariant $\mathcal{Q}_0(x_0)$ with bounded-input ellipsoids $\mathcal{Q}_i(x_i)$, $i = 1, \ldots, m$ each with a multiplier of α_i, we have the *SProcedure* strategy as follows. Given $\dim : R^{n \times n} \rightarrow n$, $\rho(n) = \sum_{i=0}^{n} \dim(Q_i)$, and

$$\mathcal{H}(Q_i)(s,t) = \begin{cases} Q_i(s - \rho(i-1), t - \rho(i-1)), & \rho(i-1) \leq s, t \leq \rho(i) \\ 0.0, & \text{otherwise} \end{cases}, \quad (43)$$

the *SProcedure* strategy is

$$\frac{}{\{\mathcal{Q}_0(x_0) \wedge \mathcal{Q}_1(x_1) \wedge \ldots \wedge \mathcal{Q}_m(x_m)\}\ \mathbf{SKIP}\ \{\mathcal{Q}_+(\mathrm{x}_0 \cup \mathrm{x}_1 \cup \ldots \cup \mathrm{x}_n)\}}$$
$$Q_+ = \sum_{i=0}^{N} \frac{\mu}{\alpha_i} \mathcal{H}(Q_i), \quad (44)$$

where $\alpha_0 = 1$ and $\mu = \sum_{j=0}^{m} \alpha_j$. Note the function $\mathcal{H}$ returns the block matrices in Q_+

Given a set of ellipsoidal pre-conditions $\{\mathcal{Q}_i(x_i)\}$ and a line of code $\hat{z} := Ly$, the *SProcedure* strategy is activated only when all the following conditions are satisfied.

1. For the set $\{\mathcal{Q}_i(x_i)\}, i = 0 \ldots m$, $y \subseteq \bigcup_{i=1}^{m} x_i$.
2. For $\mathcal{Q}_i(x_i), i = 0, \ldots, m$, $y \nsubseteq x_i \wedge y \cap x_i \neq \varnothing$.
3. For $0 \leq i, j \leq m$, $x_i \cap x_j = \varnothing$ for $i \neq j$.

An example usage of the *SProcedure* strategy by the prototype autocoder is displayed Figure 29, This code is generated by Gene-Auto+ from the open-loop case. The ellipsoidal pre-condition defined by the matrix variable `QMat_18` reflects the bounded-input condition. The other ellipsoidal pre-condition, defined by the matrix variable

```
/*@
  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));
*/
.
.
.
/*@
  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 EllipsoidMain_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;
 }
```

Figure 29: Application of the *SProcedure* strategy

`QMat_19` is the inductive invariant. These two ellipsoids are combined to form the post-condition ellipsoid using the *SProcedure* proof strategy. The definition of the matrix variable `QMat_20`, which defines the post-condition ellipsoid, is displayed in line 2 of Figure 29. The definition is expressed using the ACSL block matrix function `block_m` and the matrix scaling function `mat_scalar_mult`.

4.5.2.2 Sector-bound Condition

The second S-procedure ellipsoid combination rule is based on the relaxation of the sector bound condition

$$(\Delta(y) - m_1 y)\,(\Delta(y) - m_2 y) \leq 0 \tag{45}$$

in the Lyapunov stability analysis. The sector-bound block expresses the quadratic invariant $p(H, 1)(x_i)$ where

$$H = \begin{bmatrix} m_1 m_2 & -\frac{1}{2}(m_1 + m_2) \\ \frac{1}{2}(m_1 + m_2) & 1 \end{bmatrix}, \quad x_s = \begin{bmatrix} y \\ \Delta(y) \end{bmatrix}. \tag{46}$$

with a multiplier of $\beta > 0$. The sector-bound strategy is applied to ensure ellipsoid propagation when the *AffineEllipsoid* strategy could not be applied on the inductive invariant $q(Q, 1)(x)$. More specifically, the *SectorBound* strategy is executed when the following conditions are satisfied.

1. For the pre-conditions $q(Q, 1)(x)$ and $p(H, 0)(x_s)$, $x \nsubseteq x_s$ and $x_s \nsubseteq x$.

2. The affine semantics of the code is $\hat{z} := Ls$ where $s \nsubseteq x$ and $s \nsubseteq x_s$.

3. The variable y in x_s is equal to a linear combination of the variables in x.

We can assume there exists a transformation matrix $C \in \mathbb{R}^{1\times n}$ such that $y = Cx$ since this is one of the conditions to be satisfied before the strategy is executed. The *SectorBound* strategy is

$$\overline{\{\mathcal{Q}(x) \wedge p(H,0)(x_s)\}\,\mathbf{SKIP}\,\{\mathcal{Q}_+(\mathrm{x} \cup \mathrm{x_s})\}}$$

$$Q_+ = \begin{bmatrix} Q & 0 \\ 0 & 0 \end{bmatrix} + \begin{bmatrix} \frac{QC^\intercal}{CQC^\intercal} & 0 \\ 0 & 1 \end{bmatrix} \left(\left(\left(\begin{bmatrix} (CQC^\intercal)^{-1} & 0 \\ 0 & 0 \end{bmatrix} + \beta H \right)^{-1} - I \right) \begin{bmatrix} \frac{QC^\intercal}{CQC^\intercal} & 0 \\ 0 & 1 \end{bmatrix}^\intercal \right. . \tag{47}$$

The ellipsoid combination rule in 47 has been proved in the PVS theorem prover and can be checked automatically using the prototype backend.

4.5.3 Verifying the Inductive Condition

The final output of *sp*-calculus is an alternative post-condition for the update function. To show that the ellipsoid invariant obtained from the stability analysis is inductive, we only need to check if the alternative post-condition is contained within it. This inductive condition is normally expected to hold unless mistakes are introduced into the model or there are bugs in the translation. Once the inductive condition is verified, credible autocoding terminates with a positive result. In another words, if this inductive condition holds, then we can claim the generated code satisfies the property of stability.

To communicate this final proof step to the backend, an additional ACSL contract is generated with the proof strategy *PosDef*. For the closed-loop example, this contract is displayed in lines 15 and 16 of Figure 30. The pre-condition in the contract is

$q(Q_{32}, 1)(x)$ and the post-condition is precisely $q(Q_1, 1)(x)$. This last ACSL contract simply tells the backend to verify the inductive condition $q(Q_{32}, 1)(x) \rightarrow q(Q_1, 1)(x)$ by verifying if $Q_1 - Q_{32}$ is positive-definite. For the closed-loop example, because of a bug in the original Gene-Auto, which causes the sign of a gain parameter to be flipped during code generation, the inductive condition could not be discharged until the bug was fixed.

```
1  /*@
2    .
3    requires in_ellipsoidQ(QMat_1,vect_of_4_scalar(
        ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
        ↪ Plant_0_1[0],Plant_0_1[1]));
4    requires in_ellipsoidQ(QMat_2,vect_of_1_scalar(_io_->yd));
5    ensures in_ellipsoidQ(QMat_1,vect_of_4_scalar(
        ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
        ↪ Plant_0_1[0],Plant_0_1[1]));
6    .
7  */
8  void cl_result_compute(t_cl_result_io *_io_, t_cl_result_state *
      ↪ _state_) {
9      REAL A11;
10     REAL A12;
11     .
12     .
13     .
14 /*@
15   requires in_ellipsoidQ(QMat_32,vect_of_4_scalar(
        ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
        ↪ Plant_0_1[0],Plant_0_1[1]));
16   ensures in_ellipsoidQ(QMat_1,vect_of_4_scalar(
        ↪ _state_->Integrator_1_memory,_state_->Integrator_2_memory,
        ↪ Plant_0_1[0],Plant_0_1[1]));
17   @ PROOF_TACTIC (use_strategy (PosDef));
18  */
19  {
20
21  }
22 }
```

Figure 30: Verifying the inductive condition

Chapter V

FLOATING-POINT COMPUTATION ISSUES IN CREDIBLE AUTOCODING

5.1 Introduction

In computing, the manipulation of real numbers is carried out with a finite-precision approximation. This is due to both practical reasons (finite memory and power) and a theoretical bound on the quantity of information that can be stored within a bounded volume of the universe [9]. A type of finite-precision representation, floating-point number, has seen wide adoption in computing and is increasingly used in many safety critical embedded applications [62]. It was noted in [36] that floating-point computations can produce unpredictable and possibly large errors. In this chapter, we present a refinement of the credible autocoding process, which makes it sound with regard to floating-point computation errors.

5.1.1 Reasons not to Ignore Floating-point Computation Errors

Floating-point computation errors can affect systems in serious ways. Some high-profile accidents caused by floating-point computation errors include the Ariane 5 explosion [67] and the Patriot missile overshoot incident [86]. These costly accidents highlight a reason why a rigorous treatment of floating-point computation errors is needed. But even more importantly, since credible autocoding is about providing rigorous proofs on the code, we must make sure the process itself is correct. Up to

now we have ignored the semantic gap that exists between floats and reals. Without closing this gap, we cannot claim credible autocoding is sound.

5.1.2 A Robust Control Approach?

From the perspective of control theorist, a framework that immediately comes to mind, which can be applied towards analyzing control systems with floating-point computational errors, is robust control [49]. If one can model the floating-point errors in the system using multiplicative uncertainties, then we can analyze the stability of the system using the μ-analysis approach [32]. However, the problem with any robust control analysis is that, there are no guarantees of bounds on the errors produced by the floating-point computations performed in the analysis itself. In the context of credible autocoding, without bounding those errors, we cannot claim the robust control argument is sound.

5.2 Floating-point Numbers

A prototypical binary floating-point number [44] is encoded by three binary integers s, τ and m. The first integer s is a sign bit. The two other integers τ and m encode the exponent and mantissa respectively. Let M be the bitwise size of the mantissa, and m_i be the ith bit of the mantissa, then the value of the float $s\tau m$ is precisely $(-1)^s \left(1 + \sum_{i=1}^{M} m_{i-1} 2^{-i}\right) \times 2^{\tau}$.

Example 5.2.1 Let $M = 3$, $s = 0$, $\tau = 10$, and $m = 100$. The binary floating-point number 010100 in decimal is exactly 6.

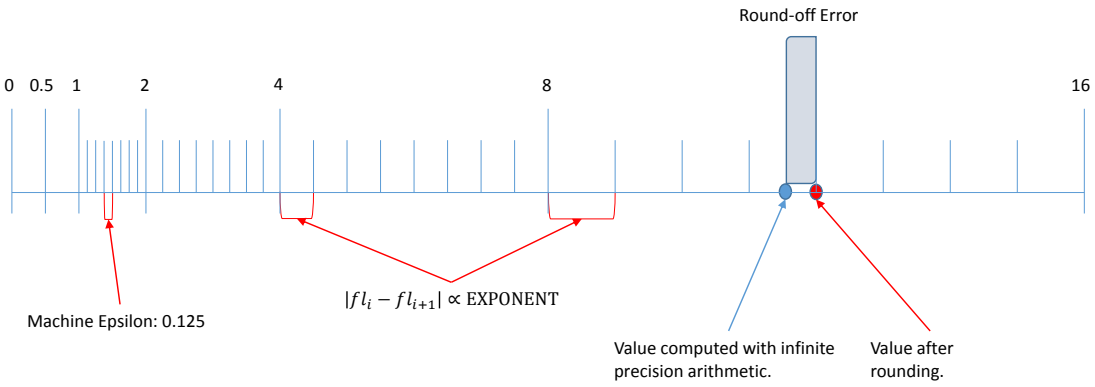


Figure 31: A example floating-point number system with 3-bit mantissa and exponent

A set of floating-point numbers with the exponent τ can be visualized as a set of 2^M equally spaced points on the real interval $\left[2^\tau, 2^{\tau+1}\right)$. The distance between any two adjacent floating-point numbers, both with exponent τ, is $\frac{2^\tau}{2^M}$. The example floating-point number system in Figure 31 has a 3-bit mantissa and a 3-bit signed exponent. For this example, spacing between consecutive floating-point numbers with an exponent of 2 is 0.5. This distance increases to 1 for $\tau = 3$. Note that most of the floats between 0 and 1 in Figure 31 are not displayed since the spacings between them are too small for them to be displayed clearly. The semantics of floating-point arithmetic operators are defined using a rounding function that maps a real number to its adjacent floating-point numbers. Let $\mathcal{F}^\diamond : \mathbb{R}^n \to \mathbb{F}^n$ denotes a rounding function with a rounding mode $\diamond$. The rounding modes $\diamond \in \{\uparrow, \downarrow, 0, \epsilon\}$ corresponds to the directions: towards $+\infty$, towards $-\infty$, towards zero, and towards the nearest [36]. A correct implementation of the IEEE 754 standard implies the floating-point arithmetic operators $\{\oplus, \ominus, \otimes, \oslash\}$ return a value as if it was computed using infinite precision arithmetic and then rounded to the nearest floating-point number. We have

$$
\begin{aligned}
a \oplus b &= \mathcal{F}^{\epsilon}\left(a + b\right), \\
a \ominus b &= \mathcal{F}^{\epsilon}\left(a - b\right), \\
a \otimes b &= \mathcal{F}^{\epsilon}\left(a \times b\right) \\
a \oslash b &= \mathcal{F}^{\epsilon}\left(a \,/\, b\right), b \neq 0.
\end{aligned}
\tag{48}
$$

The product or sum of two floats is unlikely to be a float, which leads to the frequent round-off errors in floating-point computations. As indicated in Figure 31, the round-off error per operation is bounded by the distance between the two floating-point numbers that are closest to the result computed with real arithmetic. This distance is $\frac{2^x}{2^M}$, and when normalized by the exponent 2^x, we get a bound of $\frac{1}{2^M}$ on the relative round-off error. This important quantity, denoted by the symbol υ, is the machine epsilon. For the example in Figure 31, the 3-bit mantissa means a machine of epsilon of $\frac{1}{2^3}$ or 0.125. Floating-point operators are commutative but not associative i.e. $a \oplus (b \oplus c)$ does not have to equal to $(a \oplus b) \oplus c$. The final accumulated error for a set of operations depends on the order of the operations. For the product operator, there is also a possibility of an underflow error, that is, rounding error incurred when the magnitude of the computed result is smaller than the smallest positive floating-point number. The underflow error is bounded by the smallest positive floating-point number, which is denoted using the symbol η. In Figure 31, the smallest representable floating-point number is $\eta = 2^{-4}\frac{1}{2^3}$ or 2^{-7}.

An overflow error can also occur when the result of a computation exceeds the range of the numbers representable by the floating-point number system. In this chapter, we assume no overflows. To summarize, we have the following properties on

errors produced by floating-point addition and multiplication [87]. Given $a, b \in \mathbb{F}$,

$$\begin{aligned} a \oplus b &= (a+b)(1+\epsilon_1) && |\epsilon_1| \leq \upsilon, \\ a \otimes b &= (a \times b)(1+\epsilon_2) + \eta_2 && |\epsilon_2| \leq \upsilon, |\eta_2| \leq \eta. \end{aligned} \tag{49}$$

The value of υ only depends on the bitwise size of the mantissa. The value of η also depends on the bitwise size of the exponent. For example, the IEEE 754 double precision type allocates 53 bits to the mantissa, and 10 bits to the exponent, which means $\upsilon = 2^{-53}$ and $\eta = 2^{-1074}$ for the double precision type.

5.2.1 Interval Arithmetic

Here we introduce interval arithmetic, which will be used later in this chapter to bound floating-point computation errors. Let J be either a scalar of one, a vector of ones or a matrix of ones. Let a be either a scalar, a vector or a matrix. Let the dimensions of J be equal to the dimensions of a. For a scalar $\epsilon \geq 0$, an interval $\mathbf{a} = [a - \epsilon J, a + \epsilon J]$ has a center at a and a radius of ϵ. The function μ takes in an interval and returns its radius $\mu(\mathbf{a})$. The radius of an interval can also be indicated using a subscript i.e. $\mathbf{a}_\epsilon$. An interval $\mathbf{a}$ has a lower bound of $\underline{a}$ and an upper bound of $\overline{a}$. We overload operators $\{+, -\times, /\}$ to take intervals as arguments. They are defined such that for any $x \in \mathbf{a}$ and $y \in \mathbf{b}$,

$$\begin{aligned} x + y &\in \mathbf{a} + \mathbf{b} \\ x - y &\in \mathbf{a} - \mathbf{b} \\ x \times y &\in \mathbf{a} \times \mathbf{b} \\ x \,/\, y &\in \mathbf{a} \,/\, \mathbf{b}, 0 \notin \mathbf{b}. \end{aligned} \tag{50}$$

To satisfy the property in (50) when there are floating-point errors, interval arithmetic operators employ outward rounding. To illustrate outward rounding, we have

$$\mathbf{a} \oplus \mathbf{b} = \left[\mathcal{F}^{\downarrow}\left(\underline{a} + \underline{b}\right), \mathcal{F}^{\uparrow}\left(\overline{a} + \overline{b}\right)\right]. \tag{51}$$

Interval arithmetic is associative, commutative but only sub-distributive i.e. $\mathbf{a}\left(\mathbf{b} + \mathbf{c}\right) \subseteq \mathbf{ab} + \mathbf{ac}$ [63]. However, the distributivity property does hold for a special case.

