Numerical Computations and Proofs: from Proof-Assistants to Aerospace Applications

Pierre Roux

October 22, 2025

Contents

Chapter 1

Introduction

Since about three decades, modern commercial aircraft use digital flight commands, that is a computer program repeatedly reads orders from the pilots, interprets them and send commands to the various govern actuators on the wings, horizontal and vertical stabilizers, acting on the flow of air around the plane, hence on its trajectory. This enables multiple things, among which:

- Improve both pilot and passenger comfort. For instance a recent commercial airplane used to make some passengers seated at the back dizzy, due to uncomfortable movements. An update of the flight command software was enough to fix the issue.
- Enable different aircraft to feel similar for pilots. This optimizes pilot training and management for the airline companies, that can thus more easily operate similar planes, for instance of different sizes.
- Improve safety, by preventing dangerous attitudes / efforts that might be accidentally required by pilots. This also enable pilots to require prompt actions, for instance in case of emergency, without fear of damaging the plane.
- Last but not least, as explained on Figure 1.1, planes are designed to be inherently stable. This requires a downward force at the tail, hence a larger wing to compensate with a larger upward force, both of which create drag, hence an additional fuel burn. An unstable plane would be very hard to master for a human pilot whereas it can be handled by a computer.[1]

Obviously, seating between pilots and the plane, those digital flight commands are critical systems, as their failure could lead to dramatic consequences. We thus have an interest to perform formal or mechanized proofs of correctness for such systems. Such numerical proofs can be tedious, so it can additionally be good to obtain them as automatically as possible. For instance, an important mechanism of digital flight commands is the flight-envelope protection. Among other things, this prevents the pilots from putting the plane in a position in which it would stall.[2]. Considering the flight command program, a model of the plane and hypotheses about its environment, one could like to prove that stall conditions are indeed unreachable.

In practice, we are not yet able to perform such proofs. Actual flight command programs are far from trivial. They are based on basic linear controllers but there are usually not a single controller but multiple one for different flight points (altitude, speed, mass of the plane and position of its center of gravity)[3] which are then interpolated. On top of that, multiple discrete behaviors are added, for instance to implement the flight envelope protection, yielding a complex dynamical system. As a starting point, we nevertheless studied the search for proofs for simple systems, that is linear systems with a few nonlinear additive behaviors.

Thus, during my PhD thesis [Rou13], advised by Pierre-Loïc Garoche and under the direction of Virginie Wiels, I focused on inference of quadratic invariants (geometrically speaking, ellipsoids) for mostly-linear programs. Quadratic invariants are well suited to bound the behaviors of the linear cores of the programs studied. The invariants were automatically inferred combining convex numerical solvers, called SDP solvers, and static program-analysis methods, called policy iteration. This, as well as later work in this direction, is summarized in Chapter 2.

I later extended that line of research to inference of polynomial invariants for programs with polynomial expressions. After a first attempt using Bernstein polynomials [RG13a] at the end of my PhD,[4], I then looked for sum of squares polynomials, still using SDP solvers. This is presented in Section 3.2.1,

[1]Note that modern combat aircraft are designed this way to be unstable, but it's more a matter of maneuverability than fuel efficiency.

[2]That is, experiencing a large loss of lift on the wings, usually due to a bad combination of low speed and high angle of incidence.

[3]Each of these variables changes during a flight, as the plane lightens, burning its fuel.

[4]The current document won't discuss further this work with Bernstein polynomials.

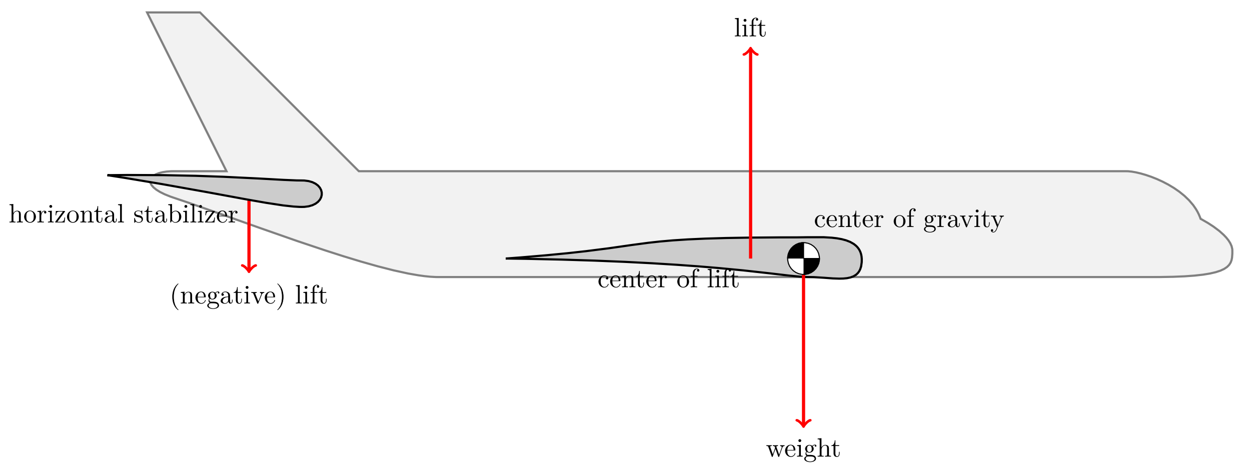


Figure 1.1: Planes are usually built to be stable. The weight is applied at the center of gravity which is designed to be at a small distance in front of the center of lift. The pitch-down torque thus created is compensated by a downward "lift", produced by the horizontal stabilizer at the tail of the plane. When the plane goes down, it accelerates and the lift increases, which creates a pitch-up torque, making the plane horizontal again. Conversely, when the plane goes up, the lift decreases, creating a pitch-down torque that makes the plane horizontal again.

For the sake of efficiency, all these invariant-inference methods, using SDP solvers, are fundamentally based on approximate computations performed with floating-point arithmetic. Section 3.1 demonstrates how easy it is to derive incorrect results, without much warning. I thus investigated some effort in rigorous verification methods to derive strong a-posteriori guarantees of correctness at small overhead costs, as described in Section 3.2.

Whereas such rigorous verifications were first performed in the prototype program analyzer I developed for my PhD, I was later interested in further developing them and making them available in wider contexts. In particular I investigated their use in SMT (Satisfiability Modulo Theories) solvers. As their name indicates, these solvers combine multiple theory solvers through a boolean SAT solver (solving boolean SATisfiability problems) to attempt to validate (or refute with counterexamples) logical formulas. They are used in a wide range of applications, including program verification. Working in collaboration with Sylvain Conchon and Mohamed Iguernlala, we got their Alt-Ergo SMT solver to solve numerical problems out of reach for the – at least then – state of the art solvers [RIC18]. This work is presented in Section 3.3.

Not long after, in collaboration with Érik Martin-Dorel, assistant professor at the nearby Paul Sabatier University in Toulouse, we developed the ValidSDP library, making these rigorous verification methods for polynomial inequalities available in the Coq proof assistant. A proof assistant is a software that helps its user build mechanized proofs, that is actually build the logical objects corresponding to mathematical proofs. This enables both a very high level of confidence in the proofs (since they are verified by a – somewhat – small piece of software (the kernel of the proof assistant) that only relies on some basic logic) and manipulation of the proofs like computer programs. For instance, understanding how and where an hypothesis is used (or not) in a proof, is just a matter of commenting out that hypothesis and recompiling the proof. Note that later in this document, as well as in the literature, the terms "formal proof" are often used to refer to those mechanized proofs. Thus, the ValidSDP implementation provides both an easy access to the polynomial verification methods in virtually any proof that can be performed with Coq, as well as a mechanized proof of the – far from trivial – internals of the method. All this is detailed in Section 3.4.

As a side effect, the previous work on ValidSDP sparked interest for more efficient handling of floating-point arithmetic in Coq. Working again with Érik, we coadvised the L3 internship of Guillaume Bertholon (summer 2018) which eventually led to the addition of the primitive floats feature [BMDR19] in Coq (end of 2019). To combine soundness and efficiency, this raised a number of interesting questions, as presented in Chapter 4. Some roots of that work and my interest in formal proofs about floating-point arithmetic can in some sense be traced back to a short six-months postdoctoral work, coadvised by Guillaume Melquiond and Sylvie Boldo at the LRI lab in 2014,[5] during which I studied conditions for innocuous double rounding on arithmetic operators, with formalized proofs in Coq [Rou14].

Besides that interest in formal proofs of numerical properties, after getting a permanent position at ONERA, I got the opportunity to start a still-ongoing fruitful collaboration with my colleague Marc Boyer.

[5] I basically had some PhD funding remaining, wanted some serious experience with Coq and asked them if they got a subject to feed me.