Lemma 2.1 For intervals $\mathbf{a}, \mathbf{b}$ and $\mathbf{c}$, if $\mu(\mathbf{a}) = 0$ i.e. $\mathbf{a} = a \in \mathbb{R}$ then

$$a\left(\mathbf{b} + \mathbf{c}\right) = a\mathbf{b} + a\mathbf{c}. \tag{52}$$

This property extends to a matrix or vector interval multiplied by either a matrix or a vector. We have the following lemma which will be used later in this chapter.

Lemma 2.2 For $z \in \mathbb{R}^{l \times n}$ and $\mathbf{A}, \mathbf{B} \in \mathbb{IR}^{n \times m}$ then

$$z\left(\mathbf{A} + \mathbf{B}\right) = z\mathbf{A} + z\mathbf{B}. \tag{53}$$

Proof. For $i = 1, \ldots, l$, $\left(x\left(\mathbf{A} + \mathbf{B}\right)\right)(i) = \sum_{j=1}^{n} x_{ij}\left(\mathbf{A}_{ij} + \mathbf{B}_{ij}\right)$, which by Lemma (2.1) is exactly equal to $\sum_{j=1}^{n} x_{ij}\mathbf{A}_{ij} + x_{ij}\mathbf{B}_{ij}$. This is precisely the ith row of $x\mathbf{A} + x\mathbf{B}$.

5.2.2 Other Notations and Definitions

$\mathbb{R}$ denotes the set of real numbers. $\mathbb{F} \subset \mathbb{R}$ denotes a set of real numbers exactly representable by the floating-point number system. The entries of a matrix or a

vector starts at 0[1]. For a matrix expression A, $A(i,j)$ or $(A)(i,j)$ denotes the element in the $i-1$th row and $j-1$th column of A. For A, the notation $A(i,:)$ or $(A)(i,:)$ denotes its $i-1$th row. For a vector expression v, $v(i)$ or $(v)(i)$ denotes the $i-1$th element of v. The infinity norm of a vector $v \in \mathbb{R}^n$ is $\|v\|_\infty = \max_{0 \le i \le n-1} |v(i)|$. The infinity norm of a matrix $A \in \mathbb{R}^n$ is $\|A\|_\infty = \max_i \left\{ \sum_j |A(i,j)| \right\}$. The 2-norm of a vector $v \in \mathbb{R}^n$ is $\|v\|_2 = \sqrt{\sum_{i=0}^{n-1} v(i)^2}$.

5.3 Refinement of Credible Autocoding

In this section, we introduce a refinement of the credible autocoding process described in the previous chapter to account for floating-point computation errors.

5.3.1 Example of Credible Autocoding

The illustrating example is the linear system $x_+ = Ax$, in which $A = \begin{bmatrix} 0.4990 & 0.1 \\ 0.01 & 0.98 \end{bmatrix}$. Its C implementation is a while loop that updates the array variable x with the value of Ax during each iteration. The body of the loop is shown in Figure 32 along with the ACSL comments expressing the control semantics of the system. Here we review the predicate notation used to express ellipsoids on C code. For $M \in \mathbb{R}^n$, $M \succeq 0$, $x \in \mathbb{R}^n$, $c \in \mathbb{R}$ and $c \ge 0$, the family of predicates $q(M,c)(x)$ parameterized by M and c is defined as

$$q(M,c)(s) = \begin{bmatrix} c & s^\intercal \\ s & M \end{bmatrix} \succ 0. \tag{54}$$

Recall the notation $q(M,c)(x)$ is overloaded to indicate the ellipsoidal set $\{x \in \mathbb{R}^n \mid q(M,c)(x)\}$.

[1] To follow the convention in accessing arrays in C

The control semantics expressed in Figure 32 include the loop invariant $q(Q,1)(x)$, where $Q = \begin{bmatrix} 0.8879 & -0.1344 \\ -0.1344 & 0.53889 \end{bmatrix}$. The parameter Q was obtained from a stability analysis, They also include the ellipsoidal pre- and post-conditions generated for every line of code during the execution of *sp*-calculus i.e. a collection of ellipsoidal transformation rules for forward propagation. For example, line 11 of Figure 32 computes $0.4990x(0) + 0.1x(1)$, assigns the result to y_0. Line 11 is a linear transformation $T = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ 0.4990 & 0.1 \end{bmatrix}$ on x. Apply the affine transformation rule on the pre-condition $q(Q,1)(x)$ leads to the post-condition ellipsoid $q(Q_1,1)(s_1)$, $s_1 = \begin{bmatrix} x(0) & x(1) & y_0 \end{bmatrix}^\intercal$. The same procedure is applied to line 18 to get the post-condition ellipsoid $q(Q_2,1)(s_2)$, $s_2 = \begin{bmatrix} x(0) & x(1) & y_0 & y_1 \end{bmatrix}^\intercal$. The last post-condition generated is $q(Q_4,1)(x)$ in line 29 of Figure 32. This alternative post-condition of the loop body leads to an inductive condition $q(Q_4,1)(x) \rightarrow q(Q,1)(x)$, which can be verified by checking if the inclusion

$$q(Q_4,1)(x) \subseteq q(Q,1)(x) \tag{55}$$

holds. If (55) holds, then the loop invariant $q(Q,1)(x)$ is inductive and credible autocoding terminates with a positive result.

5.3.2 Sources of Floating-point Errors

The are two sources of floating-point computation errors that we have to consider. The primary floating-point errors come from the execution of the program on a target machine. Denote $\mathcal{M}_r$ as a virtual machine that takes as argument, a numerical

```
/*@
  requires q(Q,1)(x);
  ensures  q(Q,1)(x);
*/
{
  /*@
    requires q(Q,1)(x);
    ensures  q(Q1,1)(s1);
  */
  {
    y0=0.4990*x[0]+0.1*x[1];
  }
  /*@
    requires q(Q1,1)(s1);
    ensures  q(Q2,1)(s2);
  */
  {
    y1=0.01*x[0]+0.98*x[1];
  }
  /*@
    requires q(Q2,1)(s3);
    ensures  q(Q3,1)(s4);
  */
  {
    x[0]=y0;
  }
  /*@
    requires q(Q3,1)(s4);
    ensures  q(Q4,1)(x);
  */
  {
    x[1]=y1;
  }
}
```

Figure 32: $x_+ = Ax$ Annotated

expression $\mathcal{X}$, evaluates it with infinite-precision arithmetic, and returns the result $\mathcal{M}_r(\mathcal{X})$. For brevity's sake, instances of $\mathcal{M}_r(\mathcal{X})$ are abbreviated to $\mathcal{X}$ in this chapter. Denote $\mathcal{M}_t$ as another virtual machine with the same floating-point number system as the one on the target machine. For the example C code in Figure 32, we are looking for a bound on the error $\mathcal{M}_t(Ax) - \mathcal{M}_r(Ax)$.

The secondary floating-point errors are the byproducts of the computations performed in the credible autocoding process. We assume that credible autocoding is carried out by another machine represented by the virtual machine $\mathcal{M}_a$. As in the case of the target machine, we can set the rounding mode in $\mathcal{M}_a$ to the directions $+\infty$, $-\infty$, or 0. This is denoted by $\mathcal{M}_a^{\diamond}$, and $\diamond \in \{\uparrow, \downarrow, \epsilon\}$. The directional rounding is used to over-approximate the computations of upper bounds.

5.3.3 Credible Autocoding with Floating-point Errors

A method is proposed in this section, which makes credible autocoding sound with regards to floating-point computation errors. The main idea behind the method is that credible autocoding is already sound for programs executed with infinite-precision arithmetic. For these hypothetical programs, the proof annotations generated by the prototype autocoder are already correct modulo the secondary errors. We just need to check if the proof annotations are also correct on the actual program, which can be done as follows.

1. Compute a bound on the difference between the traces of the hypothetical program and the actual program.

2. Treat the bound computed in step 1 as a perturbation term, and check to see

if the original ellipsoid invariant is still inductive with the perturbation.

We now illustrate the method on the annotated C program example from Figure 32. Denote the C program from Figure 32 as $\mathcal{P}$. One can duplicate $\mathcal{P}$ and then replace all the floating-point variables and operations by their real number counterparts. The duplicate program is a hypothetical program, which we denote as $\mathcal{R}$. The duplicate program can be inserted into the original program $\mathcal{P}$ in the form of ACSL ghost code statements (see Figure 33). Ghost code with real semantics is included in the ACSL specifications though not implemented yet in Frama-C [7].

Notationally speaking, we now need to distinguish between the variables in $\mathcal{P}$ and their counterparts in $\mathcal{R}$. The floating-point variables shall be referenced with a tilde i.e. the array variable `x` is $\tilde{x}$. The real number variables in $\mathcal{R}$ shall be referenced without a tilde i.e. the array variable `x_real` is x. For $\mathcal{R}$, the ACSL contracts from Figure 32, including the loop invariant $q(Q, 1)(x)$, are already sound modulo the secondary errors produced by $\mathcal{M}_a$. For now, assume there are no secondary errors, which means the annotations from Figure 32 can be duplicated and inserted as correct annotations for $\mathcal{R}$. For $\mathcal{R}$, we can then say the loop invariant $q(Q, 1)(x)$ is correct i.e. the loop inductive condition

$$q(Q_4, 1)(x) \to q(Q, 1)(x) \tag{56}$$

holds. To express a similar loop invariant on the actual program $\mathcal{P}$, the floating-point analog of $q(Q, 1)(x)$ is introduced here. Let $M \in \mathbb{R}^{n\times n}$, $s \in \mathbb{R}^n$ and $M \succ 0$, we have a family of predicates

$$qf(M, c)(s) = q(M, c)(s) \wedge s \in \mathbb{F}^n. \tag{57}$$

```
/*@
  requires q(Q,1)(x_real);
  ensures  q(Q,1)(x_real);
*/
{
  /*@
    requires q(Q,1)(x_real);
    ensures  q(Q1,1)(s1_real);
  */
  {
    /*@
      ghost y0_real=0.4990*x_real[0]-0.1*x_real[1];
    */
    y0=0.4990*x[0]+0.1*x[2];
  }
  /*@
    requires q(Q1,1)(s2_real);
    ensures  q(Q2,1)(s3_real);
  */
  {
    /*@
      ghost y1_real=0.01*x_real[0]+0.98*x_real[1];
    */
    y1=0.01*x[0]+0.98*x[1];
  }
  /*@
    requires q(Q2,1)(s3_real);
    ensures  q(Q3,1)(s4_real);
  */
  {
    /*@
      ghost x_real[0]=y0_real;
    */
    x[0]=y0;
  }
  /*@
    requires q(Q3,1)(s4_real);
    ensures  q(Q4,1)(x_real);
  */
  {
    /*@
      ghost x_real[1]=y1_real;
    */
    x[1]=y1;
  }
}
```

Figure 33: Annotations for $\mathcal{R}$ embedded within $\mathcal{P}$

Inserting the candidate loop invariant $qf(Q,1)(\tilde{x})$ into $\mathcal{P}$ results in the first ACSL contract in line 1 of Figure 34. Since `x_real[0]=x[0]` and `x_real[1]=x[1]`, by the substitution rule, we get the post-condition $qf(Q,1)(x)$ in line . This completes the second ACSL contract in line in Figure 34. Note that $qf(Q,1)(x) \rightarrow q(Q,1)(x)$, hence by the consequent rule from Hoare logic, we can strengthen the pre-condition in line 7 of Figure 33 to $qf(Q,1)(x)$. This results in the pre-condition in line 16 of Figure 34. The remaining ellipsoidal invariants $q(Q_1,1)(s_1)$ to $q(Q_4,1)(x)$ hold as in Figure 33.

Now we take into account the primary floating-point error. We compute the small-step error incurred during execution of each line of $\mathcal{P}$. For line 17 of Figure 34, computing a $\delta_0 > 0$ such that $|\tilde{y}_0 - y_0| \leq \delta_0$ for all $\tilde{x}$ belonging to $qf(Q,1)(\tilde{x})$ results in the post-condition in line 12 of Figure 34. Next, for line 26 of Figure 34, computing a $\delta_1 > 0$ such that $|\tilde{y}_1 - y_1| \leq \delta_1$ for all $\tilde{x}$ belonging to $qf(Q,1)(\tilde{x})$ results in the post-condition in line 21 of Figure 34. We will describe later this chapter how to compute the error bounds δ_i. Moving further along into the code in Figure 34, we have the assignment statements `x[0]=y0;` and `x[1]=y1;`. The assignment operator does not cause floating-point computation errors, so we get the post-condition

$$\bigwedge_{i=0}^{1} |\tilde{x}(i) - x(i)| \leq \delta_i \tag{58}$$

for the loop body (see line 39 of Figure 34).

With the floating-point error bounds in Figure 34, the inductive condition for the loop invariant $qf(Q,1)(\tilde{x})$ becomes

$$q(Q_4,1)(x) \wedge |\tilde{x}(0) - x(0)| < \delta_0 \wedge |\tilde{x}(1) - x(1)| < \delta_1 \rightarrow qf(Q,1)(\tilde{x}). \tag{59}$$

```
/*@
  requires qf(Q,1)(x);
  ensures  qf(Q,1)(x);
*/
{ /*@
      requires qf(Q,1)(x);
      ensures  qf(Q,1)(x_real);
  */
  { /*@ ghost x_real[0]=x[0]; ghost x_real[1]=x[1]; */ }
  /*@
    requires qf(Q,1)(x_real);
    ensures  q(Q1,1)(s1_real) && abs(y0-y0_real)<delta_0;
  */
  { /*@
      ghost y0_real=0.4990*x_real[0]-0.1*x_real[1];
    */
    y0=0.4990*x[0]+0.1*x[2];
  }
  /*@
    requires q(Q1,1)(s2_real) && abs(y0-y0_real)<delta_0;;
    ensures  q(Q2,1)(s3_real) && abs(y0-y0_real)<delta_0 && abs(
       ↪ y1-y1_real)<delta_1;
  */
  { /*@
      ghost y1_real=0.01*x_real[0]+0.98*x_real[1];
    */
    y1=0.01*x[0]+0.98*x[1];
  }
  /*@
    requires q(Q2,1)(s3_real) && abs(y0-y0_real)<delta_0 && abs(
       ↪ y1-y1_real)<delta_1;;
    ensures  q(Q3,1)(s4_real) && abs(x[0]-x_real[0])<delta_0 && abs(
       ↪ y1-y1_real)<delta_1;
  */
  { /*@
      ghost x_real[0]=y0_real;
    */
    x[0]=y0;
  }
  /*@
    requires q(Q3,1)(s4_real) && abs(x[0]-x_real[0])<delta_0 && abs(
       ↪ y1-y1_real)<delta_1;
    ensures  q(Q4,1)(x_real) && abs(x[0]-x_real[0])<delta_0 && abs(x
       ↪ [1]-x_real[1])<delta_1;
  */
  { /*@
      ghost x_real[1]=y1_real;
    */
    x[1]=y1;
  }
}
```

Figure 34: Annotating $\mathcal{P}$ with Floating-point Error Bounds

Remark 5 A method will be presented later to soundly check the validity of (59). If (59) holds, then the candidate $qf(Q,1)(\tilde{x})$ is a loop invariant of $\mathcal{P}$, which means credible autocoding will terminate with a positive answer. If (59) does not hold, then credible autocoding is not feasible with the computed floating-point error bounds. However this outcome is unlikely to happen as floating-point errors are small compared to other uncertainties in a control system. With a well-designed controller, a closed-loop system should be stable for disturbances orders of magnitudes larger than floating-point computational noise. Later in this chapter, we present some numerical experimentations that give some idea on the minimum size of mantissa needed to ensure that credible autocoding with floating-point errors is feasible.

To summarize, we introduce the following modified credible autocoding process that is sound with respect to the primary floating-point errors:

1. Duplicate $\mathcal{P}$ with semantics of real numbers to create $\mathcal{R}$.

2. Insert the duplicate $\mathcal{R}$ into the original program $\mathcal{P}$ in the form of ACSL ghost code statements.

3. Execute credible autocoding as described in the previous chapter on $\mathcal{R}$.

4. Translate the loop invariant $q(Q,1)(x)$ from $\mathcal{R}$ into its floating-point analog $qf(Q,1)(\tilde{x})$, and then insert $qf(Q,1)(\tilde{x})$ as the candidate loop invariant for $\mathcal{P}$.

5. Perform floating-point error analysis of $\mathcal{P}$ to obtain bounds δ_i on $|\tilde{v}_i - v_i|$, where $\tilde{v}_i$ and v_i are respectively variables in $\mathcal{P}$ and their counterparts in $\mathcal{R}$.

6. Annotate each line of code in $\mathcal{P}$ with post-conditions $|\tilde{v}_i - v_i| \leq \delta_i$, where δ_i are the bounds computed in step 5. For the running example, this step results in the post-condition $\bigwedge_{i=0}^{0} |\tilde{x}(i) - x(i)| \leq \delta_i$ in line 42 of Figure 34.

7. Verify that the loop inductive condition holds with the floating-point error bounds. For the running example, this step involves checking that (59) holds.

In the next few sections, we give a solution to eliminate the secondary errors. We also give a formula to compute the bounds on the errors $\mathcal{M}_t\left(A(i,:)x\right) - Ax$. Finally we propose a method to numerically check the inductive condition in (59).

5.3.4 Secondary Errors

There are secondary errors as the result of the floating-point computations in the execution of *sp*-calculus on $\mathcal{R}$. A way to eliminate these errors is to compute the ellipsoid transformations exactly with rational numbers [50]. Since the matrix Q that defines the loop invariant $q(Q,1)(x)$ is computed using floats, we can be sure it does not contain any irrational parameters. In the C program, any irrational constant such as π is approximated using a floating-point number. Rational arithmetic are computationally more expensive. For the affine ellipsoid transformation rule, which is

$$T_i Q T_i^{\mathsf{T}}, \tag{60}$$

the number of arithmetic operations grows cubically with the dimensions of Q. However, we only need to use rational arithmetic once and in an off-line fashion during the generation of the annotations. For the degenerate ellipsoids such as $q(Q_1,1)(s_1)$

where Q_1 is singular, using exact computations in ellipsoid transformations avoids the problem of showing that $\tilde{Q}_1$, computed with floats, is positive-semidefinite.

5.3.5 Bounding the Floating-point Errors

We give a formula to compute explicit values for error bounds in (58). The problem is computing $\delta_i > 0$ such that $|\mathcal{M}_t\left(A(i,:)x\right) - A(i,:)x| \leq \delta_i$, $i = 0, \ldots, n-1$. Note for the example $x_+ = Ax$, useful error bounds can be computed because we can make the assumption that x belongs to the bounded set $qf(Q,1)(x)$.

Algorithm 1 Algorithm to compute the dot product of $a, b \in \mathbb{F}^n$

1. For a set $S = \{a_1 b_1, \ldots, a_n b_n\}$ with at least two elements. Choose any two elements from the set.
2. Evaluate the elements if needed.
3. Sum the two elements together and put the result back into the set S.
4. Repeat until S has fewer than 2 elements.

For Algorithm 1, which allows arbitrary order of computations, we have a classic result from Higham [41].

Theorem 3.1 Consider two vectors $a, b \in \mathbb{F}^n$ and their dot product $a^\intercal b$, then

$$|\mathcal{M}_t\left(a^\intercal b\right) - a^\intercal b| \leq \frac{n\upsilon}{1 - n\upsilon} a^\intercal b. \tag{61}$$

Next, we have Algorithm 2, which computes $A(i,:)x$, $i = 0, \ldots, n-1$ in any order permitted by Algorithm 1. Since the order of computations in Algorithm 1 is arbitrary, so is the order of computations in Algorithm 2. For the autocoding prototype, it is better to have an error bound that works for any order of computation,

Algorithm 2 Algorithm for $x_+ = Ax$

1. For $i = 0, \ldots, n-1$, let $a = A(i,:)^\intercal$ and $b = x$, compute $a^\intercal b$ using Algorithm 1 and then assign the result to $y(i)$.
2. Update $x(i)$ with $y(i)$ from step 1 for $i = 0, \ldots, n-1$.

since it is not known before code generation how Ax is computed. Using (61), a bound on $\mathcal{M}_t\left(A(i,:)x\right) - A(i,:)x$ can be obtained using the following result.

Proposition 5.3.1 For x belonging to $qf(Q,1)(x)$, $|\mathcal{M}_t\left(A(i,:)x\right) - A(i,:)x| \leq \delta_i = \dfrac{n\upsilon}{1-n\upsilon}\sqrt{A(i,:)QA^\intercal(i,:)}$.

Proof. The proofs follows from Theorem 3.1 and the fact that $A(i,:)x \leq \sqrt{A(i,:)QA(i,:)^\intercal}$ for x in $q(Q,1)(x)$.

Remark 6 To soundly over-approximate the computation of δ_i, we can employ directed rounding in the analyzer machine. The numerator is over-approximated while the denominator is under-approximated. We get

$$\delta_i = \mathcal{M}_a^{\uparrow}\left(\frac{\mathcal{M}_a^{\uparrow}\left(n\upsilon\sqrt{A(i,0)^2Q(0,0)+,\ldots,+A(i,n-1)^2Q(n-1,n-1)}\right)}{\mathcal{M}_a^{\downarrow}\left(1-n\upsilon\right)}\right) \tag{62}$$

5.3.6 Verification of the Inductive Condition with Floating-point Errors

In this section, we present a sound method to verify the inductive condition (59). First we rewrite (59) for the general case of $x_+ = Ax$, $A \in \mathbb{F}^{n\times n}$. For $M_1 \in \mathbb{R}^{n\times n}$, $M_2 \in \mathbb{R}^{n\times n}$, $M_2 \succ M_1 \succ 0$, scalars $\delta_i > 0$, $x \in \mathbb{R}^n$, and $\tilde{x} \in \mathbb{F}^n$, we want to check whether

$$\bigwedge_{i=0}^{n-1} |\tilde{x}(i) - x(i)| \leq \delta_i \wedge q(M_1,1)(x) \rightarrow qf(M_2,1)(\tilde{x}). \tag{63}$$

Let $\delta = \max_{0 \leq i \leq n-1} \{\delta_i\}$. Let $S_{M_1,\delta} = \left\{ \tilde{x} \in \mathbb{F}^n \mid q(M_1, 1)(x) \wedge \bigwedge_{i=0}^{n-1} |\tilde{x}(i) - x(i)| < \delta_i \right\}$. Checking (63) is equivalent to checking the set inclusion

$$S_{M_1,\delta} \subseteq qf(M_2, 1)(\tilde{x}). \tag{64}$$

5.3.6.1 Cholesky Decomposition

The Cholesky decomposition is introduced here since it will be used to check (63). Cholesky decomposition (see Algorithm 3) is an algorithm that decomposes a positive-definite matrix M into the product of a lower triangular matrix L and its transpose L^{T}. If M is nearly singular, then the Cholesky algorithm can fail due to round-off

Algorithm 3 Cholesky Decomposition Algorithm

1. $L(i,j) = \dfrac{1}{L(j,j)} \left(A(i,j) - \sum_{k=0}^{j-2} L(i,k)L(j,k) \right)$ for $i > j$.
2. $L(j,j) = \sqrt{M(j,j) - \sum_{k=0}^{j-2} L(j,k)^2}$

errors. This failure usually occurs when computing the diagonal entry $L(j,j)$, which requires computing the square root of the value $D(j,j) = M(j,j) - \sum_{k=0}^{j-2} L(j,k)^2$. If we assume exact computations, then the algorithm should always terminate with a positive answer when the input is a positive-definite matrix. With inexact computations, the algorithm may return false positives or false negatives. In this chapter, we use an interval Cholesky decomposition algorithm, which accounts for all the round-off errors produced during its execution. In an interval Cholesky algorithm, all the numerics are replaced by intervals, and the floating-point operators are replaced by their interval

counterparts. The complexity of an interval Cholesky decomposition algorithm with regards to the dimensions of the matrix input is the same as the regular algorithm. An interval Cholesky algorithm has the advantage of producing only false negatives i.e. inputs that are positive-definite but cannot be determined to be so due to floating-point errors. More importantly, the interval Cholesky algorithm can also be used to check efficiently if all matrices belonging to the interval $\mathbf{M}$ are positive-definite. To see this is true, note that due to the fundamental property of interval arithmetic operators in (50), the interval $\mathbf{D}(j,j) = \mathcal{M}_a\left(\mathbf{M}(j,j) - \sum_{k=0}^{j-2} \mathbf{L}(j,k)^2\right)$ contains, for $M(j,j) \in \mathbf{M}(j,j)$. all possible values of $\mathcal{M}_a\left(M(j,j) - \sum_{k=0}^{j-2} L(j,k)^2\right)$.

5.3.6.2 Verification Method

Before we present the method to check the inductive condition in (63), first we have a pair of lemmas, which allows us to verify inclusions between sets of floats by checking their corresponding sets of reals.

Lemma 3.2 Let S be a subset of $\mathbb{F}^n$. Let $M \in \mathbb{R}^{n\times n}$ be a positive-definite matrix. If $S \subset q(M,1)(x)$, then $S \subseteq qf(M,1)(x)$.

Proof. If $S \subset q(M,1)(x)$, then for all $x \in S$, $x^\intercal M^{-1} x \leq 1$, which implies that every member x of S is also a member of $qf(M,1)(x)$.

Lemma 3.3 Let $M_1 \in \mathbb{R}^{n\times n}$ be a positive-definite matrix. Let $M_2 \in \mathbb{R}^{n\times n}$ be another positive-definite matrix. If $q(M_1,1)(x) \subset q(M_2,1)(x)$, then $qf(M_1,1)(x) \subseteq qf(M_2,1)(x)$.

Proof. If $qf(M_1, 1)(x) \not\subseteq qf(M_2, 1)(x)$, then there exists $z \in \mathbb{R}^n$ such that $qf(M_1, 1)(z)$ is true, and $z^\intercal M_2^{-1} z > 1$, Hence $qf(M_1, 1)(x) \not\subseteq qf(M_2, 1)(x)$ implies $q(M_1, 1)(x) \not\subset q(M_2, 1)(x)$. This completes the proof.

Now we are ready to give the main result for verifying (63). To check that $S_{M_1,\delta}$ is contained within $q(M_2, 1)(\tilde{x})$, we can search for an ellipsoid $q(\gamma M_1, 1)(x)$ that is similar to $q(M_1, 1)(x)$, but scaled with the factor $\gamma > 1$ such that $q(\gamma M_1, 1)(x)$ contains $S_{M_1,\delta}$.

Theorem 3.4 If there exist scalar $\gamma > 1$ such that $S_{M_1,\delta} \subset q(\gamma M_1, 1)(x)$ and $q(\gamma M_1, 1)(x) \subset q(M_2, 1)(x)$, then the inductive condition in (63) holds.

Proof. By Lemma 3.2, $S_{M_1,\delta} \subset q(\gamma M_1, 1)(x)$ implies $S_{M_1,\delta} \subseteq qf(\gamma M_1, 1)(x)$. By Lemma 3.3, $q(\gamma M_1, 1)(x) \subset q(M_2, 1)(x)$ implies $qf(\gamma M_1, 1)(x) \subseteq qf(M_2, 1)(x)$. The sets $qf(M_2, 1)(x)$ and $qf(M_2, 1)(\tilde{x})$ are equivalent, hence we have that $S_{M_1,\delta} \subseteq qf(M_2, 1)(\tilde{x})$. Using theorem 3.4, we can check if (63) holds using the following steps:

1. Computing a scaling factor $\gamma > 1$ such that ellipsoid $q(\gamma M_1, 1)(x)$ contains $S_{M_1,\delta}$.

2. Checking if $q(\gamma M_1, 1)(x) \subset q(M_2, 1)(x)$ holds. This can be done by checking $M_2 - \gamma M_1 \succ 0$ using an interval Cholesky algorithm.

Here we give a sound method to compute the scaling factor γ. Denote the minimum eigenvalue of M_1 as λ_{min}. For two similar and concentric ellipsoids $q(M_1, 1)(x)$ and $q(\gamma M_1, 1)(x)$, the minimum distance between their boundaries is $\sqrt{\gamma\lambda_{min}} - \sqrt{\lambda_{min}}$. We also have $|\tilde{x}(i) - x(i)| \leq \delta_i$, which implies $\|\tilde{x} - x\|_2 \leq \sqrt{\sum_i \delta_i^2}$. Hence a

good choice of γ is one that ensures $\sqrt{\gamma\lambda_{min}} - \sqrt{\lambda_{min}} \geq \sqrt{\sum_i \delta_i^2}$, which is satisfied by

$$\gamma = \left(\frac{\sqrt{\lambda_{min}} + \sqrt{\sum_i \delta_i^2}}{\sqrt{\lambda_{min}}} \right)^2 . \tag{65}$$

We have the follwoing procedure to compute the scaling factor γ soundly.

1. Compute a matrix interval $\mathbf{M}_1$ such that $M_1 \in \mathbf{M}_1$. This is done because M_1 is computed using *sp*-calculus, which we assume that we are going to use rational arithmetic for.

2. Compute the minimum eigenvalue of M_1 use any off-the-shelf numerical algorithm. Let the result of that be $\tilde{\lambda}$. Check if $\mathbf{M}_1 - \tilde{\lambda} I \succ 0$ with an interval Cholesky algorithm. If the answer is negative then decrease $\tilde{\lambda}$ until it returns a positive result.

3. Compute the scaling factor γ by replacing the λ_{min} in (65) with $\tilde{\lambda}$ computed in step 2. Apply directional rounding to either over-approximate or under-approximate when evaluating (65) i.e. use $\mathcal{M}_a^{\uparrow}$ on the numerator in (65) and use $\mathcal{M}_a^{\downarrow}$ on the denominator.

5.3.7 Numerical Values

This section populates the annotation variables used in Figure 34 with their numerical definitions. The values are computed using results from Sections5.3.5 and 5.3.6. The analyzer machine $\mathcal{M}_a$ has the floating-point accuracy of IEEE 754 double-precision type or a machine epsilon of $\upsilon = 2^{-53}$. The computations with rationals are carried out using the symbolic toolbox of Matlab. The interval arithmetic and directed

rounding are provided by the interval arithmetic package INTLAB [81]. An interval Cholesky algorithm is implemented by replacing all variables and operators in Algorithm 3 with their interval counterparts.