Marc is our specialist in embedded networks. He particularly developed an expertise in network calculus, a method used to certify embedded networks in commercial aircraft during the last two decades. Indeed, modern planes are packed with systems that need to communicate all kind of data between them. Some of these data require some guarantee that they won't be lost during their journey on the network or that they will arrive within a strict time bound. A main line of our common work centered around a Coq formalization of important building bricks of network calculus. In particular, we coadvised the PhD thesis of Lucien Rakotomalala. This work is detailed in Chapter 5.

Finally, to support all these work, I conduct a number of more technical activities. For multiple reasons, these activities are not amenable to publications, although they can represent a nonnegligible part of my working time. The Chapter 6 gives some examples of those activities. I also value teaching and transmission of knowledge and despite my position not mandating it, I try to keep a small[6] amount of teaching activities over the years. This however remains outside the scope of the current document.

The Figure 1.2 sums-up all the publications and pieces of software I significantly contributed to, with their relative links. In this graph, edges between programs are the usual notion of software dependency. We extended it to publications when a publication builds on top of some previous work.[7] One can see my PhD [Rou13] work on the bottom left corner. Following work on invariant inference appears on top of it, still on the left. Then work on rigorous proofs and floating-point arithmetic comes further right, in the middle of the figure and work on network calculus sits on the right.

When writing the current document, valuing clarity and – at least relative – conciseness more than completeness, I had to make some choices on what I wanted to put forward, among a bit more of a decade of various work, as summarized on Figure 1.2. Thus, some publications [BR16, BRDP21, CDGR11, DGG$^+$18, GR11, PGH$^+$21, PGH$^+$23, RDG10, RG13a, RJG15, Rou14, RQB22, RS10, WGR$^+$16] are voluntarily left completely out of the scope of this document. This includes the PhD thesis work of Baptiste Pollien [PGH$^+$21, PGH$^+$23] I recently coadvised with Christophe Garion from the nearby school ISAE (aerospace engineering, former SUPAERO), Gautier Hattenberger from the nearby school ENAC (civil aviation school) and Xavier Thirioux, also from ISAE. Baptiste performed some verification work on Paparazzi, an UAV autopilot developed at ENAC.

[6]Usually around 60 hours per year.

[7]I stole the idea from the HDR thesis of Guillaume Melquiond. Thanks to him!

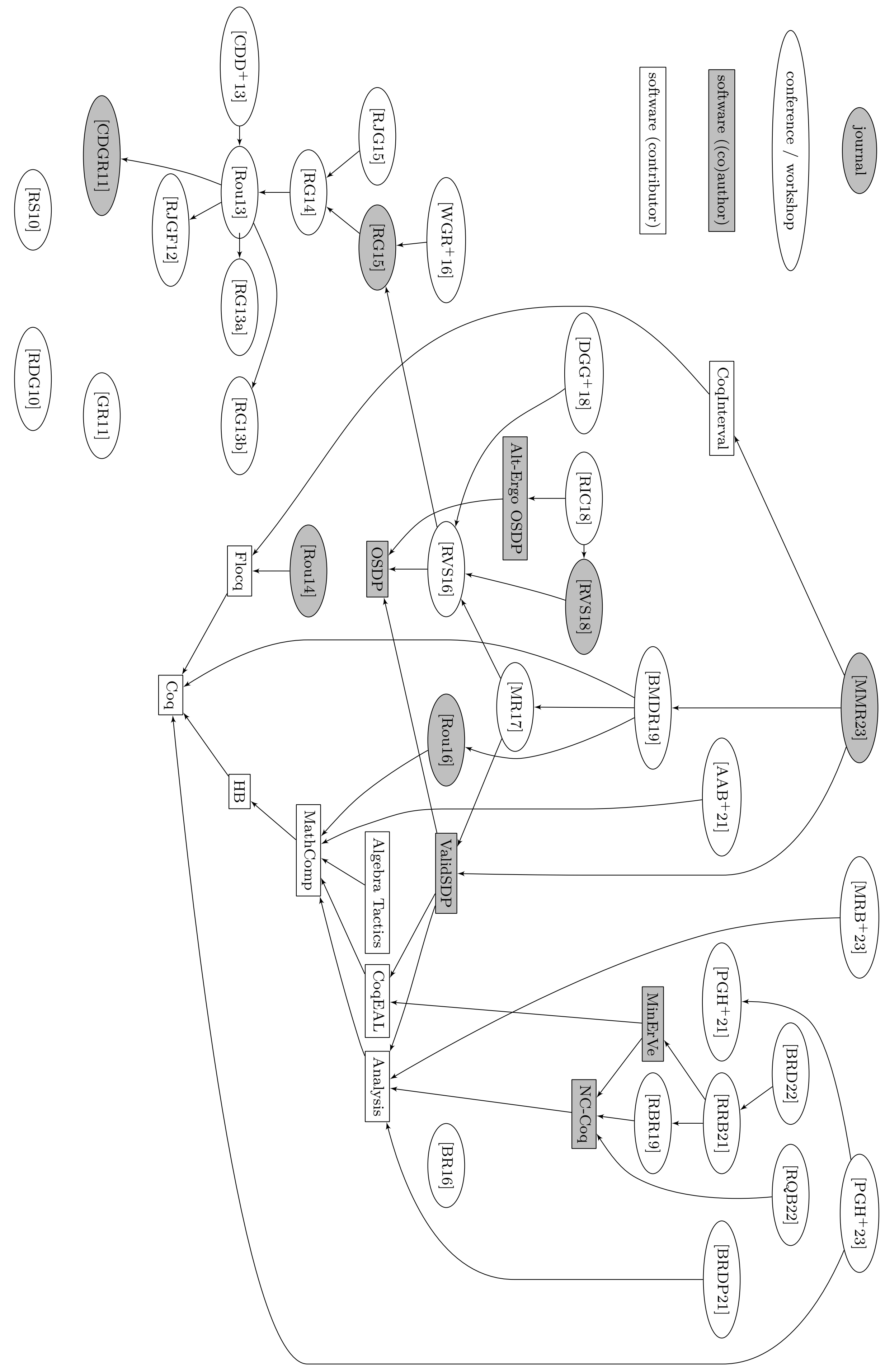


Figure 1.2: Summary of publications and programs. Publications are ordered by year, most recent on right.

Chapter 2

Convex Optimization and Policy Iterations

2.1 Introduction

During my PhD thesis [Rou13], I developed an interest in verification of programs implementing control-command regulators, mostly linear with potentially a few nonlinear features such as saturations. Although actual control-command programs of modern airliners are much more complicated, they remain based on such linear cores. More precisely, I targeted the synthesis of quadratic invariants for such programs, using a method called policy iterations, based on tools called convex-optimization solvers.

The current chapter will first introduce the considered problem and its then state of the art, the policy iteration methods (both min- and max-policy), some refinements we developed to make their use more practical in our context [RG13b, RG15, RJGF12] and finally a practical comparison on some benchmark programs between min- and max-policy iterations [RG14]. This chapter can be seen as a motivation for rigorously proving polynomial inequalities with a high degree of automation. The reader only interested in those numerical proofs themselves, can directly jump to next Chapter 3

Since the first proposals in the 70s, static analysis, and more specifically abstract interpretation through Kleene-based fixpoint computation [CC77, CC79, CC92], has been widely developed and is now considered usable on realistic systems. Those techniques provide an over-approximation of the program semantics by computing *iteratively* a fixpoint in an *abstract domain*. To cope with convergence issues, the iterative sequence of increasing elements is itself approximated using the (in)famous widening operator.

When the target property, e.g., boundedness of the variable values, or unreachability of bad values, cannot be guaranteed, the following two elements are usually to blame: the choice of the abstract domain and the use of widening. Since almost half a century, the set of abstractions proposed by the static analysis community is mainly bound to linear abstractions: from interval arithmetic based analysis, to convex polyhedra [CH78], or less expensive yet precise abstractions such as zonotopes [GGP09] or octagons [Min01]. Regarding the widening, its use is mandatory but hard to control. Threshold widening, or widening up to [HPR97], for instance strongly depends on the set of thresholds chosen *a priori*, and can give radically different results depending on this choice.

Among the solutions offered for those issues, policy[1] iterations were introduced [GGTZ07, GS07]. It is inspired by classical approaches in game theory. In short, using policy iterations could replace widening and compute precise fixpoints using numerical solvers. They also enable the use of other than linear abstractions, such as quadratic polynomials [AGG10, GS10, GSA$^+$12], thanks to the availability of efficient numerical solvers for convex optimization.

While policy iteration can have a wide impact in static analysis, its use was hampered by practical issues:

1. It relies on an *a priori* known set of templates on which the computation is performed. This choice of templates can have a dramatic impact on the result.
   How to choose appropriate templates for a given problem/program?