First, we computed a P such that P satisfies the Lyapunov equation $A^{\intercal}PA - P \prec 0$. The computed $P \succ 0$ is inverted using $\mathcal{M}_a$ and we get the loop invariant $q(Q, 1)(x)$, where

$$Q = \begin{bmatrix} \frac{181648207}{204580391} & -\frac{25726097}{191463773} \\ -\frac{25726097}{191463773} & \frac{61289389}{113736401} \end{bmatrix}. \tag{66}$$

Next, the ellipsoid transformations are computed exactly using rational arithmetic. The transformations matrices $T_i, i = 1, \ldots, 4$ and the resulting ellipsoidal post-conditions $q(Q_i, 1)(s_i), i = 1, \ldots, 4$ are as follows.

$$T_1 = \begin{bmatrix} 1 & 0 \\ 0 & 1 \\ \frac{499}{1000} & \frac{1}{10} \end{bmatrix}. \tag{67}$$

$$Q_1 = \begin{bmatrix} \frac{499846779888669}{562949953421312} & -\frac{302563871607443}{2251799813685248} & \frac{60464861593564939}{140737488355328000} \\ -\frac{302563871607443}{2251799813685248} & \frac{4853729624558199}{9007199254740992} & -\frac{14818065659079541}{1125899906842624000} \\ \frac{60464861593564939}{140737488355328000} & -\frac{14818065659079541}{1125899906842624000} & \frac{59973480228900820597}{281474976710656000000} \end{bmatrix}. \tag{68}$$

$$T_2 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \\ \frac{1}{100} & \frac{49}{50} & 0 \end{bmatrix}. \tag{69}$$

$$Q_2 = \begin{bmatrix} \frac{499846779888669}{562949953421312} & -\frac{302563871607443}{2251799813685248} & \frac{60464861593564939}{140737488355328000} & -\frac{13825936148987369}{112589990684262400} \\ -\frac{302563871607443}{2251799813685248} & \frac{4853729624558199}{9007199254740992} & -\frac{14818065659079541}{1125899906842624000} & \frac{47445524772027373}{90071992547409920} \\ \frac{60464861593564939}{140737488355328000} & -\frac{14818065659079541}{1125899906842624000} & \frac{59973480228900820597}{281474976710656000000} & -\frac{484225770920637753}{562949953421312000} \\ -\frac{13825936148987369}{112589990684262400} & \frac{47445524772027373}{90071992547409920} & -\frac{484225770920637753}{562949953421312000} & \frac{11596501696848731647}{22517998136852480000} \end{bmatrix}. \tag{70}$$

$$T_3 = \begin{bmatrix} 0 & 0 & 1 & 0 \\ 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 \end{bmatrix}. \tag{71}$$

$$Q_3 = \begin{bmatrix} \frac{59973480228900820597}{281474976710656000000} & -\frac{14818065659079541}{1125899906842624000} & -\frac{484225770920637753}{562949953421312000} \\ -\frac{14818065659079541}{1125899906842624000} & \frac{4853729624558199}{9007199254740992} & \frac{47445524772027373}{90071992547409920} \\ -\frac{484225770920637753}{562949953421312000} & \frac{47445524772027373}{90071992547409920} & \frac{11596501696848731647}{22517998136852480000} \end{bmatrix}. \tag{72}$$

$$T_4 = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix}. \tag{73}$$

$$Q_4 = \begin{bmatrix} \frac{59973480228900820597}{281474976710656000000} & -\frac{484225770920637753}{562949953421312000} \\ -\frac{484225770920637753}{562949953421312000} & \frac{11596501696848731647}{22517998136852480000} \end{bmatrix}. \tag{74}$$

Assume $\mathcal{M}_t$ has a floating-point accuracy of IEEE 754 single-precision type or $\upsilon = 2^{-23}$. To compute the floating-point error bounds in (58), we apply Proposition 5.3.1 with $n = 2$ and we get

$$\begin{aligned} \delta_0 &= 1.100525040801444 \times 10^{-7} \\ \delta_1 &= 1.710955867399860 \times 10^{-7}. \end{aligned} \tag{75}$$

for the error bounds. Now we apply Theorem 3.4 by computing a scaling factor γ such that $S_{Q_4,\delta} \subseteq q(\gamma Q_4, 1)(x)$. Since Q_4 in (74) is not representable with floats, we over-approximate it with the interval $\mathbf{Q}_4$ such that $\mathcal{M}_a^{\uparrow}(Q_4), \mathcal{M}_a^{\downarrow}(Q_4) \in \mathbf{Q}_4$. The rational representation of $\mathbf{Q}_4$ is

$$\mathbf{Q}_4 = \begin{bmatrix} \left[\frac{26343265}{123637479}, \frac{36757143}{172513183}\right] & \left[-\frac{2151750}{250157599}, -\frac{5684351}{660849819}\right] \\ \left[-\frac{2151750}{250157599}, -\frac{5684351}{660849819}\right] & \left[\frac{46885511}{91041926}, \frac{43708163}{84872176}\right] \end{bmatrix}. \tag{76}$$

Note that (76) is exactly representable with floats. To compute soundly the minimum of the smallest eigenvalues of the matrices in $\mathbf{Q}_4$, we search for a $\hat{\lambda} > 0$ such that $\mathbf{Q}_4 - \hat{\lambda} I \succ 0$. The initial guess for $\hat{\lambda}$ is the minimum eigenvalue of $\mathcal{M}_a(Q_4)$ computed using the eigenvalue function in Matlab. This value i.e. $\lambda_{min} = 0.212823746331862$ resulted in $\mathbf{Q}_4 - \lambda_{min} \succ 0$ failing the interval Cholesky check. We rescaled λ_{min} by a factor of 0.9999 to obtain

$$\hat{\lambda} = 0.212802463957228. \tag{77}$$

Using the interval Cholesky algorithm, we have $\mathbf{Q}_4 - \hat{\lambda} I \succ 0$ with $\hat{\lambda}$ from (77). By substituting λ_{min} and δ_i in (65) with (77) and (75), we obtain the scaling factor $\gamma = 1.000000881991857$. Finally, note that Q from (66) is exactly representable using floats since it is the output of a Matlab inverse algorithm. The Cholesky decomposition of $Q - \gamma \mathbf{Q}_4$ returns a positive result which completes the credible autocoding process.

5.4 Stability Proofs with Floating-point Errors

In the last part of this chapter, we look for answers to the following two questions.

1. Can we check soundly without going through credible autocoding, if the stability proof generated for the control system, which is valid with exact computations, is also valid with floating-point errors?

2. At what bit-size of mantissa should we expect that credible autocoding will fail?

The first question arises from the concern about credible autocoding being non-iterative. For example, if the inductive condition does not hold, then credible autocoding terminates with a negative result. It is not an iterative procedure that updates the loop invariant until the inductive condition holds. For this reason, we need to have some idea beforehand if credible autocoding will terminate with a positive result. This was usually the case when floating-point errors were ignored. However by taking floating-point computation errors into account, credible autocoding could be infeasible if the size of the error bounds in Figure 34 are large. The second question is related to the first question. It is partly motivated by the hypothesis that floating-point errors are very small, hence only a toy floating-point number system can make credible autocoding infeasible for most cases. The first question is addressed in this section. The second question is addressed in the section on numerical experimentations.

For linear systems, checking if a stability proof still holds with floating-point errors can be reduced to a problem of checking the positive-definite property of a matrix interval.

Proposition 5.4.1 Consider the linear system $x_+ = Ax$, $A \in \mathbb{F}^{n\times n}$ and its C program implementation $\mathcal{P}$. Also consider $\mathcal{R}$, which is a duplicate of $\mathcal{P}$ with semantics

of real numbers. Assume the ellipsoid $p(P,1)(x)$ is an inductive invariant of $\mathcal{R}$ and $P \succ 0$ is a matrix such that $A^\intercal PA - P \prec 0$. Let $y = Ax$, and let $\tilde{y} = \mathcal{M}_t(Ax)$. Assume the error $\tilde{y} - y$ is bounded by $\delta > 0$ for any x belonging to $pf(P,1)(x)$. Let $\mathbf{y}_\delta$ denote the interval $[y - \delta, y + \delta]$. Assume there exists $\beta > 0$ such that $\mathbf{y}_\delta \subseteq \mathbf{A}_\beta x$ for all x belonging to $pf(P,1)(x)$. The set $pf(P,1)(x)$ is guaranteed to be an inductive invariant of $\mathcal{P}$ if

$$P - \mathbf{A}_\beta^\intercal P \mathbf{A}_\beta \succ 0. \tag{78}$$

Proof. By the definition of δ, we have that $\mathcal{M}_t(Ax) \in \mathbf{y}_\delta$ for all x in $pf(P,1)(x)$. If $\mathbf{y}_\delta \subseteq \mathbf{A}_\beta x$ for all x in $pf(P,1)(x)$, then $\mathcal{M}_t(Ax) \in \mathbf{A}_\beta x$ for all x in $pf(P,1)(x)$. From (78), we also have that $\mathbf{A}_\beta^\intercal P \mathbf{A}_\beta - P \prec 0$, which implies, for all x in $pf(P,1)(x)$, $x^\intercal \left(\mathbf{A}_\beta^\intercal P \mathbf{A}_\beta - P\right) x < 0$. By the distributive property in lemma 2.2, we have

$$x^\intercal \left(\mathbf{A}_\beta^\intercal P \mathbf{A}_\beta - P\right) x < 0 \implies \left(x^\intercal \mathbf{A}_\beta^\intercal P \mathbf{A}_\beta x - x^\intercal P x\right) < 0 \tag{79}$$

for all x in $pf(P,1)(x)$. Recall that $\mathcal{M}_t(Ax) \in \mathbf{A}_\beta x$ for all x in $pf(P,1)(x)$, hence

$$\mathcal{M}_t(Ax)^\intercal P \mathcal{M}_t(Ax) - x^\intercal P x < 0 \tag{80}$$

for all x in $pf(P,1)(x)$.

Remark 7 Using Proposition 5.4.1, we can check the feasibility of credible autocoding beforehand by running an interval Cholesky decomposition algorithm on the matrix interval in (78). This is assuming that we can compute a $\beta > 0$ such that $\mathbf{y}_\delta \subseteq \mathbf{A}_\beta x$ for all x in $pf(P, x1)(x)$. In the next section, we give a sound method to compute $\beta > 0$ that only depends on the properties of the matrix A and the machine epsilon ν.

The result in Proposition 5.4.1 extends to a linear system with bounded input u. For the linear system $x_+ = Ax + Bu$, $A \in \mathbb{F}^{n\times n}$, $B \in \mathbb{F}^{n\times m}$, the analog of (78) is

$$\begin{bmatrix} \mathbf{A}_\beta^\mathsf{T} P \mathbf{A}_\beta - P + \alpha P & \mathbf{A}_\beta^\mathsf{T} P \mathbf{B}_\beta \\ \mathbf{B}_\beta^\mathsf{T} P \mathbf{A}_\beta & \mathbf{B}_\beta^\mathsf{T} P \mathbf{B}_\beta - \alpha I \end{bmatrix} \prec 0, \tag{81}$$

for a multiplier $\alpha > 0$. Since checking (81) is exactly the same type of problem as checking (78), we can apply the interval Cholesky method.

5.4.1 Algebraic Expression for the Error Bounds

In this section, we give an algebraic expression for estimating the quantity β from Proposition 5.4.1. The results in this section hold for Algorithm 2, which allows arbitrary order of computations.

First we obtain an alternative bound $\hat{\delta}$, which is a function of $\|x\|_\infty$, on $\|\mathcal{M}_t(Ax) - Ax\|_\infty$.

Proposition 5.4.2 Let $A \in \mathbb{F}^{n\times n}$ and $x \in \mathbb{F}^n$. Let $\tilde{y} = \mathcal{M}_t(Ax)$ and $y = Ax$. Let $\theta = \|A\|_\infty$, and $\phi = \|x\|_\infty$. If $nu \leq 0.5$, where u is the machine epsilon of $\mathcal{M}_t$, then

$$\hat{\delta} = \frac{nu}{1-nu}\theta\phi \tag{82}$$

is a bound for $\|\mathcal{M}_t(Ax) - Ax\|_\infty$.

Proof. From (61), we have that

$$|\mathcal{M}_t(A(i,:)x) - A(i,:)x| \leq \frac{nu}{1-nu}A(i,:)x \leq \frac{nu}{1-nu}\sum_j A(i,j)\phi \leq \frac{nu}{1-nu}\sum_j |A(i,j)|\phi. \tag{83}$$

By the definition of θ, we have $\theta \geq \sum_j |A(i,j)|$ for $i = 0, \ldots, n-1$. Hence $|\mathcal{M}_t(A(i,:)x) - A(i,:)x| \leq \frac{nu}{1-nu}\theta\phi$ for $i = 0, \ldots, n-1$.

Remark 8 Note that n, θ, and ν are parameters of the system. Also note that ϕ depends on x, which is a variable. The expression in (82) results in a family of functions

$$\hat{\delta}(n, \theta, \nu)(\phi) = \frac{n\upsilon}{1 - n\upsilon}\theta\phi, \tag{84}$$

parameterized by n, θ, and υ. The parameters θ and n do not vary for a fixed A. The machine epsilon ν is a property of the target architecture and is also constant during runtime. For fixed parameters n, θ, ν, $\hat{\delta}(n, \theta, \nu)(\phi)$ can be simplified to $\hat{\delta}(\phi)$.

With the error bound function $\hat{\theta}$ from (84), we can find a $\beta > 0$ that is independent of x. We have the following Proposition.

Proposition 5.4.3 Let $\phi = \|x\|_\infty$, $\theta = \|A\|_\infty$, and $\hat{\delta}(x) > 0$ be the error bound function from (84). Let $y = Ax$ and let $\mathbf{y}_\delta = [y - \delta, y + \delta]$, in which $\delta = \hat{\delta}(x)$. If $\beta = \dfrac{n\upsilon}{1 - n\upsilon}\theta$, then for all $x \in \mathbb{F}^n$,

$$\mathbf{y}_\delta \subseteq \mathbf{A}_\beta x. \tag{85}$$

Proof. Let $\hat{\mathbf{0}}$ and $\bar{\mathbf{0}}$ denote respectively a vector and matrix intervals, both centered at 0. Since $\delta = \hat{\theta}(x) = \dfrac{n\upsilon}{1 - n\upsilon}\theta\phi$, $\beta = \dfrac{n\upsilon}{1 - n\upsilon}\theta$ implies $\beta = \dfrac{1}{\phi}\delta$. If $\beta = \dfrac{1}{\phi}\delta$ then,

$$\left(\bar{\mathbf{0}}_\beta x\right)(j) = \left[-\frac{1}{\phi}\delta\sum_{i=1}^{n}|x_i|,\ \frac{1}{\phi}\delta\sum_{i=1}^{n}|x(i)|\right],\ j = 0, \ldots, n-1. \tag{86}$$

Since $\phi = \|x\|_\infty \leq \displaystyle\sum_{i=0}^{n-1}|x(i)|$, then $[-\delta, \delta] \subseteq \left(\bar{\mathbf{0}}_\beta x\right)(j)$, $j = 0, \ldots, n-1$. Hence we have that $\hat{\mathbf{0}}_\delta \subseteq \bar{\mathbf{0}}_\beta x$, which implies, for $y = Ax$, $\mathbf{y}_\delta = y + \hat{\mathbf{0}}_\delta \subseteq Ax + \bar{\mathbf{0}}_\beta x$. By Lemma 2.2, we have $Ax + \bar{\mathbf{0}}_\beta x = \left(A + \bar{\mathbf{0}}_\beta\right)x = \mathbf{A}_\beta x$.

Remark 9 To over-approximate the bound $\beta = \dfrac{n\upsilon}{1-n\upsilon}\theta$, we can set appropriate directional rounding in the analyzer machine to over-approximate on the numerator $n\upsilon\theta$, and under-approximate on the denominator $1 - n\upsilon$. For the linear system $x_+ = Ax + Bu$ with bounded input u, the results from Proposition 5.4.3 on β applies with only slight modifications. The modifications include replacing the dimensional parameter n by $n + m$, and let $\theta = \| \begin{bmatrix} A & B \end{bmatrix} \|_\infty$.

5.5 Numerical Experimentation

In this section, we describe some numerical experimentations of using the proof-checking technique described in Section 5.4 to check if stability proofs are still valid with floating-point computation errors. The purpose of this experimentation is to find an approximate estimate for the smallest bit-size wise mantissa or the largest machine epsilon ν that could cause credible autocoding to become infeasible.

For linear systems $x_+ = Ax$, $A \in \mathbb{F}^{n\times n}$, and $A^\intercal PA - P \prec 0$, we check if $P - \mathbf{A}_\beta^\intercal P\mathbf{A} \succ 0$ using the same interval Cholesky algorithm used in Section 5.3.7. We consider variations in the dimensions of A, the machine epsilon ν and the spectral radius $\rho(A) = \max\{|\lambda_1|, \ldots, |\lambda_n\}$ where λ_i are the eigenvalues of A. We have the following two scenarios.

1. Varying $\rho\,(A)$ and n.

2. Varying $\rho\,(A)$, n, and the machine epsilon υ of $\mathcal{M}_t$.

The system matrices A are randomly generated using $U\Sigma U^{\mathrm{T}}$ where Σ is a full-rank diagonal that is stable i.e. $\rho\,(\Sigma) < 1$, and U is the left matrix from a singular value

decomposition of a randomly generated full-rank square matrix. For the first scenario, we assume a IEEE 754 single precision system for $\mathcal{M}_t$ i.e. $\upsilon = 2^{-23}$. The analyzer machine $\mathcal{M}_a$ has a IEEE 754 double precision system. The variation of n is from 14 to 86. The spectral radius of A are varied according to the scheme $\rho_i = \sum_{j=1}^{i} 9 \times 10^{-j}$. A portion of the numerical results are listed in Table (3). A positive result is indicated by 1 and a negative result is indicated by 0. A positive result means the stability proof holds and implies that credible autocoding is feasible. Since here we are using a more conservative estimate of the bound on the floating-point error than the interval analysis method proposed for credible autocoding in Section 5.3.5, a negative result only means that credible autocoding may fail. As expected, as the spectral radius of

Table 3: Varying spectral radius and dimension of A

	$\rho = 0.9$	$\rho = 0.99$	$\rho = 0.999$	$\rho = 0.9999$
$n = 14$	1	1	1	0
$n = 23$	1	1	1	0
$n = 32$	1	1	0	0
$n = 41$	1	1	0	0
$n = 50$	1	0	0	0
$n = 59$	1	0	0	0

the system matrix approaches 1, the proof checking technique from Section 5.4 fails for all $n \geq 14$.

In the second scenario, the quality of the floating-point number system on the target machine is incorporated into our analysis. We increase the machine epsilon υ of $\mathcal{M}_t$ from the double-precision of 2^{-53} to 2^{-8} until the proof-checking process fails. We also vary the dimensions of A and the spectral radii as done in scenario 1. The setup is as follows. We have n ranging from 5 to 50 and the spectral radii ranges

from 0.9 to 0.99999. The results are listed in Table 4. The entries of Table 4 are the machine epsilons at which the proof-checking process returned a negative result.

Table 4: Maximum machine epsilon ν

	$\rho = 0.9$	$\rho = 0.99$	$\rho = 0.999$	$\rho = 0.9999$	$\rho = 0.99999$
$n = 5$	2^{-9}	2^{-11}	2^{-15}	2^{-18}	2^{-22}
$n = 14$	2^{-13}	2^{-16}	2^{-19}	2^{-22}	2^{-26}
$n = 23$	2^{-15}	2^{-18}	2^{-21}	2^{-25}	2^{-28}
$n = 32$	2^{-17}	2^{-20}	2^{-23}	2^{-26}	2^{-30}
$n = 41$	2^{-18}	2^{-21}	2^{-24}	2^{-27}	2^{-31}
$n = 50$	2^{-19}	2^{-22}	2^{-24}	2^{-28}	2^{-32}

5.5.1 Control System Example

Lastly, we try the same proof-checking process on a linear system with inputs. Consider a lead-lag compensator that stabilizes a double integrator system with closed-loop damping of 0.583. The lead-lag compensator

$$\begin{aligned} x_+ &= \begin{bmatrix} 0.9940 & -0.0005 \\ 0.01 & 1.000 \end{bmatrix} x + \begin{bmatrix} 0.01 \\ 0 \end{bmatrix} y \\ u &= \begin{bmatrix} 3.6667 & 1.4167 \end{bmatrix} x + 5y \end{aligned} \tag{87}$$

is discretized using a Euler scheme at a sample rate of 0.01. Using the tool SeDuMi [88], we get the certificate of stability

$$P = \begin{bmatrix} 0.005291827052310 & 0.000471456978220 \\ 0.000471456978220 & 0.000278807543788 \end{bmatrix} \tag{88}$$

for $\alpha = 1e-4$. Assume IEEE single precision for $\mathcal{M}_t$, we get a β of 2.1191×10^{-5}. From the matrix error length, we get

$$\begin{aligned} \mathbf{A} &= \begin{bmatrix} [0.9939, 0.9941] & [-0.0006, -0.0004] \\ [0.0099, 0.0101] & [0.9999, 1.0001] \end{bmatrix} \\ \mathbf{B} &= \begin{bmatrix} [0.0099, 0.0101]] \\ [-0.0001, 0.0001] \end{bmatrix} \end{aligned} \tag{89}$$

for the interval $\mathbf{T}_\beta = \begin{bmatrix} \mathbf{A}_\beta & \mathbf{B}_\beta \end{bmatrix}$. Note that the numbers above are truncated for display purpose. From $\mathbf{T}_\beta$, we get the interval linear matrix inequality

$$1 \times 10^{-3} \times \begin{bmatrix} [0.0531, 0.0535] & [0.0025, 0.0027] & [-0.0528, -0.0525] \\ [0.0025, 0.0027] & [0.0003, 0.0004] & [-0.0047, -0.0046] \\ [-0.0528, -0.0525] & [-0.0047, -0.0046] & [0.1000, 0.1001] \end{bmatrix} \prec 0, \tag{90}$$

which we verified with an interval Cholesky Decomposition algorithm implemented in INTLAB.

5.6 Conclusion

In this chapter, we have provided a method to account for floating-point errors in the credible autocoding of control software. The main contribution of this chapter is a sound translation of Lyapunov stability proofs into the code domain. We also provided a proof-checking process, which enables one to evaluate the correctness of the stability proof in the floating-point number domain prior to credible autocoding. Although the methods presented in this chapter are conservative since they are based

on interval arithmetic, they can be adapted to use affine arithmetic which will give less conservative estimates of the error bound. Some of the proposed extensions in this chapter are already implemented in the credible autocoding prototype. The current prototype is capable of annotating the original program with the duplicate program, computing the floating-point error bounds and then annotating them on the code. The parts that are yet to be integrated into the prototype include computing with rational arithmetic and checking the inductive condition with the additional formulas expressing floating-point error bounds.

Chapter VI

AN EXAMPLE FROM INDUSTRY

The credible autocoding process (see Figure 35) is tested on a real Full Digital Authority Engine Control (FADEC) system provided by Price Induction. The test system used in this study is a high fidelity model of the DGEN 380 turbofan engine in the form of a virtual test bench [71]. In credible autocoding, the results of stability

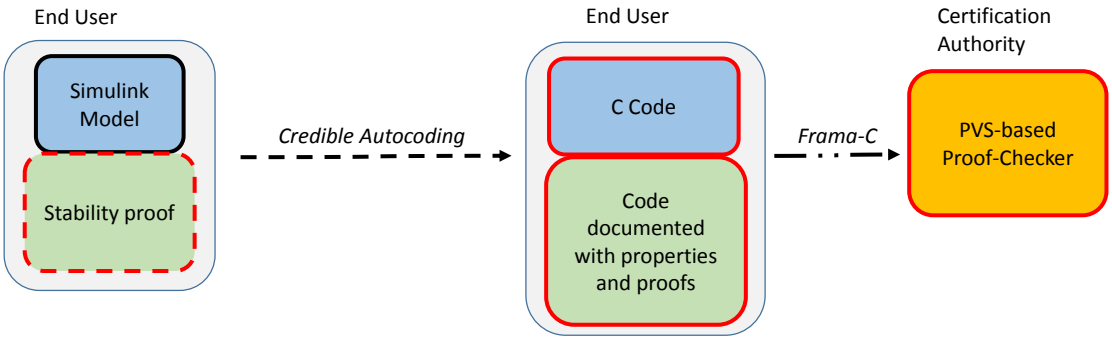


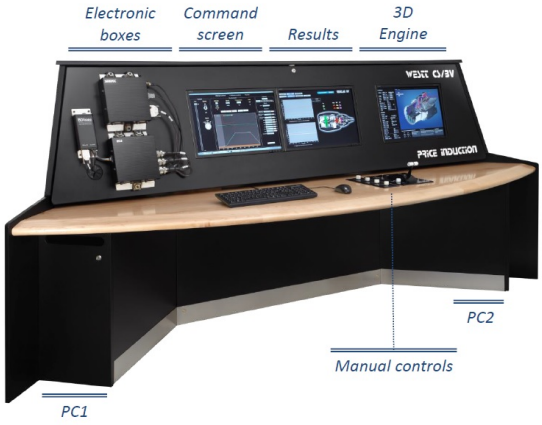
Figure 35: A new software development process with credible autocoding

analysis are translated automatically along with a model of the controller into documented code. Given the documented code, the certification authorities can check its correctness using only a proof-checker. For the example engine controller, an open-loop stability analysis is performed. The results are used to annotate the Simulink model of the FADEC system. From the annotated Simulink model, a documented code is auto-generated using the prototype autocoder Gene-Auto+. This documented code is proof-checked and then compiled into a binary. The binary is tested on Price Induction's virtual test bench for validation.

6.1 DGEN 380 Turbofan Engine

The DGEN 380, shown in Figure 36a, is a two-spool, high bypass ratio (7.6), unmixed flow turbofan engine. Its simple architecture yields up to 560 pounds of thrust in a compact and lightweight format (the engine weighs 175 pounds and is 4 feet long) while maintaining low noise and pollution levels. Beside its optimized performances, the engine innovates with its all-electric system: its starter-generator located directly on the high-pressure shaft, and oil and fuel pumps driven by electric motors are controlled by the Engine Control Unit (ECU), allowing for a really fine and optimized tuning of the DGEN control laws.

(a) DGEN 380 lightweight turbofan engine (©Price Induction) [71]



(b) Price Induction WESTT CS-BV: DGEN 380 turbofan engine virtual test bench (©Price Induction) [71]

6.1.1 Engine Hardware-in-the-Loop Simulator

The WESTT CS-BV, shown in Figure 36b, is a product dedicated to the study of the DGEN 380 turbofan and its control. With the DGEN 380 actual ECU hardware and its model running real-time and generating its sensors analog outputs, the CS-BV constitutes a control Hardware-In-the-Loop (HIL) platform for the testing of engine

control design. The hardwares used in the platform include the actual FADEC from the turbofan engine. The on-board cpu MPC555, which constitutes the core of the ECU, can be easily programmed through the already existing code framework with different control logics and tested in real time with the use of the SIMMOT (software real-time simulation of the engine). All engine outputs are displayed on screen and all data recorded for later performance analysis.

6.2 Application of the Tool-chain on the Price Induction Engine Controller

In this experimental study, the DGEN 380 engine controller model is pre-processed, by hand, into a Simulink model accepted by Gene-Auto+. From the pre-processed Simulink model, a state-space model of the controller is computed symbolically. An open-loop stability analysis of the state-space model of the controller is performed. As expected, it did not yield a common Lyapunov function for the entire range of operating conditions of the engine controller. Instead, a weaker property which holds for a set of operating points is expressed on the Simulink model and translated by Gene-Auto+ into ACSL annotations.

6.3 Constructing the Input Model

The Simulink model of the DGEN 380 FADEC, provided by Price Induction, is displayed in Figure 37. The model contains three top-level subsystems. The "Pilote & Conditions exterieurs" subsystem computes the high and low-pressure turbine speed set-points `NHc` and `NLc`. The set-points are functions of the throttle input `PLA` and other factors such as the temperature and pressure of the turbine.

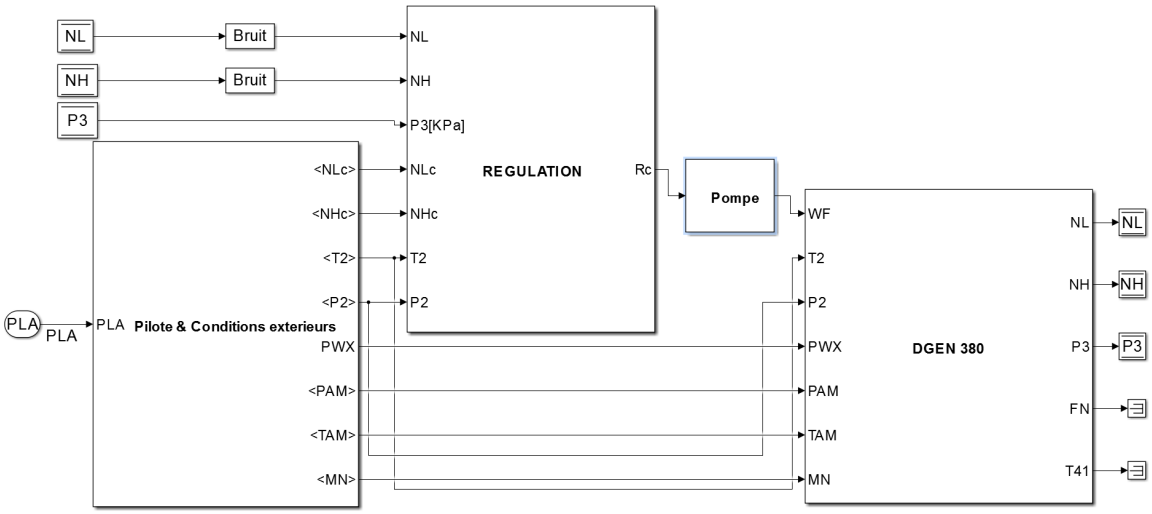


Figure 37: Simulink model of the Price Induction DGEN 380 controller

The "REGULATION" subsystem from Figure 37 contains the engine controller. The controller is designed using a *g*ain scheduling technique. In gain-scheduling, the model of the plant is linearized about points within a range of operating conditions. For the example engine controller, the varying condition or the scheduling parameter is the high-pressure turbine spool speed NH measured as a percentage of a reference maximum spool rate per minute (rpm). The controller gains are designed for each of the linearized system and typically arranged in a look-up table. During runtime, the gains are computed by interpolating on the look-up table.

The subsystem "DGEN 380" is a Matlab model of the DGEN 380 engine. It is not part of the input model to the autocoder since the property of interest is open-loop stability. The original Simulink model from Price Induction contains Matlab functions and compound blocks such as the transfer function block, the look-up table block, and the saturation function block. None of these functions or blocks are accepted by the prototype autocoder. The first step in the credible autocoding of the engine

controller is to pre-process the model by rewriting any blocks not accepted by the prototype with blocks that are. The pre-processing was done manually as there are no automatic tools that can perform this action.

6.3.1 Stability Analysis

The result of the open-loop stability analysis of the controller subsystem in Figure 38 is given in this subsection. The controller subsystem is comprised of two PID controllers

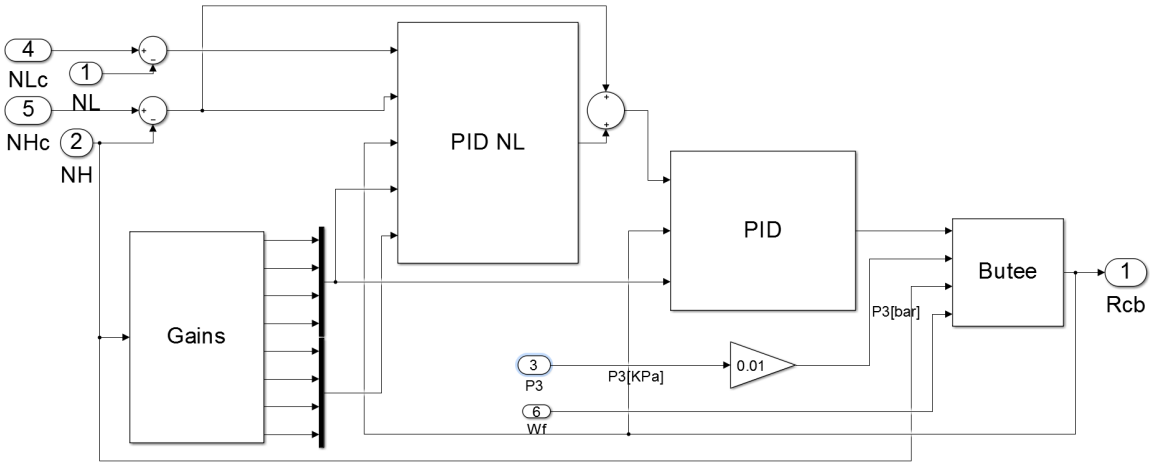


Figure 38: Controller subsystem

and a "Butee" subsystem arranged in a feedback loop. The Butee subsystem is a safety limiter on the output from the PID controllers. The reference inputs to the controller are the high and low pressure turbine spool speed commands NHc and NLc. The sensor inputs to the controller are the high and low-pressure turbine spool speeds NH and NL. In each of the PID subsystems, there are anti-windups for the integrators. For example, the "PID NL" control subsystem in Figure 39 has two anti-windup subsystems. The output $u \in \mathbb{R}$ from the "PID" subsystem in Figure 38 is the input to the Butee subsystem. The Butee has two modes of operations. If the