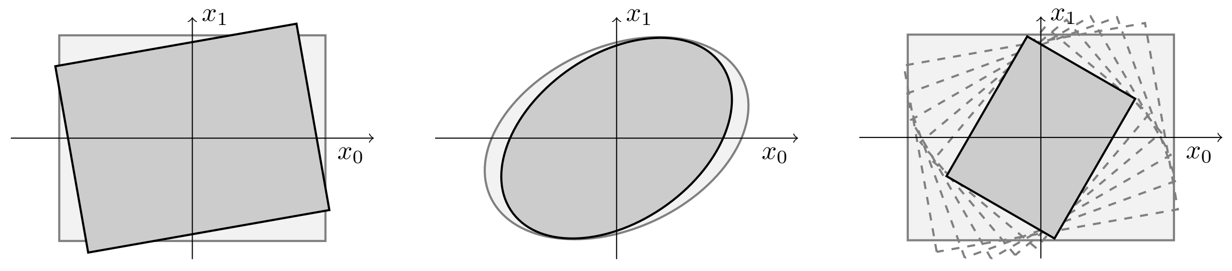
2. It uses numerical solvers, relying on floating-point arithmetic, and analyzes programs, themselves using floating-point arithmetic.
   How to trust the validity of the resulting invariant?

Targeted programs are the typical time-triggered linear controllers as found in a wide range of critical cyber-physical systems such as car engines, flight commands of an aircraft or even medical devices such as

[1]The term *strategy* is also used in the literature, with equivalent meaning.

```
x0 := 0; x1 := 0; x2 := 0;
while −1 ≤ 0 do
  in := ?(−1, 1); x0’ := x0; x1’ := x1; x2’ := x2;
  x0 := 0.9379 x0’−0.0381 x1’−0.0414 x2’+0.0237 in;
  x1 := −0.0404 x0’+0.968 x1’−0.0179 x2’+0.0143 in;
  x2 := 0.0142 x0’−0.0197 x1’+0.9823 x2’+0.0077 in;
od
```

Figure 2.1: Example of a control-command program. These are essentially infinite loops updating a few variables (here “x0”, “x1” and “x2”) as a linear combination of their previous values and some input read from a sensor (here “in” takes some arbitrary value in the $[-1, 1]$ interval, changing at each loop iteration). In an actual controller, the loop iterations would be periodically spaced in time (for instance every 10 milliseconds) and an output would be computed from “x0”, “x1” and “x2” and sent to an actuator at each iteration.



(a) an interval property: not inductive (b) a quadratic property: inductive (c) the interval property is 6-inductive.

Figure 2.2: Intervals vs quadratic properties for a linear transformation.

pacemakers or insulin pumps. Most of them are based on a linear update performed within an infinite loop.

For such systems, we offer

- an algorithm to compute appropriate quadratic templates;
- an *a posteriori* proof that the invariant, computed using numerical solvers, is actually valid, even in presence of floating point computations in the analyzed program.

The first step is presented in Section 2.4 and relies on numerical optimization to identify appropriate templates. This mostly consists in a more comprehensive exposition of material already presented in previous papers [CDD$^+$13, RG13b, RG14, RJGF12]. The second step, detailed in Section 2.5, is based on a set of theorems about error bounds on floating point computations. Their proofs being particularly tedious and error prone, they are supported and mechanically checked by a proof assistant (Coq [Coq24]). Before that, Sections 2.2, and 2.3 introduce respectively our interest for quadratic invariants, compared to linear ones and the use of policy iterations. Finally, Section 2.6 gives experimental results.

2.2 Need for Quadratic Invariants

This section introduces quadratic invariants and compares them to the more usual linear invariants for static analysis of control-command systems.

2.2.1 Linear Domains

Most control systems are based on a linear core. This is for instance the case of the Linear Quadratic Gaussian regulator given in Figure 2.1. Unfortunately, these are hard to analyze using simple linear abstract domains, such as the intervals domain (for instance, the previous regulator does not admit any nontrivial invariant in this domain). Figure 2.2a gives an intuition of this point. An interval property on two variables undergoes a small rotation composed with an homothety of factor $0.92 < 1$ (i.e., a strictly contracting linear transformation), showing that the property is not inductive. Nevertheless, as control theorists know for long, stable linear systems admit quadratic invariants (called Lyapunov functions [BEEFB94, Lya47]). Such invariants can be depicted as ellipsoids. On Figure 2.2b, an ellipsoid is depicted along with its image by the same linear transformation as previously. This time, the property appears to be inductive.

In practice, when a quadratic invariant exists, approximating it with enough faces can give an invariant in classic linear abstract domains such as the polyhedra [CH78] or the zonotope domains [GGP09]. Thus, it could be thought that quadratic invariants are useless and that the same results can be obtained using solely linear invariants. However, this suffers two, often prohibitive, issues making it a mere theoretical approach:

Large Number of Faces. "enough faces" can be way too large to be actually tractable, particularly when the number of variables grows. This issue becomes even more stringent in presence of weakly contracting transformations, intuitively requiring the linear invariant to be "smooth" enough to be inductive, hence composed of a large number of faces. More than the memory space required to store these objects, the cost of their manipulation may become intractable. Compared to linear invariants, quadratic invariants have a space complexity quadratic in the number of variables and are intrinsically "smooth".

Ineffectiveness of Kleene Iterations. Existence of an invariant does not mean existence of a practical way to compute it. In particular, Kleene iterations with polyhedra are known to perform poorly when trying to generate such linear invariants [SB13]. Moreover none of the classic widening strategies allows to find such results without performing a large number of iterations.

Moreover, control systems can also contain guards. For instance, resets and saturations are two common kinds of guards. *Resets* can at any time reset the value of all program variables to some constant (possibly different from the initial value). They are usually rather easy to handle since they can just be considered as an additional initial value. *Saturations* force a variable to remain in some range by keeping it constant when it reaches the boundaries of the range. They can be much harder to handle. Adding a saturation to a stable linear system can even make it unstable.

2.2.2 Unrolling

Unrolling constitutes a practical alternative to the search for linear inductive invariants with myriads of faces. For purely linear systems, unrolling to depths k ranging from a few hundreds to a few thousands allows to compute precise k-inductive invariants while keeping the number of faces reasonably small [Fer04, GGP09]. Some work [SB13] even demonstrates that precise bounds (i.e., the maximum reachable values) can be computed with simple support functions by fully unrolling the system. Intuitively, unrolling turns a contracting transformation into a more contracting one. Thus, properties which are not inductive may appear k-inductive. This is illustrated on Figure 2.2c.

These results are definitely interesting but only produce k-inductive invariants for large values of k which exhibits the following drawbacks:

Checking Results. When the user does not trust the analyzer and wants to check its results a posteriori, not having a simple inductive invariant can seriously complicate the task[2].

Difficulty of Unrolling in Presence of Guards. Unrolling purely linear systems[3] works well because the size of the unrolled system is linear in the unrolling depth k. However, when the system contains guards, things can become more intricate. Considering all paths through k iterations, can lead to a system of exponential size 2^k which rapidly becomes intractable for large values of k.

2.2.3 Quadratic Invariants

Although quadratic invariants have been known for a long time, as quadratic *Lyapunov functions*, from control theorists [BEEFB94, Lya47], their use in static analysis is more recent. The first famous use of quadratic invariants for static analysis was two dimensional ellipsoids to bound second order filters [Fer04, Fer05, Mon05]. Bounds on filters of order n could then be computed by decomposing them into filters of order 1 (bounded with intervals) and 2 (bounded with ellipsis) and refining the obtained bounds by means of some kind of unrolling [Fer04, Fer05, Mon05]. The method then presents the same advantages (precision of the computed bounds) and drawbacks (no inductive invariant is produced) as other unrolling methods. Other works offer to compute quadratic inductive invariants of higher dimensions on larger classes of linear systems [AFP09, AGG10, GS10, RFM05]. Such invariants are computed thanks to the use of some numerical solvers, namely semidefinite programming solvers. Table 2.1 summarizes the respective advantages and drawbacks of linear invariants with unrolling and quadratic invariants.

Example 1. *In the remainder of this chapter, the following invariant*[4] *will be* fully automatically *computed on the code of Figure 2.1:* $6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_0x_2 + 2.4182x_1x_2 \leq 1.0029 \wedge |x_0| \leq 0.4236 \wedge |x_1| \leq 0.3371 \wedge |x_2| \leq 0.5251$*. This invariant is a cropped ellipsoid as displayed on Figure 2.3.*

[2]Although the k-inductive invariants can be made (1)-inductive by adding extra variables, representing past values of program variables, in their expression.