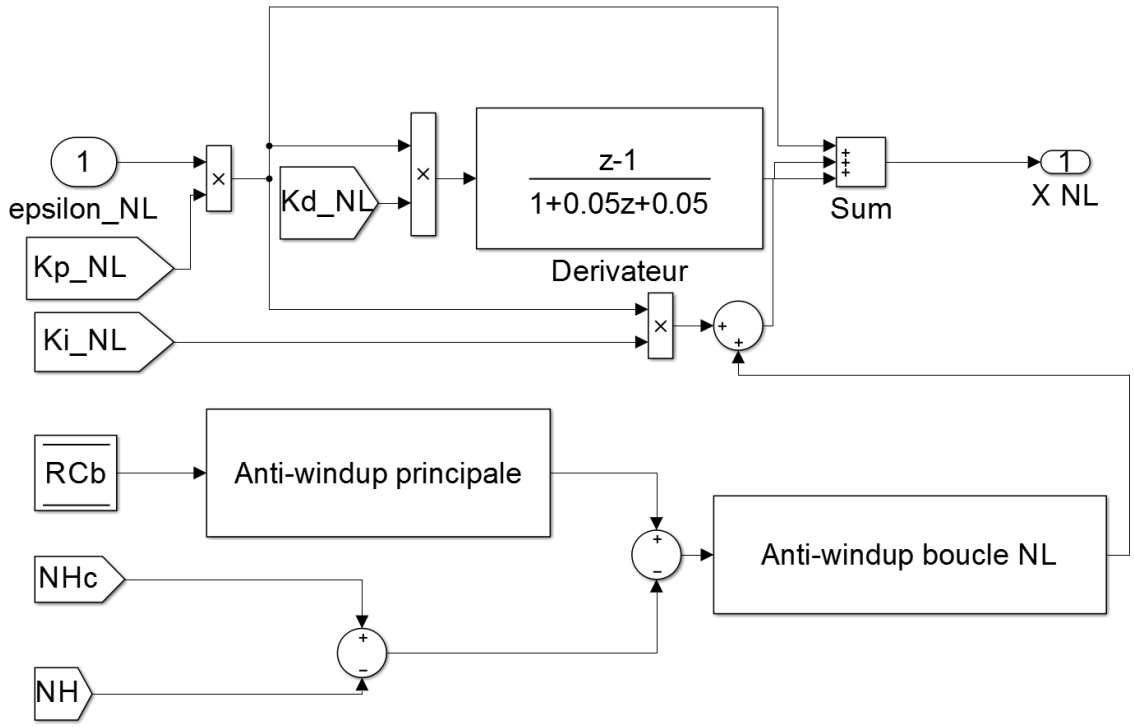


Figure 39: Inside the "PID NL" subsystem block of the controller subsystem

simple mode is switched on, then the Butee behaves like a saturation operator with a range of $[0.07, 0.098]$. In the complex mode, the Butee employs a min/max switching strategy typically used in engine controllers to prevent the violation of performance limits. In this analysis, only the simple mode is considered. This is because even in the complex mode, the output from the Butee is saturated by the same saturation operator executed in the simple mode. The output $\hat{u}$ from the Butee subsystem is fed back to both the PID subsystem and the PID NL subsystem. We can assume that $\hat{u}$ is bounded since it is an output of a saturation operator.

The "Gains" subsystem block takes as input the scheduling parameter NH and returns the gains used by the PID controllers. The controller gains are interpolated using 4 polynomial functions $p_n, n = 1, \ldots, 4$ of degree 6 that map high-pressure turbine spool speed NH to the PID gains K_p, K_i, K_d, and T_d. The coefficients of

the polynomials are provided by Price Induction. Another set of gains $K_{p,NL}$, $K_{i,NL}$, $K_{d,NL}$, and $T_{d,NL}$ are computed from the PID gains using a constant scaling factor. The PID subsystem only use the PID gains while the PID NL subsystem use both sets of gains.

There are total of 11 discrete-time integrators in the model, which makes the dimension of the state-space representation 11. Let $y_d = \begin{bmatrix} NHc & NLc \end{bmatrix}^\intercal$ and $y = \begin{bmatrix} NH & NL \end{bmatrix}^\intercal$. Let $x \in \mathbb{R}^n$, $\hat{y} = y - y_d$, and $\bar{y} = \begin{bmatrix} \hat{y}^\intercal & \hat{u} \end{bmatrix}^\intercal$. The controller subsystem in Figure 38 can be expressed as the discrete-time linear parameter varying state-space system

$$\begin{aligned} x_+ &= A(NH)x + B(NH)\bar{y}, \\ u &= C(NH)x + D(NH)\hat{y}, \end{aligned} \tag{91}$$

where

$$A\,(NH) \in \mathbb{R}^{11\times 11}, \quad B\,(NH) \in \mathbb{R}^{11\times 3}, \quad C\,(NH) \in \mathbb{R}^{1\times 11}, \quad D\,(NH) \in \mathbb{R}^{1\times 2} \tag{92}$$

are matrix rational polynomial functions of NH. The parameter varying matrices in (92) are obtained through a manual analysis of the controller subsystem. First the matrices are expressed as functions of the gain interpolation polynomials $p_i, i = 1, \ldots, 4$ and then as functions of the scheduling parameter NH.

Proposition 6.3.1 Consider the open-loop system in (91) and an operating range $NH \in [NH_{min}, NH_{max}]$. Assume that $\|\bar{y}\| \leq 1$. Let $\mathbf{A} = \left[\underline{A}, \overline{A}\right]$, where $\underline{A}(i,j) \leq (A(NH))(i,j) \leq \overline{A}(i,j)$ for all $NH \in [NH_{min}, NH_{max}]$. Let $\mathbf{B} = \left[\underline{A}, \overline{A}\right]$, where $\underline{B}(i,j) \leq (B(NH))(i,j) \leq \overline{B}(i,j)$ for all $NH \in [NH_{min}, NH_{max}]$. If there exist

$P \succ 0$, $\alpha > 0$, such that

$$\begin{bmatrix} \mathbf{A}^\intercal P\mathbf{A} - P + \alpha P & \mathbf{A}^\intercal P\mathbf{B} \\ \mathbf{B}^\intercal P\mathbf{A} & \mathbf{B}^\intercal P\mathbf{B} - \alpha I \end{bmatrix} \prec 0, \tag{93}$$

then the set $p(P,1)(x)$ is an ellipsoid invariant for (91).

Of the 154 total possible entries in the matrix $\begin{bmatrix} A & B \end{bmatrix}$, 31 of them are parameter varying. The rest are either constant or zero. Computing the existence of $P \succ 0$ such that the linear matrix inequality in (93) holds requires computing a P for 2^{31} corner cases. Solving a feasibility problem with 2^{31} linear matrix inequality constraints is not computationally practical. The problem can be relaxed, as done in [10], to solving a feasibility problem with 32 linear matrix inequality constraints. The relaxation is necessarily conservative [10], with a precise measure of the conservatism by the analytic formula in [58].

The range of operating conditions for the engine is from idle ($NH = 75$) to the maximum thrust ($NH = 106$). In the open-loop case, a single ellipsoid invariant $P(P,1)(x)$ that holds for the entire range of operations is not feasible. For the credible autocoding of this example, a weaker property than stability is showcased. Instead of computing a P such that (93) holds, we look for a common P such that the ellipsoidal set $p(P,1)(x)$ is an invariant for

$$\begin{aligned} x_+ &= A_i + B_i\bar{y} \\ u &= C_i x + D_i\hat{y}, \end{aligned} \tag{94}$$

where $A_i = A(NH_i)$, $B_i = B(NH_i)$, $C_i = C(NH_i)$, $D_i = D(NH_i)$ and $NH_i \in [85, 106]$, $i = 0, \ldots$. The range $NH = 85$ to $NH = 106$ includes operating conditions

normal cruise, maximum recommended cruise ($NH = 95.7$), maximum continuous climb ($NH = 97.7$), and take-off power ($NH = 101$). Figure 40 shows some of the parameter varying entries of the system matrices and the sample points NH_i used in the ensuing analysis.

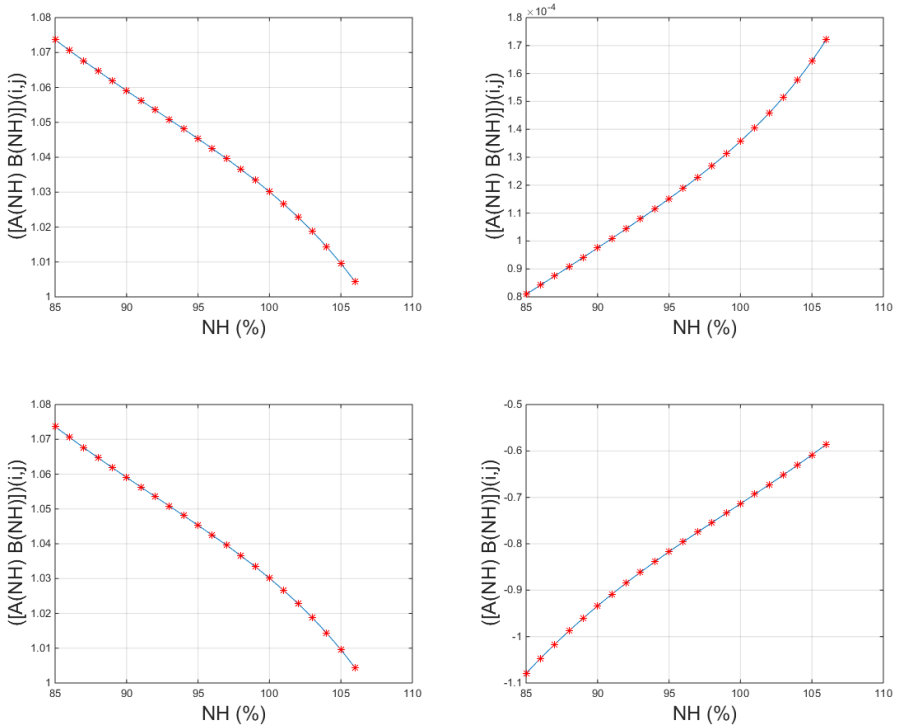


Figure 40: Examples of the parameter varying entries of the engine controller system

Proposition 6.3.2 Assume that $\|\bar{y}\| \leq 1$. If there exists a positive-definite matrix P and a scalar $\xi > 0$ that satisfies

$$\begin{bmatrix} A_i^\intercal P A_i - P + \xi P & A_i^\intercal P B \\ B_i^\intercal P A_i & B_i^\intercal P B_i - \xi I \end{bmatrix} \prec 0, \tag{95}$$

then $p(P, 1)(x)$ is an invariant set with respect to (94).

Remark 10 The property given by Proposition 6.3.2 is weaker than bounded-input

bounded-output stability. However it is still useful in the context of credible autocoding since the idea is to be able to express a high-level property of the system and then prove it on the code. In this case, the property is an inductive loop invariant for the controller program at a sample of different operating conditions.

If the engine controller is re-designed in such way that the controller matrices $\{A(NH), B(NH), C(NH), D(NH)\}$ are computed by interpolating on the matrices $\{A_i, B_i, C_i, D_i\}$, then (95) implies a much stronger property of bounded-input bounded-output stability. In this experimental test, however we did not redesign the controller since the test is about demonstrating the credible autocoding process on the controller. It is not about finding a modified controller that is best suited for credible autocoding. For the FADEC example, solving (95) for 22 equally distributed points in $[85, 106]$ results in the ellipsoidal invariant $p(P, 1)(x)$ where

$$P = 1 \times 10^{-6} \begin{bmatrix} 0.6688 & -0.0274 & 0.0004 & -0.0076 & 0.0008 & -0.0000 & -0.0000 & -0.0000 & -0.3625 & -0.0002 & -0.0005 \\ -0.0274 & 0.0388 & 0.0001 & 0.0095 & 0.0007 & 0.0000 & -0.0000 & -0.0001 & -0.0052 & -0.0000 & -0.0007 \\ 0.0004 & 0.0001 & 0.0000 & 0.0000 & 0.0000 & -0.0000 & -0.0000 & 0.0000 & -0.0002 & -0.0000 & -0.0000 \\ -0.0076 & 0.0095 & 0.0000 & 0.0089 & 0.0007 & 0.0000 & -0.0000 & 0.0001 & -0.0008 & -0.0000 & -0.0005 \\ 0.0008 & 0.0007 & 0.0000 & 0.0007 & 0.0078 & 0.0000 & -0.0000 & -0.0000 & -0.0010 & -0.0000 & -0.0025 \\ -0.0000 & 0.0000 & -0.0000 & 0.0000 & 0.0000 & 0.0115 & -0.0000 & -0.0000 & -0.0000 & 0.0000 & -0.0000 \\ -0.0000 & -0.0000 & -0.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0115 & 0.0000 & 0.0000 & 0.0000 & 0.0000 \\ -0.0000 & -0.0001 & 0.0000 & 0.0001 & -0.0000 & -0.0000 & 0.0000 & 0.0118 & -0.0000 & -0.0000 & 0.0000 \\ -0.3625 & -0.0052 & -0.0002 & -0.0008 & -0.0010 & -0.0000 & 0.0000 & -0.0000 & 0.6511 & 0.0004 & 0.0015 \\ -0.0002 & -0.0000 & -0.0000 & -0.0000 & -0.0000 & 0.0000 & 0.0000 & -0.0000 & 0.0004 & 0.0000 & 0.0000 \\ -0.0005 & -0.0007 & -0.0000 & -0.0005 & -0.0025 & -0.0000 & 0.0000 & 0.0000 & 0.0015 & 0.0000 & 0.0078 \end{bmatrix}. \tag{96}$$

6.3.2 Annotating the Simulink Model

The annotated FADEC Simulink model, which is the input model to the credible autocoding prototype, is displayed in Figure 41. There are three annotation blocks in the model, indicated by the three vamux blocks. The pair of quadratic blocks `stability` and `bounded_input`, which combined, express the invariant property of the engine controller given by Proposition 6.3.2. The block `stability` expresses the inductive ellipsoid invariant $p(P,1)(x)$, in which P is from (96). The block `bounded_input` expresses the assumption of a bound on $\bar{y}$. The third annotation block `sampled_nh`, which is a *constant* block, expresses the set of operating conditions i.e. values of NH in which the analysis result in (96) holds.

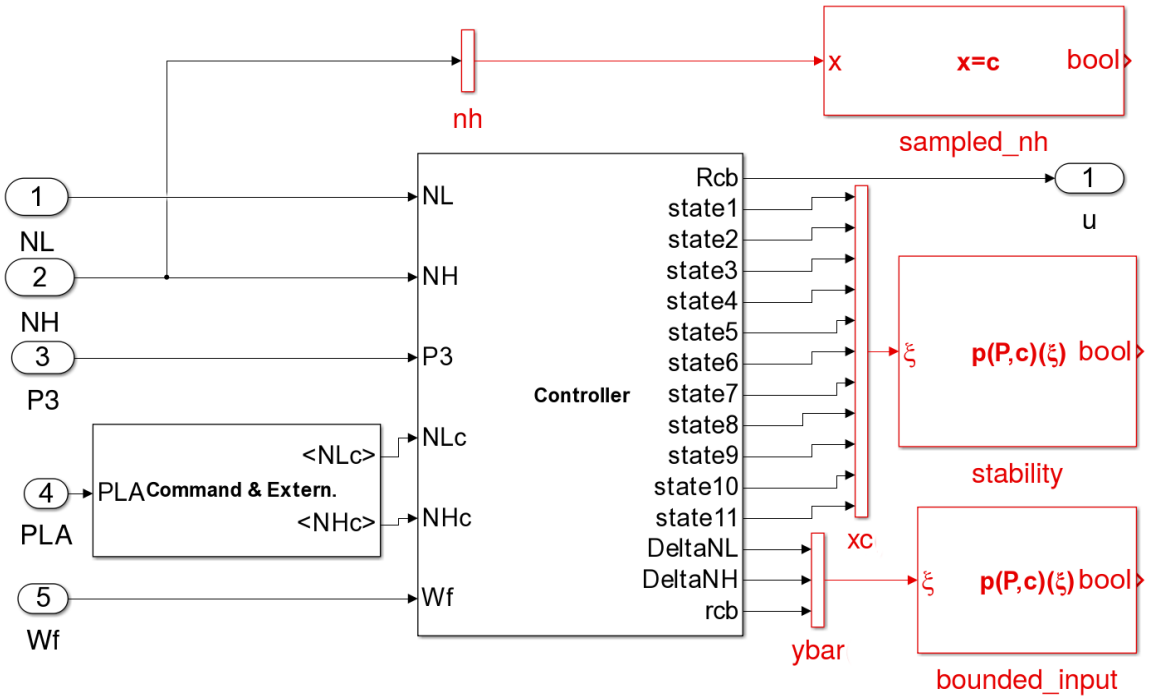


Figure 41: Input FADEC Model to Gene-Auto+

```
1  /*@
2    .
3    .
4    requires in_ellipsoidQ(QMat_0,vect_of_11_scalar(
        ↪ _state_->delay_aw0_memory,
        ↪ _state_->delay_aw1_memory,_state_->delay_E0_memory,
        ↪ _state_->delay_E1_memory,_state_->delay_D0_memory,
        ↪ _state_->delay_D1_memory,_state_->delay_x1_memory,
        ↪ _state_->delay_x2_memory,_state_->delay_aw2_memory,
        ↪ _state_->delay_E2_memory,_state_->delay_D2_memory));
5    requires \valid(_io_) && \valid(_state_);
6    ensures in_ellipsoidQ(QMat_1, vect_of_11_scalar(
        ↪ _state_->delay_aw0_memory,
        ↪ _state_->delay_aw1_memory,_state_->delay_E0_memory,
        ↪ _state_->delay_E1_memory,_state_->delay_D0_memory,
        ↪ _state_->delay_D1_memory,_state_->delay_x1_memory,
        ↪ _state_->delay_x2_memory,_state_->delay_aw2_memory,
        ↪ _state_->delay_E2_memory,_state_->delay_D2_memory));
7  */
8  void pla_compute(t_pla_io *_io_, t_pla_state *_state_) {
9      REAL NL;
10     REAL NH;
11     REAL P3_KPa_;
12     REAL PLA;
13     .
14     .
15     .
16 }
```

Figure 42: The *function contract* expressing the ellipsoid invariant on the update function

6.4 Output Annotated Code

The autocoding generated two functions. The first one is the controller initialization function. The second one is the controller update function. Figure 42 shows a *function contract* on the update function. This function contract express the invariant ellipsoid set computed from the open-loop stability analysis. The *function contract* is duplicated for each behaviors of the code. There are a total of 22 behaviors generated for this example, which results in 22 proofs on the code. Each behavior corresponds to a sampled operating point (see Figure 43). The annotated code took about 40 minutes

```
/*@
  behavior sampled_nh01:
  requires _io_->NH==85.0;
  .
  .
  behavior sampled_nh02:
  requires _io_->NH==86.0;
  .
  .
  behavior sampled_nh22:
  requires _io_->NH==106.0;
  .
  .
*/
.
.
.
void pla_compute(t_pla_io *_io_, t_pla_state *_state_) {
    REAL NL;
    REAL NH;
    REAL P3_KPa_;
    REAL PLA;
    .
    .
```

Figure 43: Multiple behaviors of the controller update function

for the autocoding prototype to produce and the output lines of ACSL annotations exceeds $150,000$.

6.4.1 Verification of the Annotated Code

The linear algebra libraries in PVS, used to check the correctness of the code annotations generated for the Lead/Lag compensator example in Chapter 4, is sufficiently rich to also check the proof annotations generated for the engine example. The automated proof-checking of this example only requires the two custom strategies *AffineEllipsoid* and *S-Procedure*.

6.5 *FADEC in the loop Simulation*

This section presents the simulation result of the verified auto-generated code, shown in the previous section, on the engine test bench running in close-loop with the real FADEC.

Figure 44 shows a snapshot of the WESTT command screen for the case where the original Price Induction binary is executing on the DGEN 380 turbofan engine virtual test bench. The simulation plots displayed in the left half of the snapshot illustrate the evolution of both engine spool speeds that closely follow their reference signals, and also the engine fuel and oil pressure time histories.

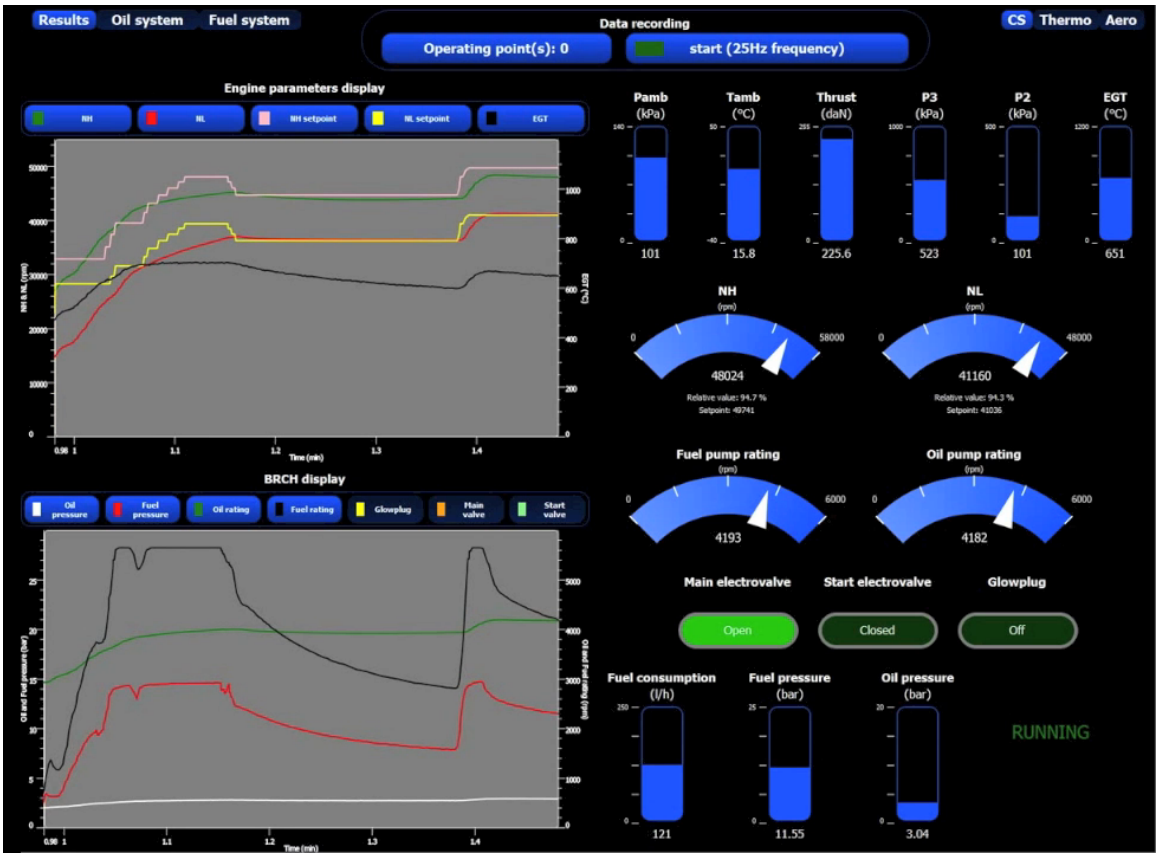


Figure 44: Snapshot of the DGEN 380 turbofan engine virtual test bench running the original code

Figure 45 shows a snapshot of the WESTT command screen for the case where the annotated code generated by Gene-Auto+ is running on the DGEN 380 turbofan engine virtual test bench. The simulation outputs in the left half of the snapshot confirm that the annotated code can run the FADEC-in-loop test bench just like the C code provided by Price Induction.

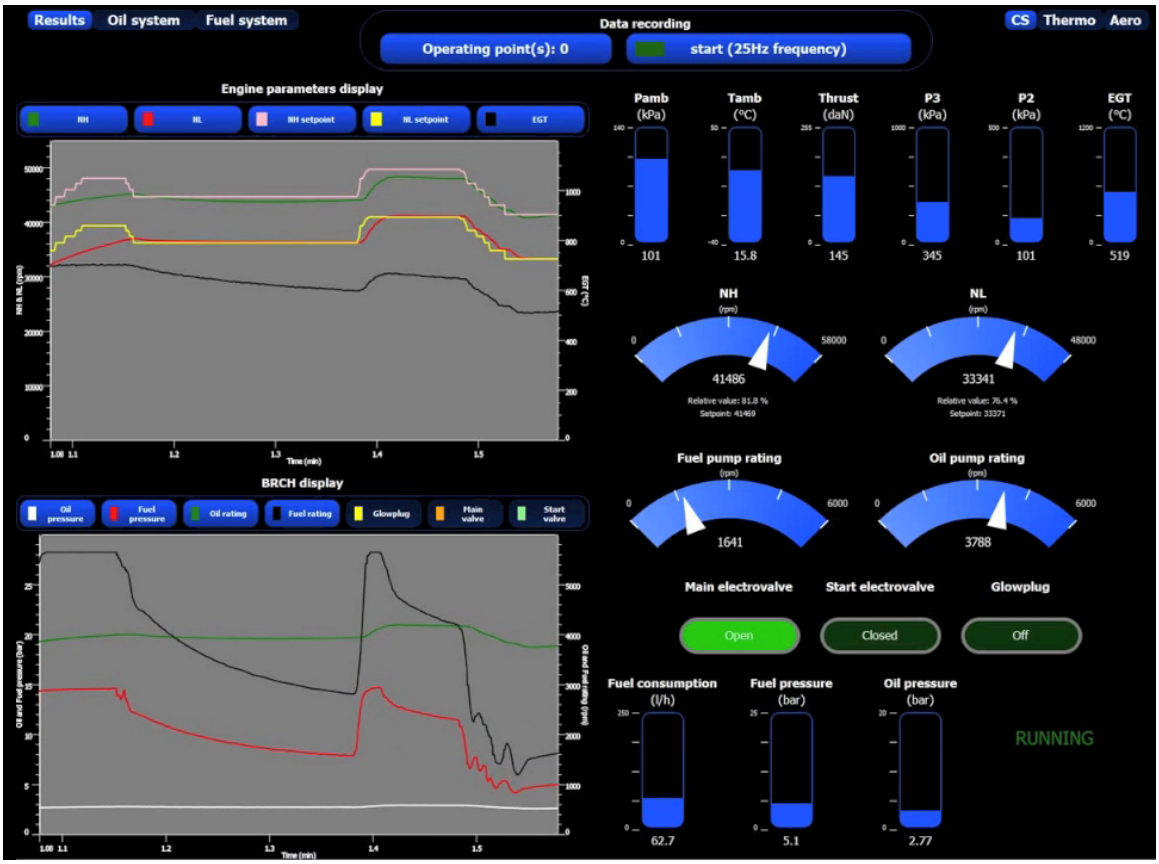


Figure 45: Snapshot of the DGEN 380 turbofan engine virtual test bench running on annotated code produced by Gene-Auto+

The right halves of the snapshots display the visualizations of the engine-related avionics as well as real-time measurements of pressures (P_{amb}, P_2, and P_3), temperatures (T_{amb}, and EGT), speeds (NH, and NL), thrust, fuel pump rating, oil pump rating, fuel consumption, fuel pressure, and oil pressure.

Chapter VII

CONCLUSION AND FUTURE DIRECTIONS

This research concludes with a proof of concept prototype that is capable of producing control software documented with verifiable properties of open-loop and closed-loop stability. The prototype has also been further extended to handle observer-based fault-detection systems as well as gain-scheduled control systems. Algorithms have been proposed and partially implemented in the prototype to account for floating-point errors. Finally, the credible autocoding approach was validated on a FADEC model provided by the industry.

There are several new directions of future research. The first direction is an extension of the framework to the domain of real-time, convex optimization-based control systems, about which we have an initial work [98]. For convex optimization algorithms, such as a primal-dual interior-point algorithm, the invariant is provided by a monotonically decreasing duality gap function. While this property exists for many interior-point algorithms, there are two remaining challenges in credible autocoding of convex optimization algorithms. First, the floating-point errors in interior-point algorithms are more difficult to analyze. Unlike in control algorithms, an interior-point algorithm computes a Hessian which requires inverting a matrix. The existing literature [95, 104, 103] only provides estimates of the floating-point error using the order

notation. Another difficulty lies in the fact that for semi-definite programming algorithms, the lack of strict complementarity makes precise floating-point error analysis problematic [100]. Second, the predicted convergence rates of interior-point algorithms are often conservative compared to their actual performances in practice. For this reason, another challenge is to find alternative proofs of convergence for interior-point methods, perhaps using a Lyapunov-based approach, that might yield less conservative predictions on their performances. This idea is inspired by a recent work of Lessard and Packard, where the IQC framework was used to compute Lyapunov-type certificates of performances for gradient-based optimization algorithms [52].

The quadratic invariant described in this thesis generalizes to the sum of squares (SOS) polynomial type invariant. Hence another direction of research is to extend credible autocoding to cover the much larger class of polynomial systems by automating the translation and verification of SOS type invariants. Along the same line of research, credible autocoding could potentially extend to: robustness properties such as vector margin; other useful measure of performance such as rise times and settling times; the properties of fault-detection systems other than the observer-based ones; and the properties of probabilistic systems. Furthermore, the dynamics of real-time scheduling and the effects of time delays in the system need to be eventually explored and accounted for in credible autocoding.

The current prototype only served as a proof of concept hence it was built in an ad hoc manner. For the purpose of qualification and portability, the author suggests further research into generalizing the library of control semantics and their translations using a model-driven engineering approach.

REFERENCES

[1] "Geneauto metamodel with verification annotations documentation." `http://block-library.enseeiht.fr/html/Progress/geneautoAnnot.html`.

[2] Abdulla, P., Deneux, J., Stalmarck, G., Agren, H., and Akerlund, O., "Designing safe, reliable systems using scade," in *Leveraging Applications of Formal Methods* (Margaria, T. and Steffen, B., eds.), vol. 4313 of *Lecture Notes in Computer Science*, pp. 115–129, Springer Berlin Heidelberg, 2006.

[3] Agrawal, A., Simon, G., and Karsai, G., "Semantic translation of Simulink/Stateflow models to hybrid automata using graph transformations," *Electronic Notes in Theoretical Computer Science*, vol. 109, pp. 43 – 56, 2004. Proceedings of the Workshop on Graph Transformation and Visual Modelling Techniques (GT-VMT 2004).

[4] Alur, R., Dang, T., and Ivančić, F., "Predicate abstraction for reachability analysis of hybrid systems," *ACM Trans. Embed. Comput. Syst.*, vol. 5, pp. 152–199, Feb. 2006.

[5] Arthan, R., Martin, U., and Oliva, P., "A hoare logic for linear systems," *Formal Aspects of Computing*, vol. 25, no. 3, pp. 345–363, 2013.

[6] Bagnara, R., Hill, P., Ricci, E., and Zaffanella, E., "Precise widening operators for convex polyhedra," in *Static Analysis* (Cousot, R., ed.), vol. 2694 of *Lecture Notes in Computer Science*, pp. 337–354, Springer Berlin Heidelberg, 2003.

[7] Baudin, P., Filliâtre, J.-C., Marché, C., Monate, B., Moy, Y., and Prevosto, V., "Acsl: Ansi/iso c specification language. version 1.7.." `http://frama-c.com/download/acsl.pdf`.

[8] Baudin, P., Filliatre, J.-C., Marche, C., Monate, B., Moy, Y., and Prevosto, V., *ACSL: ANSI/ISO C Specification Language*, 2008. http://frama-c.cea.fr/acsl.html.

[9] Bekenstein, J. D., "Black holes and entropy," *Physical Review D*, vol. 7, no. 8, p. 2333, 1973.

[10] Ben-Tal, A. and Nemirovski, A., "On tractable approximations of uncertain linear matrix inequalities affected by interval uncertainty," *SIAM Journal on Optimization*, vol. 12, no. 3, pp. 811–833, 2002.

[11] Berry, G., Gonthier, G., Gonthier, A. B. G., and Laltte, P. S., "The esterel synchronous programming language: Design, semantics, implementation," 1992.

[12] Bordin, M., Naks, T., Toom, A., and Pantel, M., "Compilation of heterogeneous models: Motivations and challenges," in *ERTS*, (http://www.sia.fr), Société des Ingénieurs de l'Automobile, 2012.

[13] Bouissou, O. and Chapoutot, A., "An operational semantics for simulink's simulation engine," in *Proceedings of the 13th ACM SIGPLAN/SIGBED International Conference on Languages, Compilers, Tools and Theory for Embedded Systems*, LCTES '12, (New York, NY, USA), pp. 129–138, ACM, 2012.