[3]Like the one in Figure 2.1.

[4]All figures are rounded to the fourth digit.

	linear domains with unrolling [Fer04, Fer05, GGP09, Mon05, SB13]	quadratic domains [AFP09, AGG10, GS10, GSA$^+$12, RJGF12]
pure linear systems	+ high precision	− less precise
simple guards (e.g., reset)	+ easily handled	+ easily handled
other guards (e.g., saturation)	− cannot be handled	+ often handled
size of generated invariants	− potentially huge	+ quadratic

Table 2.1: Pros and cons of linear domains with unrolling and quadratic domains.

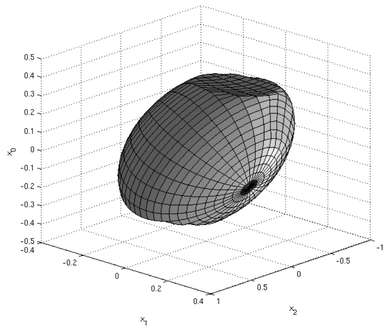


Figure 2.3: Invariant for our running example.

Remark 1 (Exact Reachable State Space is not an Ellipsoid)**.** *Despite the ability of quadratic invariants to bound any stable linear system, it should be noted that the reachable state space of such systems is usually not an ellipsoid. Thus, although ellipsoids are good invariants, they will not always yield the tightest possible bounds. See [RG15, Example 3] for an example.*

This section advocated how nice ellipsoids are to bound linear systems. Yet, they suffer a significant disadvantage compared to classic abstract domains such as polyhedra, octagons, zonotopes,. . . : the set of ellipsoids with the inclusion order can hardly be equipped with a sensible join operator[5], as seen in [RG15, Fig. 7]. This constitutes a major obstacle to practical computations through Kleene iterations on the whole set of ellipsoids as usually done in the abstract interpretation framework. A common solution is to choose — prior to the analysis — ellipsoid *shapes* and then only compute their *radii*. Section 2.3 reviews policy iteration, an efficient technique to compute such radii, while Section 2.4 offers a technique to determine relevant ellipsoid shapes.

2.3 Policy Iterations: State of the Art

Computation of precise invariants on numerical programs can be hard to achieve using classic Kleene iterations with widening. Policy iterations [AGG10, CGG$^+$05, GS07, GS10, GSA$^+$12] is one of the alternatives to simple widening developed during the last decades [BSC12, FG10, GR06, HH12, SJ11, and references therein]. This technique allows computing precise postfixpoints, usually by relying on mathematical optimization solvers. Such techniques have been developed for the computation of quadratic invariants for linear systems [AGG10, GS10, GSA$^+$12]. Policy iterations basically perform iterations with two phases

- *Compute a policy*, that is locally simplify the fixpoint problem;
- *Solve the policy* with efficient tools specialized for this simpler problem.

These two phases are alternatively performed until a good result is reached.

2.3.1 Template Domains

Policy iteration is performed on template domains. Given a finite set $\{t_1, \dots, t_n\}$ of expressions on program variables $\mathbb{V}$, the template domain $\mathcal{T}$ is defined as $\overline{\mathbb{R}}^n = (\mathbb{R} \cup \{-\infty, +\infty\})^n$ and the meaning of an abstract value $(b_1, \dots, b_n) \in \mathcal{T}$ is the set of environments

$$\gamma_{\mathcal{T}}(b_1, \dots, b_n) = \{\rho \in (\mathbb{V} \to \mathbb{R}) \mid [\![t_1]\!](\rho) \leq b_1, \dots, [\![t_n]\!](\rho) \leq b_n\}$$

where $[\![t_i]\!](\rho)$ is the result of the evaluation of expression t_i in environment ρ. In other words, the abstract value $(b_1, \dots, b_n)$ represents all the environments satisfying all the constraints $t_i \leq b_i$.

[5]Although the minimum volume (Löwner-Johns) ellipsoid [BV04, Section 8.4] could be a least unreasonable choice.

$$\begin{tikzcd}[column sep=large] \textcircled{1} \arrow[r, "\begin{array}{l} x_0 := 0 \\ x_1 := 0 \\ x_2 := 0 \end{array}"] & \textcircled{2} \arrow[loop right, "-1 \le in \le 1 \ , \ \begin{array}{l} x_0 := 0.9379\,x_0 - 0.0381\,x_1 - 0.0414\,x_2 + 0.0237\,in \\ x_1 := -0.0404\,x_0 + 0.968\,x_1 - 0.0179\,x_2 + 0.0143\,in \\ x_2 := 0.0142\,x_0 - 0.0197\,x_1 + 0.9823\,x_2 + 0.0077\,in \end{array}"] \end{tikzcd}$$

Figure 2.4: Control flow graph for our running example.

Example 2. *Given the quadratic templates* $t_1 := 6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_0x_2 + 2.4182x_1x_2$*,* $t_2 := x_0^2$*,* $t_3 := x_1^2$ *and* $t_4 := x_2^2$*, the quadratic invariant from Example 1, page 7, can be written* $(1.0029,\ 0.1795,\ 0.1136,\ 0.2757) \in \mathcal{T}$.

Indeed, many common abstract domains can be rephrased as template domains. For instance the intervals domain is obtained with templates $-x_i$ and x_i for all variables $x_i \in \mathbb{V}$ and the octagon domain [Min01] by adding all the $\pm x_i \pm x_j$. The shape of the templates to be considered for policy iteration depends on the optimization tools used. For instance, linear programming [GGTZ07, GS07] allows any linear templates whereas quadratic templates can be handled thanks to semidefinite programming [AGG10, GS10, GSA+12]. This chapter focuses on the latter case.

2.3.2 System of Equations

While Kleene iterations iterate locally through each construct of the program, policy iterations require a global view on the analyzed program. For that purpose, the whole program is first translated into a system of equations which is later solved.

Starting from the control flow graph of the analyzed program, a system of equations is defined with a variable $b_{i,j}$ for each template t_i and each vertex j of the graph.

Example 3. *Figure 2.4 displays the control flow graph for our running example (Figure 2.1, page 6). Here is its translation as a system of equations:*

$$\left\{\begin{array}{lcl} b_{1,1} & = & +\infty \qquad b_{2,1} = +\infty \qquad b_{3,1} = +\infty \qquad b_{4,1} = +\infty \\ b_{1,2} & = & \max\{0 \mid \mathrm{be}(1)\} \sqcup \max\{r(t_1) \mid (-1 \le in \le 1) \wedge \mathrm{be}(2)\} \\ b_{2,2} & = & \max\{0 \mid \mathrm{be}(1)\} \sqcup \max\{r(t_2) \mid (-1 \le in \le 1) \wedge \mathrm{be}(2)\} \\ b_{3,2} & = & \max\{0 \mid \mathrm{be}(1)\} \sqcup \max\{r(t_3) \mid (-1 \le in \le 1) \wedge \mathrm{be}(2)\} \\ b_{4,2} & = & \max\{0 \mid \mathrm{be}(1)\} \sqcup \max\{r(t_4) \mid (-1 \le in \le 1) \wedge \mathrm{be}(2)\} \end{array}\right. \tag{2.1}$$

where $\mathrm{be}(j)$ *denotes* $(t_1 \le b_{1,j}) \wedge (t_2 \le b_{2,j}) \wedge (t_3 \le b_{3,j}) \wedge (t_4 \le b_{4,j})$ *and* $r(t)$ *is the template* t *in which variable* x_0 *is replaced by* $0.9379\,x_0 - 0.0381\,x_1 - 0.0414\,x_2 + 0.0237\,in$*, variable* x_1 *is replaced by* $-0.0404\,x_0 + 0.968\,x_1 - 0.0179\,x_2 + 0.0143\,in$ *and variable* x_2 *is replaced by* $0.0142\,x_0 - 0.0197\,x_1 + 0.9823\,x_2 + 0.0077\,in$*. The usual maximum on* $\overline{\mathbb{R}}$ *is denoted*[6] $\sqcup$.

Each $b_{i,j}$ bounds the template t_i at program point j and is defined in one equation as a maximum over as many terms as incoming edges in j. More precisely, each edge between two vertices v and v' translates to a term in each equation $b_{i,v'}$ on the pattern: $\max\left\{r(t_i) \;\middle|\; c \wedge \bigwedge_j (t_i \le b_{i,v})\right\}$ where c and r are respectively the constraints and the assignments associated to this edge. This expresses the maximum value the template t_j can reach in destination vertex v' when applying the assignments r on values satisfying both the constraints c of the edge and the constraints $t_i \le b_{i,v}$ of the initial vertex v. Finally, the program starting point is initialized to $(+\infty, \dots, +\infty)$, meaning all equations for b_{i,j_0}, where j_0 is the starting point, become $b_{i,j_0} = +\infty$.