[14] Boyd, S., El Ghaoui, L., Feron, E., and Balakrishnan, V., *Linear Matrix Inequalities in System and Control Theory*, vol. 15 of *Studies in Applied Mathematics*. Philadelphia, PA: SIAM, June 1994.

[15] Boyd, S. and Vandenberghe, L., *Convex Optimization*. New York, NY, USA: Cambridge University Press, 2004.

[16] Brosgol, B. and Comar, C., "Do-178c: A new standard for software safety certification," tech. rep., DTIC Document, 2010.

[17] Caspi, P., Curic, A., Maignan, A., Sofronis, C., and Tripakis, S., "Translating discrete-time Simulink to Lustre," in *Embedded Software* (Alur, R. and Lee, I., eds.), vol. 2855 of *Lecture Notes in Computer Science*, pp. 84–99, Springer Berlin / Heidelberg, 2003. 10.1007/978-3-540-45212-67.

[18] Ceruzzi, P., *Beyond the Limits: Flight Enters the Computer Age*. Books; 13, MIT Press, 1989.

[19] Clarke, E. M., Emerson, E. A., and Sistla, A. P., "Automatic verification of finite-state concurrent systems using temporal logic specifications," *ACM Trans. Program. Lang. Syst.*, vol. 8, pp. 244–263, April 1986.

[20] Clarke, E. M., "The birth of model checking," in *25 Years of Model Checking*, pp. 1–26, Springer, 2008.

[21] Clarke, E. M., Grumberg, O., Hiraishi, H., Jha, S., Long, D. E., McMillan, K. L., and Ness, L. A., "Verification of the futurebus+ cache coherence protocol," *Formal Methods in System Design*, vol. 6, no. 2, pp. 217–232, 1995.

[22] Cofer, D. D., Whalen, M. W., and Miller, S. P., "Model-checking of safety-critical software for avionics.," *ERCIM News*, vol. 2008, no. 75, 2008.

[23] 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.

[24] Cousot, P., Cousot, R., Feret, J., Mauborgne, L., Miné, A., Monniaux, D., and Rival, X., "The astreé analyzer," in *ESOP*, pp. 21–30, 2005.

[25] Dabney, J. B. and Harman, T. L., *Mastering simulink*. Pearson, 2004.

[26] de Graaff, A., "Aviation safety, an introduction," *Air & Space Europe*, vol. 3, no. 3–4, pp. 203 – 205, 2001.

[27] 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.

[28] Denney, E., "Certifying auto-generated flight code." `http://shemesh.larc.nasa.gov/LFM2008/slides/Session3/Denney.ppt`, 2008.

[29] Dieumegard, A., *Formal Guarantees for Safety Critical Code Generation: the Case of Highly Variable Languages*. PhD thesis, INP Toulouse, 2015.

[30] Dijkstra, E. W., "Guarded commands, nondeterminacy and formal derivation of programs," *Communications of the ACM*, vol. 18, no. 8, pp. 453–457, 1975.

[31] Dijkstra, E. W., *A Discipline of Programming*. Upper Saddle River, NJ, USA: Prentice Hall PTR, 1st ed., 1997.

[32] Doyle, J., "Analysis of feedback systems with structured uncertainties," *Control Theory and Applications, IEE Proceedings D*, vol. 129, pp. 242 –250, november 1982.

[33] Feret, J., "Static analysis of digital filters," in *European Symposium on Programming (ESOP'04)*, no. 2986 in LNCS, Springer-Verlag, 2004.

[34] Féron, É., "From control systems to control software," *Control Systems Magazine, IEEE*, vol. 30, no. 6, pp. 50–71, 2010.

[35] Floyd, R. W., "Assigning meanings to programs," *Proceedings of Symposium on Applied Mathematics*, vol. 19, pp. 19–32, 1967.

[36] Goldberg, D., "What every computer scientist should know about floating point arithmetic," *ACM Computing Surveys*, vol. 23, no. 1, pp. 5–48, 1991.

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

[38] Goubault, E. and Putot, S., "A zonotopic framework for functional abstractions," *CoRR*, vol. abs/0910.1763, 2009.

[39] Haddad, W. M. and Chellaboina, V., *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2008.

[40] 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.

[41] Higham, N., *Accuracy and Stability of Numerical Algorithms*. Philadelphia: SIAM Publications, 2nd ed., 2002.

[42] Hoare, C. A. R., "An axiomatic basis for computer programming," *Commun. ACM*, vol. 12, pp. 576–580, October 1969.

[43] Hueschen, R. M., *Development of the Transport Class Model (TCM) aircraft simulation from a sub-scale Generic Transport Model (GTM) simulation*. National Aeronautics and Space Administration, Langley Research Center, 2011.

[44] IEEE, "IEEE standard for floating-point arithmetic," *IEEE Std 754-2008*, pp. 1 –58, 29 2008.

[45] Izerrouken, N., Pantel, M., Thirioux, X., and Ssi Yan Kai, O., "Integrated formal approach for qualified critical embedded code generator," in *Formal Methods for Industrial Critical Systems*, vol. 5825 of *Lecture Notes in Computer Science*, pp. 199–201, Springer Berlin Heidelberg, 2009.

[46] Jobredeaux, R., *Formal Verification of Control Software*. PhD thesis, Georgia Institute of Technology, 2015.

[47] Jobredeaux, R., Wang, T. E., and Feron, E. M., "Autocoding control software with proofs I: Annotation translation," in *Digital Avionics Systems Conference (DASC), 2011 IEEE/AIAA 30th*, pp. 1 –19, Oct. 2011.

[48] Kalman, R. E., "Lyapunov functions for the problem of Lur'e in automatic control," *Proceedings of the National Academy of Sciences of the United States of America*, vol. 49, no. 2, p. 201, 1963.

[49] Kemin Zhou, J. D. and Glover, K., *Robust and Optimal Control*. Upper Saddle River, New Jersey: Prentice Hall, 1996.

[50] Kredel, H., "The java algebra system (jas)," tech. rep., Technical report (since 2000), http://krum. rz. uni-mannheim. de/jas, 2000.

[51] Kurzhanski, A. B. and Valyi, I., *Ellipsoidal calculus for estimation and control*. Systems & control, Birkhauser, 1997.

[52] Lessard, L., Recht, B., and Packard, A., "Analysis and design of optimization algorithms via integral quadratic constraints," *arXiv preprint arXiv:1408.3595*, 2014.

[53] Lur'e, A. I., *Some Nonlinear Problems of Automatic Control Theory*. London H.M. Stationery Office, 1951.

[54] Lyapunov, A. M., *The General Problem of Relative Stability*. PhD thesis, Moscow University, 1892.

[55] Maisonneuve, V., *Static Analysis of Control-Command Systems, Floating-point and Integer Invariants*. PhD thesis, Ecole des Mines, 2015.

[56] Martin, R., Peters, G., and Wilkinson, J., "Symmetric decomposition of a positive definite matrix," *Numerische Mathematik*, vol. 7, no. 5, pp. 362–383, 1965.

[57] Mccarthy, J., "A basis for a mathematical theory of computation," in *Computer Programming and Formal Systems*, pp. 33–70, North-Holland, 1963.

[58] McCullough, S. A., Klep, I., Schweighofer, M., and Helton, J. W., "Dilations, linear matrix inequalities, the matrix cube problem and beta distributions," 2015.

[59] Megretski, A. and Rantzer, A., "System analysis via integral quadratic constraints," *Automatic Control, IEEE Transactions on*, vol. 42, no. 6, pp. 819–830, 1997.

[60] Miller, J. C. and Maloney, C. J., "Systematic mistake analysis of digital computer programs," *Commun. ACM*, vol. 6, pp. 58–63, Feb. 1963.

[61] Miller, S., Anderson, E., Wagner, L., Whalen, M., and Heimdahl, M., "Formal verification of flight critical software," in *AIAA Guidance, Navigation, and Control Conference*, 2005.

[62] Monniaux, D., "The pitfalls of verifying floating-point computations," *ACM Transactions on Programming Languages and Systems (TOPLAS)*, vol. 30, no. 3, p. 12, 2008.

[63] Moore, R. E., *Methods and Applications of Interval Analysis*. Philadelphia SIAM, 1979.

[64] 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.

[65] Naur, P., "Proof of algorithms by general snapshots," *BIT Numerical Mathematics*, vol. 6, no. 4, pp. 310–316, 1966.

[66] NSA, "Correct by construction: Advanced software engineering," in *High Confidence Software and Systems*, vol. 19, pp. 22–31, National Security Agency, 2011.

[67] Nuseibeh, B., "Ariane 5: Who dunnit?," *Software, IEEE*, vol. 14, pp. 15–16, May 1997.

[68] Owre, S., Rushby, J., and Shankar, N., "PVS: A prototype verification system," in *Automated Deduction* (Kapur, D., ed.), vol. 607 of *Lecture Notes in Computer Science*, pp. 748–752, Springer Berlin / Heidelberg, 1992.

[69] Parrilo, P. A., *Structured semidefinite programs and semialgebraic geometry methods in robustness and optimization*. PhD thesis, California Institute of Technology, 2000.

[70] Pressman, R. S., *Software Engineering: A Practitioner's Approach*. McGraw-Hill Higher Education, 5th ed., 2001.

[71] Price Induction, "DGEN 380 turbofan engine," 2013. URL: `http://www.price-induction.com/en`.

[72] Queille, J.-P. and Sifakis, J., "Specification and verification of concurrent systems in CESAR," in *Proceedings of the 5th Colloquium on International Symposium on Programming*, (London, UK, UK), pp. 337–351, Springer-Verlag, 1982.

[73] Rice, H. G., "Classes of recursively enumerable sets and their decision problems," *Transactions of the American Mathematical Society*, pp. 358–366, 1953.

[74] Rinard, M., "Credible compilation," tech. rep., In Proceedings of CC 2001: International Conference on Compiler Construction, 1999.

[75] Roozbehani, M., Megretski, A., and Feron, E., "Optimization of Lyapunov invariants in verification of software systems," *IEEE Trans. Automat. Contr.*, vol. 58, no. 3, pp. 696–711, 2013.

[76] Roux, P., "Formal proofs of rounding error bounds with application to an automatic positive definiteness check." https://hal.archives-ouvertes.fr/hal-01091189/document, 2015.

[77] 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.

[78] RTCA, S. C., "DO-178B, software considerations in airborne systems and equipment certification," 1992.

[79] RTCA, S. C., "DO-178C, software considerations in airborne systems and equipment certification," 2011.

[80] RTCA, S. C., "DO-333 formal methods supplement to DO-178C and DO-278A," tech. rep., Radio Technical Commission for Aeronautics, Dec 2011.

[81] Rump, S., "INTLAB - INTerval LABoratory," in *Developments in Reliable Computing* (Csendes, T., ed.), pp. 77–104, Dordrecht: Kluwer Academic Publishers, 1999. `http://www.ti3.tu-harburg.de/rump/`.

[82] Scaife, N., Sofronis, C., Caspi, P., Tripakis, S., and Maraninchi, F., "Defining and translating a "safe" subset of Simulink/Stateflow into Lustre," in *Proceedings of the 4th ACM international conference on Embedded software*, EMSOFT '04, (New York, NY, USA), pp. 259–268, ACM, 2004.

[83] Scott, M. L., *Programming Language Pragmatics*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2000.

[84] Selic, B., "The pragmatics of model-driven development," *IEEE Softw.*, vol. 20, pp. 19–25, Sep. 2003.

[85] Shaw, M., "Toward higher-level abstractions for software systems," *Data Knowl. Eng.*, vol. 5, pp. 119–128, Jul. 1990.

[86] Skeel, R., "Roundoff error and the patriot missile," *SIAM News*, vol. 25, no. 4, p. 11, 1992.

[87] Sterbenz, P., *Floating-point Computation*. Prentice-Hall, 1974.

[88] 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.

[89] Tiwari, A., "Formal semantics and analysis methods for Simulink Stateflow models," tech. rep., SRI International, 2002. `http://www.csl.sri.com/-~tiwari/stateflow.html`.

[90] Tiwari, A. and Khanna, G., "Series of abstractions for hybrid automata," in *Hybrid Systems: Computation and Control HSCC* (Tomlin, C. J. and Greenstreet, M. R., eds.), vol. 2289 of *LNCS*, pp. 465–478, Springer, Mar. 2002.

[91] Toom, A., Izerrouken, N., Naks, T., Pantel, M., and Ssi-Yan-Kai, O., "Towards reliable code generation with an open tool: Evolutions of the Gene-Auto toolset," in *ERTS*, (http://www.sia.fr), Société des Ingénieurs de l'Automobile, 2010.

[92] Toom, A., Naks, T., Pantel, M., Gandriau, M., and Wati, I., "Gene-Auto - an automatic code generator for a safe subset of Simulink-Stateflow and Scicos," in *ERTS*, (http://www.sia.fr), Société des Ingénieurs de l'Automobile, 2008.

[93] Turing, A. M., "Checking a large routine," in *Report of a Conference on High Speed Automatic Calculating Machines*, pp. 67–69, 1949.

[94] Venet, A., "The gauge domain: Scalable analysis of linear inequality invariants," in *CAV*, pp. 139–154, 2012.

[95] Vera, J. R., "On the complexity of linear programming under finite precision arithmetic," *Mathematical Programming*, vol. 80, no. 1, pp. 91–123, 1998.

[96] Wang, T., Jobredeaux, R., Herencia-Zapana, H., Garoche, P.-L., Dieumegard, A., Feron, E., and Pantel, M., "From design to implementation: an automated, credible autocoding chain for control systems," in *Advances in Control System Technology for Aerospace Applications* (Feron, E., ed.), Springer, 2015. IN PRESS.

[97] Wang, T., Jobredeaux, R., Pakmher, M., Vivies, M., Feron, E., and Boidot, E., "Credible autocoding and verification of a gas turbine engine fadec," in *Digital Avionics Systems Conference (DASC), 2014 IEEE/AIAA 33rd*, pp. 1–20, IEEE, 2014.

[98] Wang, T., Jobredeaux, R., Pantel, M., Garoche, P.-L., Feron, E., and Henrion, D., "Credible autocoding of convex optimization algorithms," *Journal of Optimization and Engineering*, 2015, Accepted.

[99] Wang, T. E., Ashari, A. E., Jobredeaux, R., and Feron, E., "Credible autocoding of fault-detection systems," in *Proceedings of the 2014 American Control Conference*, (Portland), 2014.

[100] Wei, H., *Numerical Stability in Linear Programming and Semidefinite Programming*. PhD thesis, University of Waterloo, 2006.

[101] Whalen, M. W., Murugesan, A., Rayadurgam, S., and Heimdahl, M. P., "Structuring simulink models for verification and reuse," in *Proceedings of the 6th International Workshop on Modeling in Software Engineering*, pp. 19–24, ACM, 2014.

[102] Woodcock, J., Larsen, P. G., Bicarregui, J., and Fitzgerald, J., "Formal methods: Practice and experience," *ACM Comput. Surv.*, vol. 41, pp. 19:1–19:36, Oct 2009.

[103] Wright, S. J., "Superlinear convergence of a stabilized sqp method to a degenerate solution," *Computational Optimization and Applications*, vol. 11, no. 3, pp. 253–275, 1998.

[104] Wright, S. J., "Effects of finite-precision arithmetic on interior-point methods for nonlinear programming," *SIAM Journal on Optimization*, vol. 12, no. 1, pp. 36–78, 2001.

[105] YAKUBOVICH, V. A., "The solution of certain matrix inequalities in automatic control theory," in *Soviet Math. Dokl*, vol. 3, pp. 620–623, 1962.

[106] YAKUBOVICH, V. A., "The S-procedure in nonlinear control theory," *Vestnik Leningrad University Math*, vol. 4, pp. 73–93, 1971.

VITA

Timothy E. Wang was born in the ancient city of Nanjing, China. The city was the capital of six dynasties since 500 B.C and during the short tenure by the Nationalists who fled to Taiwan in 1949. He move to the Northern Jersey suburb of New York as a kid and grew up there until a sudden move to the deep south where he became a "damn yankee". He graduated high school from Athens, Georgia and then became an aerospace engineering undergraduate at Georgia Tech. When he came back to Georgia Tech for graduate school, he slowly and deliberately shifted his focus to computer science. He has worked at NASA on credible autocoding as well as visiting Rockwell Collins Control Technology for a summer. He currently holds a Ph.D., a M.S. as well as a B.S. in Aerospace Engineering from Georgia Institute of Technology.