Thus, for any solution $(b_{1,1}, \dots, b_{n,1}, \dots)$ of the equations, $\gamma_{\mathcal{T}}(b_{1,j}, \dots, b_{n,j})$ is an overapproximation of reachable states of the program at point j. According to the Knaster-Tarski theorem, this set of solutions has a least element which then gives the best overapproximation of the reachable state space of the program.

2.3.3 Policy Iterations

Policy iterations intend to compute the least solution of the previous system of equations. They are an iterative process with two phases. First, an abstraction of the problem is computed. This abstraction, called *policy*, can then be solved using techniques which were not applicable on the original problem. This gives an approximation of the final result enabling to find a better policy, itself giving a better approximation of the result and so on.

Two different techniques, min- and max-policies, can be found in the literature. They basically apply the previous scheme top down or bottom up respectively.

[6] $\vee$ is often used instead in the policy iteration literature.

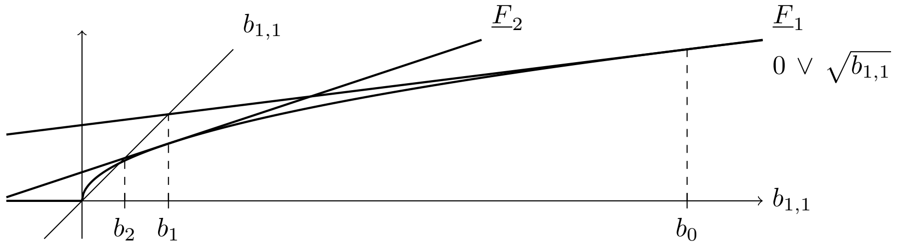


Figure 2.5: Illustration of Example 5.

Min-Policy Iterations

To some extent, min-policy iterations [AGG10] can be seen as a very efficient *narrowing*, since they perform descending iterations from a postfixpoint towards some fixpoint, working in a way similar to the Newton-Raphson numerical method. Iterations are not guaranteed to reach a fixpoint but can be stopped at any time leaving an overapproximation thereof. Moreover, convergence is usually fast.

Writing a system of equations $b = F(b)$ with $b = (b_{i,j})_{i\in[\![1,n]\!],j\in[\![1,p]\!]}$ and $F : \overline{\mathbb{R}}^{np} \to \overline{\mathbb{R}}^{np}$ (n being the number of templates and p the number of vertices in the control flow graph), a min-policy is defined as follows: $\underline{F}$ is a min-policy for F if for every $b \in \overline{\mathbb{R}}^{np}$, $F(b) \leq \underline{F}(b)$ and there exist some $b_0 \in \overline{\mathbb{R}}^{np}$ such that $\underline{F}(b_0) = F(b_0)$. We will only consider *linear* min-policies in the remaining of this chapter. For instance, for a smooth concave function, its min-policies are the tangents to its graph.

Example 4. *Considering the system of one equation* $b_{1,1} = 0 \sqcup \sqrt{b_{1,1}}$ *, where* $\sqrt{x}$ *is defined as* $-\infty$ *for negative numbers* x*,* $\underline{F}$ *defined as* $\underline{F}(b) := 0 \sqcup \left(\frac{b_{1,1}}{8} + 2\right)$ *is a min-policy. Indeed, for all* $b_{1,1} \in \overline{\mathbb{R}}$*,* $F(b) = 0 \sqcup \sqrt{b_{1,1}} \leq 0 \sqcup \frac{b_{1,1}}{8} + 2 = \underline{F}(b)$*, and for* $b_0 = 16$*,* $F(b_0) = \sqrt{16} = \frac{16}{8} + 2 = \underline{F}(b_0)$*. This is illustrated on Figure 2.5 on which* $\underline{F}_1$ *is the above* $\underline{F}$*.*

The following theorem can then be used to compute the least fixpoint of F.

Theorem 1. *Given a (potentially infinite) set* $\underline{\mathcal{F}}$ *of min-policies for* F*. If for all* $b \in \overline{\mathbb{R}}^{np}$ *there exist a policy* $\underline{F} \in \underline{\mathcal{F}}$ *interpolating* F *at point* b *(i.e.* $\underline{F}(b) = F(b)$*) and if each* $\underline{F} \in \underline{\mathcal{F}}$ *has a least fixpoint* $\operatorname{lfp}\underline{F}$*, then the least fixpoint of* F *satisfies*

$$\operatorname{lfp} F = \bigsqcap_{\underline{F}\in\underline{\mathcal{F}}} \operatorname{lfp}\underline{F}.$$

Remark 2. *This enables to better understand the name* min*-policies since, in the hypotheses of the previous theorem,* F *is the pointwise minimum of the min-policies* $\underline{F} \in \underline{\mathcal{F}}$*:*

$$F = \bigsqcap_{\underline{F}\in\underline{\mathcal{F}}} \underline{F}.$$

Iterations are done with two main objects: a min-policy $\underline{F}$ and a tuple b of values for variables $b_{i,j}$ of the system of equations. The following policy iteration algorithm starts from some postfixpoint b_0 of F and aims at refining it to produce a better overapproximation of a fixpoint of F. Policy iteration algorithms always proceed by iterating two phases: first a policy is selected then it is solved. In our case:

- find a linear min-policy $\underline{F}_{i+1}$ being tangent to F at point b_i, this can be done thanks to SDP solvers [GSA+12, Section 5.4];
- compute the least fixpoint b_{i+1} of policy $\underline{F}_{i+1}$ thanks to linear programming.

Iterations can be stopped at any point (for instance after a fixed number of iterations or when progress between b_i and b_{i+1} is considered small enough) leaving an overapproximation b of a fixpoint of F.

Example 5. *We perform min-policy iterations on the system of equations of Example 4.*

- *We start from the postfixpoint* $b_0 = 16$*. This postfixpoint could be obtained through Kleene iterations for instance*[7]*.*

[7]Or a large enough guess can be used. Thanks to the fast convergence of min-policy iterations, there is often no need for this postfixpoint to be close to the fixpoint eventually computed.

- *For each term of the unique equation, we look for a hyperplane tangent to the term at point b_0. 0 is tangent to 0 at point b_0 and $\frac{b_{1,1}}{8}+2$ is tangent to $\sqrt{b_{1,1}}$ at point b_0 (c.f., Figure 2.5), this gives the following linear min-policy:*
  $\underline{F}_1 =$ $$\left\{\ b_{1,1} = 0 \sqcup \left(\frac{b_{1,1}}{8}+2\right)\right.$$
- *The least fixpoint of $\underline{F}_1$ is then: $b_1 = \frac{16}{7} \simeq 2.2857$.*
- *Looking for hyperplanes tangent at point b_1 gives the min-policy:*
  $\underline{F}_2 =$ $$\left\{\ b_{1,1} = 0 \sqcup \left(\frac{\sqrt{7}}{8}b_{1,1}+\frac{2}{\sqrt{7}}\right)\right.$$
- *Hence $b_2 = \frac{16}{8\sqrt{7}-7} \simeq 1.1295$.*

These two first iterations are illustrated on Figure 2.5. The procedure then rapidly converges to the fixpoint $b_{1,1} = 1$ (the next iterates being $b_3 \simeq 1.0035$ and $b_4 \simeq 1.0000$) and can be stopped as soon as the accuracy is deemed satisfying.

Remark 3. *The Newton-Raphson method on a smooth concave function is a particular case of min-policy iterations.*

Example 6. *We perform min-policy iterations on the running example (Equation (2.1), page 9).*

- *We start from the postfixpoint $b_0 = (+\infty, +\infty, +\infty, +\infty, 1000000, +\infty, +\infty, +\infty)$.*
- *For each term of each equation, we look for an hyperplane tangent to the term at point b_0. This can be done thanks to SDP solvers[8] and gives the following linear min-policy:*
  $\underline{F}_1 =$ $$\left\{\begin{array}{llll} b_{1,1} = +\infty & b_{2,1} = +\infty & b_{3,1} = +\infty & b_{4,1} = +\infty \\ b_{1,2} = 0 \sqcup 0.9857\, b_{1,2} + 0.0152 & & b_{2,2} = 0 \sqcup 0.2195\, b_{1,2} + 11.0979 & \\ b_{3,2} = 0 \sqcup 0.1143\, b_{1,2} + 4.8347 & & b_{4,2} = 0 \sqcup 0.2669\, b_{1,2} + 3.9796. & \end{array}\right.$$
- *A linear programming solver allows computing the least fixpoint of $\underline{F}_1$:*
  $b_1 = (+\infty, +\infty, +\infty, +\infty, 1.0664, 11.3324, 4.9568, 4.2644)$.
- $\underline{F}_2 =$ $$\left\{\begin{array}{llll} b_{1,1} = +\infty & b_{2,1} = +\infty & b_{3,1} = +\infty & b_{4,1} = +\infty \\ b_{1,2} = 0 \sqcup 0.9857\, b_{1,2} + 0.0143 & & b_{2,2} = 0 \sqcup 0.2302\, b_{1,2} + 0.0120 & \\ b_{3,2} = 0 \sqcup 0.1190\, b_{1,2} + 0.0052 & & b_{4,2} = 0 \sqcup 0.2708\, b_{1,2} + 0.0042 & \end{array}\right.$$
- $b_2 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.2429, 0.1245, 0.2757)$.

Four additional iterations lead to $b_6 = (+\infty, +\infty, +\infty, +\infty,\ 1.0029,\ 0.1795,\ 0.1136,\ 0.2757)$ which is the invariant given in Example 1 and depicted on Figure 2.3.

Remark 4 (Number and size of semidefinite programs)**.** *At each iteration, one semidefinite program has to be solved for each term of each equation in order to compute a new policy. This leads to many semidefinite programs but each focusing on a single term, hence rather small. The computed policies being linear are then solved through linear programming. This way, at the scale of the whole system, only linear programs are solved, which scales better than semidefinite programming.*

Max-Policy Iterations

Behaving somewhat as a super *widening*, max-policy iterations [GS10] work in the opposite direction compared to min-policy iterations. They start from bottom and iterate computations of greatest fixpoints on a set of max-policies until a global fixpoint is reached. Unlike the previous approach, this terminates with a *theoretically* precise fixpoint, but the user has to wait until the end since intermediate results are not overapproximations of a fixpoint.

Max-policies are the dual of min-policies: $\overline{F}$ is a max-policy for F if for every $b \in \overline{\mathbb{R}}^{np}$, $\overline{F}(b) \leq F(b)$ and there exist some $b_0 \in \overline{\mathbb{R}}^{np}$ such that $\overline{F}(b_0) = F(b_0)$. In particular, the choice of one term in each equation is a max-policy. From now on, only this last kind of max-policies will be considered.

Theorem 2. *Given the set $\overline{\mathcal{F}}$ of max-policies for F as defined above (choice of one term in each equation), any fixpoint of F is also a fixpoint of some $\overline{F}_0 \in \overline{\mathcal{F}}$.*

[8]See [RG15, Example 9] for more details.

Iterations are again done with two main objects: a max-policy $\overline{F}$ and a tuple b of values for variables $b_{i,j}$ of the system of equations. The following policy iteration algorithm aims at finding a policy $\overline{F}_0$ as in the above theorem by solving optimization problems. The initial value $b_0 := (-\infty, \dots, -\infty)$ is chosen, then policies are iterated:

- find an improving policy $\overline{F}_{i+1}$ at point b_i, i.e. that reaches (strictly) greater values evaluated at point b_i, this can be done by evaluating each term of the system of equations at point b_i [GSA+12, Section 6.2]; if none is found, exit;
- compute the greatest fixpoint[9] b_{i+1} of policy $\overline{F}_{i+1}$ using SDP solvers [GSA+12, Section 3.7].

The last tuple b is then a fixpoint of the whole system of equations.

Remark 5. *Although min and max policies are dual concepts, we are in both cases looking for* over*approximations of the least fixpoint of the system of equations, thus the algorithms are* not *dual.*

Example 7. *We perform max-policy iterations on the running example (Equation (2.1), page 9).*

- *We start with the initial value* $b_0 = (-\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)$.
- *We now look for an improving policy* $\overline{F}_1$ *at point* b_0. *For the first four equations, there is no choice and the term* $+\infty$ *is chosen. For the four remaining equations, replacing the* $b_{i,j}$ *with values* $-\infty$ *from* b_0 *in* be(1) *and* be(2) *gives formula equivalent to* $false$*, hence both terms of these equations are maximum of the empty set and evaluate to* $-\infty$. *We can then choose any of them:*
$$\overline{F}_1 = \begin{cases} b_{1,1} = +\infty \quad b_{2,1} = +\infty & b_{3,1} = +\infty \quad b_{4,1} = +\infty \\ b_{1,2} = \max\{0 \mid \mathrm{be}(1)\} & b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{3,2} = \max\{0 \mid \mathrm{be}(1)\} & b_{4,2} = \max\{0 \mid \mathrm{be}(1)\}. \end{cases}$$
- *Hence* $b_1 = (+\infty, +\infty, +\infty, +\infty, 0, 0, 0, 0)$.
- *Now* b_1 *no longer has any* $-\infty$, *so* be(2) *is no longer* $false$ *and it becomes interesting to select the second terms in the last four equations, hence:*
$$\overline{F}_2 = \begin{cases} b_{1,1} = +\infty \quad b_{2,1} = +\infty \quad b_{3,1} = +\infty \quad b_{4,1} = +\infty \\ b_{1,2} = \max\{r(t_1) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{r(t_2) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{3,2} = \max\{r(t_3) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{4,2} = \max\{r(t_4) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\}. \end{cases}$$
- *The greatest fixpoint of* $\overline{F}_2$ *can be computed thanks to SDP solvers*[10]*:*
$b_2 = (+\infty, +\infty, +\infty, +\infty,\ 1.0077,\ 0.1801,\ 0.1141,\ 0.2771)$.
- *No more improving policy.*

After three iterations, the algorithm has found the same least fixpoint than min-policies in Example 6.

Remark 6 (Number and size of semidefinite programs)**.** *Contrary to min-policies (c.f., Remark 4), max-policies are not linear. Solving them then requires semidefinite programs whereas min-policies only solve linear programs at the scale of the whole system.*

The max-policy iteration builds an ascending chain of abstract elements similarly to Kleene iterations elements. However it is guaranteed to be finite, while Kleene iterations require the use of widening to ensure termination. Indeed, since there are exponentially many max-policies, we have a bound on the number of iterations. And despite this exponential bound, in practice, only a small number of policies are usually considered and the number of iterations remains reasonable.

2.4 Template Generation

Template domains used by policy iteration require templates to be given prior to the analyses. This greatly limits the automation of the method. However, heuristics can be designed for linear systems of the form $x_{(k+1)} = Ax_{(k)} + Bu_{(k)}$, like our running example. Those are ubiquitous in control applications where the vector x represents the internal state of the controller and u a bounded input.

After a brief introduction to Lyapunov stability theory, this section first focuses on generating templates for pure linear systems then for guarded linear systems given as a control flow graph.

[9] More precisely, first determine which $b_{i,j}$ are $\pm\infty$ in the least fixpoint in $\overline{\mathbb{R}}^{np}$ greater than b_i, then compute a greatest fixpoint for the remaining values in $\mathbb{R}$.

[10] See [RG15, Example 11] for details.

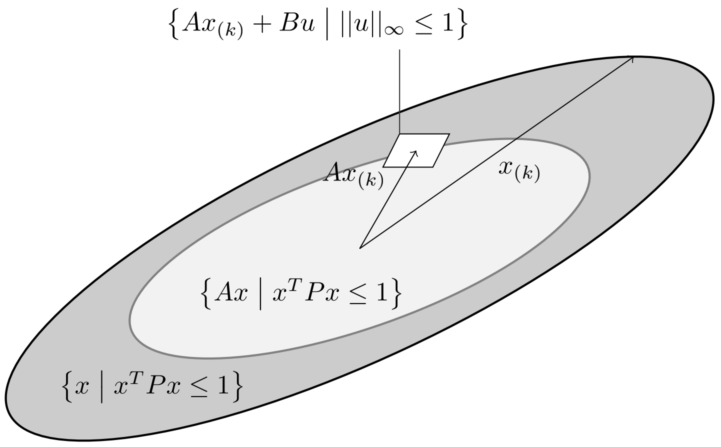

$\{Ax_{(k)} + Bu \mid ||u||_\infty \le 1\}$
$Ax_{(k)}$
$x_{(k)}$
$\{Ax \mid x^T P x \le 1\}$
$\{x \mid x^T P x \le 1\}$

Figure 2.6: Illustration of the stability concepts: if $x_{(k)}$ is in the dark gray ellipse, then, after a time step, $Ax_{(k)}$ is in the light gray one, which is exactly what is expressed by Equation (2.5). The white box represents the potential values of $x_{(k+1)}$ after adding the effect of the bounded input $u_{(k)}$. We see here the necessity that the light gray ellipse be strictly included in the dark gray one, which is the stronger condition expressed by Equation (2.7).

2.4.1 Introduction to Lyapunov Stability Theory

One common way to establish stability of a discrete, time-invariant closed (i.e., with no inputs) system described in state space form, (i.e., $x_{(k+1)} = f(x_{(k)})$) is to use what is called a Lyapunov function. It is a function $V : \mathbb{R}^n \to \mathbb{R}$ which must satisfy the following properties

$$V(0) = 0 \wedge \forall x \in \mathbb{R}^n \backslash \{0\}, V(x) > 0 \wedge \lim_{\|x\| \to \infty} V(x) = \infty \tag{2.2}$$

$$\forall x \in \mathbb{R}^n, V(f(x)) - V(x) \le 0. \tag{2.3}$$

It is shown, for instance in [HC08], that exhibiting such a function proves the Lyapunov stability of the system, meaning that its state variables will remain bounded through time. Equation (2.3) expresses the fact that the function $k \mapsto V(x_{(k)})$ decreases, which, combined with (2.2), shows that the state variables remain in the bounded sublevel-set $\{x \in \mathbb{R}^n | V(x) \le V(x_{(0)})\}$ at all instants $k \in \mathbb{N}$.

In the case of Linear Time Invariant systems [BEEFB94] (of the form $x_{(k+1)} = Ax_{(k)}$, with $A \in \mathbb{R}^{n \times n}$), one can always look for V as a quadratic form in the state variables of the system: $V(x) = x^T P x$ with $P \in \mathbb{R}^{n \times n}$ a symmetric matrix such that

$$P \succ 0 \tag{2.4}$$

$$A^T P A - P \preceq 0. \tag{2.5}$$

Now, to account for the presence of an external input to the system (which is usually the case with controllers: they use data collected from sensors to generate their output), the model is usually extended into the form

$$x_{(k+1)} = Ax_{(k)} + Bu_{(k)}, \|u_{(k)}\|_\infty \le 1. \tag{2.6}$$

The condition $\|u_{(k)}\|_\infty \le 1$ reflects the fact that values coming from input sensors usually lie in a given range. The bound 1 is chosen without loss of generality since one can always alter the matrix B to account for different bounds. Then, through a slight reinforcement of Equation (2.5) into

$$A^T P A - P \prec 0 \tag{2.7}$$

we can still guarantee that the state variables x of (2.6) will remain in the sublevel set (for some $\lambda > 0$) $\{x \in \mathbb{R}^n \mid x^T P x \le \lambda\}$, which is an ellipsoid in this case, as illustrated on Figure 2.6. This approach only enables us to study control laws that are inherently stable, i.e., stable when taken separately from the plant they control. Nevertheless a wide range of controllers remains that can be analyzed. In addition, inherent stability is required in a context of critical applications.

These stability proofs have the very nice side effect that they provide a quadratic invariant on the state variables, which can be used at the code level to find bounds on the program variables. Furthermore, there are many P matrices that fulfill the equations described above. This gives some flexibility as to the choice of such a matrix: by adding relevant constraints on P, one can obtain increasingly better bounds.

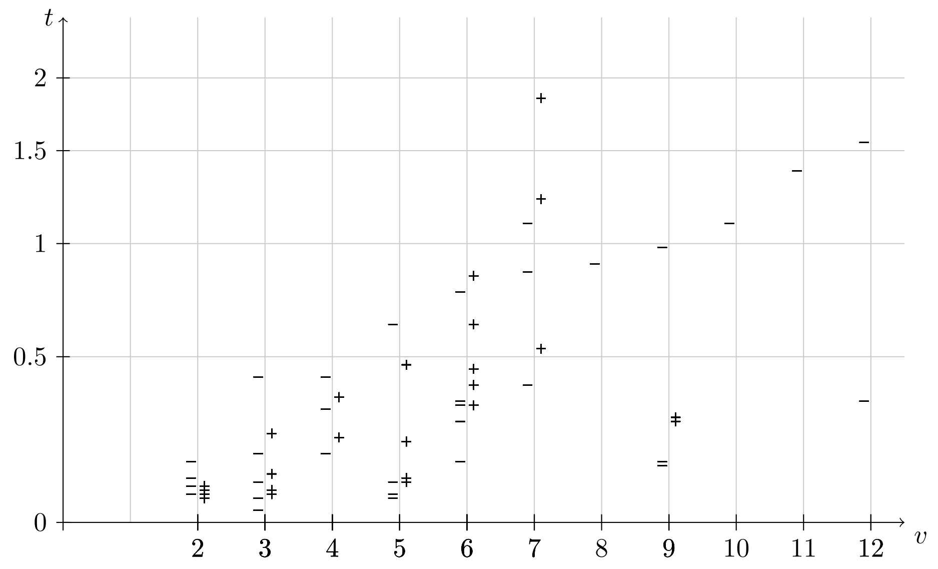


Figure 2.7: Time (t in seconds) spent performing min ($-$ signs) and max ($+$ signs) policy iterations depending on the number v of variables in the analyzed program. Fewer $+$ than $-$ in a column indicates a failure of max-policies on a benchmark. All computations were performed on an Intel Core2 @ 2.66GHz.

2.4.2 Generating Templates

Given a pure linear system ($x_{(k+1)} = Ax_{(k)} + Bu_{(k)}$ with $\|u_{(k)}\|_\infty \leq 1$), we want to generate a quadratic template enabling policy iterations to bound the system. According to Section 2.4.1, any positive definite matrix P solution of the Lyapunov equation (2.7) gives such a quadratic template $t := x^T P\, x$. semidefinite programming constitutes an efficient way to solve this equation. However, taking any random solution may lead to very grossly overapproximated invariants. It would be interesting to constrain more the set of solutions. Multiple approaches exist [RJGF12]. The one used to generate the template of Example 2 relied on looking for a matrix P corresponding to a stable ellipsoid included in a sphere of minimal radius, see [RG15, §5.2] for more details.

2.5 Floating-Point Issues

Two fundamentally different issues arise with floating-point arithmetic:

The analysis itself is carried out with floating-point computations for the sake of efficiency, this usually works well in practice but might give erroneous results, hence the need for some a-posteriori validation, this will be the topic of next Chapter 3.

The analyzed system uses floating-point arithmetic with rounding errors, making it behave differently from the way it would using real arithmetic, this is rapidly discussed below.

Since the synthesized invariants are computed through numerical methods, they contain some margin. The rounding errors performed by the analyzed programs being usually relatively small, they usually easily fit in those margins. Obtaining invariants for the floating-point program can then be done by obtaining an invariant for the ideal program computing in real arithmetic (as done until this point in this chapter), then bounding the rounding errors performed by the program and finally checking that the previous invariant still hold with additional errors within those bounds. More details can be found in [Rou16, §4.3], along with a formalization in the Coq proof assistant.

Such use of abstract domains in the real field to soundly analyze floating-point computations is not new [Min04] and some techniques even allow to finely track rounding errors and their origin in the analyzed program [GP11].

2.6 Experimental Results

All the elements presented in this chapter have been implemented as a new abstract domain in our static analyzer. Experiments were conducted on a set of stable linear systems. These systems were extracted from the literature [AGG10, Fer05, RJGF12, SB13]. The analyzer is released under GPL and available with all examples at `http://cavale.enseeiht.fr/practicalpolicy2014/`.

	n	Total (s)	Templates (s)	Iterations (s)	Check (s)
Ex. 1 From [Fer05, slides]	3	0.12	0.05	0.03	0.01
	3	0.16	0.05	0.06	0.02
	4	0.50	0.15	0.22	$\perp$ (0.01)
Ex. 2 From [Fer05, slides]	5	0.33	0.16	0.06	0.02
	5	0.44	0.15	0.10	0.04
	6	0.77	0.15	0.28	0.12
Ex. 3 Discretized lead-lag controller	3	0.20	0.07	0.10	0.02
	3	0.32	0.07	0.18	0.03
	4	0.68	0.07	0.43	0.08
Ex. 4 Linear quadratic gaussian regulator	4	0.47	0.16	0.19	0.03
	4	0.67	0.16	0.32	0.06
	5	1.13	0.20	0.64	0.13
Ex. 5 Observer based controller for a coupled mass system	6	0.96	0.46	0.16	0.06
	6	1.25	0.47	0.33	0.11
	7	2.28	0.45	0.85	0.26
Ex. 6 Butterworth low-pass filter	6	1.18	0.47	0.35	0.08
	6	1.76	0.45	0.77	0.15
	7	2.67	0.45	1.10	0.26
Ex. 7 Dampened oscillator from [AGG10]	2	0.14	0.01	0.09	0.01
	2	0.23	0.01	0.16	0.02
	3	0.36	0.01	0.20	$\perp$ (0.01)
Ex. 8 Harmonic oscillator from [AGG10]	2	0.11	0.01	0.07	0.01
	2	0.19	0.01	0.12	0.03
	3	0.65	0.01	0.43	0.10

Table 2.2: Result of the experiments: quadratic invariants inference. For each of the eight examples, the first line is for the bare linear system, the second for the same system with an added reset and the third with a saturation. Column n gives the number of program variables considered for policy iteration while column 'Total' gives the time spent for the whole analysis. The remaining columns detail the computation time: 'Templates' corresponds to the quadratic template computation (Section 2.4), 'Iterations' to the actual policy iterations (Section 2.3) and 'Check' to the soundness checking (Section 2.5). $\perp$ indicates failure of the checking (in both cases because the template generation heuristic failed to generate an appropriate template).

Comparing Min- and Max-Policies As seen in Section 2.3, two methods exist to compute invariants by policy iterations, namely min- and max-policies. Figure 2.7 compares analysis times with min and max-policy iterations. In both cases, the number of iterations always remained reasonable. For min-policies, the number of iterations performed lies between 3 and 7 when the stopping criterion is a relative progress below 10^{-4} between two consecutive iterates. For max-policies, the number of iterations was between 4 and 7. As shown on Figure 2.7, computation times for min and max-policies are comparable but the actual differences appear on the largest benchmarks for which max-policies where unable to produce sound results while min-policies did[11]. Finally, it can be noticed that, when both methods work, results obtained with min and max-policies are the same. However, due to numerical issues, min-policies often yield slightly more precise results. For all these reasons, min policies were made the default in our tool.

Benchmarks Figure 2.7 only gave times for policy iterations. Total analysis times also include building the control flow graph and the equation system, computing appropriate templates and eventually checking the soundness of the result. Time needed for control flow graph construction and soundness checking is very small compared to the time spent in policy iterations, whereas computing templates takes the same order of magnitude in time than min-policies iteration. All this is detailed in Table 2.2 for a subset of the benchmarks from Figure 2.7. Finally, Table 2.3 details the bounds obtained for each benchmark and compares them with the known maximum reachable values of the programs.

[11]This is explained by the fact that max-policies have to solve larger SDP problems, incurring more numerical difficulties [GSA+12, Conclusion] (c.f., Remarks 4, page 11 and 6, page 12).

	$\max \lvert\lambda_i\rvert$	Bounds	Reachable
Ex. 1	0.837	$15.98, 15.98$	$14.84, 14.84$
		$15.98, 15.98$	$14.84, 14.84$
		$+\infty, +\infty$	$12.31, 12.31$
Ex. 2	0.837	$1.65, 1.65, 1.00, 1.00$	$1.42, 1.42, 1.00, 1.00$
		$1.65, 1.65, 1.00, 1.00$	$1.42, 1.42, 1.00, 1.00$
		$2.20, 0.50, 1.00, 1.00$	$1.04, 0.50, 1.00, 1.00$
Ex. 3	0.999	$4.03, 20.41$	$3.97, 20.00$
		$4.03, 20.41$	$3.97, 20.00$
		$4.14, 21.41$	$2.04, 1.68$
Ex. 4	0.989	$0.43, 0.35, 0.54$	$0.38, 0.26, 0.48$
		$1.03, 1.01, 1.47$	$1.00, 1.00, 1.00$
		$0.45, 0.37, 0.56$	$0.19, 0.11, 0.17$
Ex. 5	0.840	$4.60, 4.74, 4.34, 4.38$	$2.79, 2.73, 3.50, 3.30$
		$4.60, 4.74, 4.34, 4.38$	$2.79, 2.73, 3.50, 3.30$
		$3.58, 7.04, 5.54, 6.17$	$1.28, 1.69, 3.31, 2.87$
Ex. 6	0.804	$1.42, 1.10, 1.75, 1.82, 2.57$	$1.42, 0.91, 1.44, 1.52, 2.14$
		$1.42, 1.76, 2.63, 3.14, 4.45$	$1.42, 0.91, 1.44, 1.52, 2.14$
		$1.03, 1.37, 1.99, 2.95, 4.02$	$1.03, 0.65, 0.77, 0.88, 1.16$
Ex. 7	0.995	$1.74, 1.74$	$1.29, 1.00$
		$1.74, 1.74$	$1.29, 1.00$
		$+\infty, +\infty$	$1.00, 1.00$
Ex. 8	0.955	$1.27, 1.27$	$1.10, 1.00$
		$1.27, 1.27$	$1.10, 1.00$
		$1.00, 1.01$	$1.00, 0.99$

Table 2.3: Result of the experiments: quadratic invariants inference. The examples are the same as in Table 2.2. Column 'Bounds' gives the bounds on absolute values of each variables inferred and proved by the tool whereas column 'Reachable' gives underapproximations of the maximum reachable values (obtained by random simulation) for comparison purpose. $\max |\lambda_i|$ is the maximum of modules of eigenvalues of the linear application considered, which gives an idea of 'how contractive' the linear application is.

2.7 Conclusion

In this chapter we attempted to describe a complete, yet practical, use of policy iterations to perform static analysis of programs. Policy iteration is shown to be a strong candidate to support the computation of precise post-fixpoints when over-approximating the collecting semantics of a program.

We presented the background of policy iterations and the rationale of its use, either using min-policy decreasing iterations – a kind of smart narrowing, starting from a post-fixpoint – or max-policies, performing increasing iterations. In both cases, bounds over template domains are obtained relying on numerical solvers, at each step of the computation.

We supported the use of policy iteration, as a way to replace the widening operator when it is ineffective, by addressing key points required by the technique and often left unaddressed by former works:

- the automatic computation of templates;
- handling the floating point semantics of the analyzed program;
- guaranteeing the soundness of the computation despite the possible errors of the numerical solvers used.

We believe our contribution could support a wider use of policy iterations within the abstract interpretation framework and more generally the static analysis of programs. The setting in which the current work is performed is specific: the analysis of control software, focusing on quadratic templates and using SDP solvers as optimization solvers; but it is a first step towards more extensions and a wider applicability:

- more complex templates, e.g., disjunction of quadratic forms or polynomials (for instance thanks to sum-of-squares (SOS) relaxations);
- wider class of programs analyzable precisely, e.g., complex discrete versions of controlled systems including the system (also known as plant) behavior.

Another important aspect of the approach is the capability to express and analyze more complex behaviors than just the boundedness of the considered system. The synthesized templates, even if constrained by a bound, could express high level behavior of the program. In our setting of controllers, these templates can be used to encode stability, robustness or performance properties, leading to a broader impact of static analysis when applied to critical software and systems.

Chapter 3

Verified Sum Of Squares Optimizations

The previous chapter gave us a motivation to use tools such as numerical optimization solvers, more precisely SDP solvers, to synthesize program invariants. Unfortunately, it is easy to derive incorrect results through these methods, as illustrated in Section 3.1. Thus, Section 3.2 explains how we can efficiently and automatically perform rigorous proofs of such numerical results, then Section 3.3 describes how we made this available in the Alt-Ergo SMT (Satisfiability Modulo Theories) solver [RIC18] and finally Section 3.4 explains how we made this available as a reflexive tactic in the Coq proof assistant, automatically producing fully mechanized proofs [MR17].

3.1 Motivating Examples

In this section, we illustrate through two examples the scenario where numerical SDPs give seemingly sensible solutions to simple invariant generation problems, and yet the generated invariants are not sound.

Consider the program in Figure 3.1. Does there exist an inductive invariant[1] that can be expressed as $\left\{(x_1, x_2) \in \mathbb{R}^2 \;\middle|\; p(x_1, x_2) \geq 0\right\}$ for some polynomial p? A *tractable* sufficient condition that guarantees this can be formulated using the SOS optimization approach (see Section 3.2), resulting in an SDP instance that can be solved by numerical solvers. The widely used SDPT3 [TTT03] solver reports a solution. Although all the DIMACS errors [SP] are less than 10^{-8}, not raising any suspicion, we found traces of the program that violate this purported invariant (see Figure 3.1).

As another example, we consider a program from Adjé et al. [AGM15] and the "invariant" they offer, generated with numerical solvers (Figure 3.2). Note that the purported invariant is indeed not inductive: one can find points in it whose image after one iteration of the loop body exits the invariant (Figure 3.3). Figure 3.3 also depicts an actual invariant, proved using the method in this chapter.

One could think that those inaccuracies come from the fact that the SDP solvers compute using floating-point arithmetic. However, the very fact that the interior-point algorithm they implement only provides approximate solutions, probably plays a much greater role. This means implementing an SDP solver using exact rational arithmetic, if at all possible, would only be dramatically slower but wouldn't immediately provide sound results. We provide a more thorough analysis of the sources of inaccuracies when using SDP solvers in [RVS18, §4]. The remaining of this chapter thus focuses on a-posteriori validation of solutions, as well as simple tricks to obtain such valid solutions in practice.

[1]Quite often in this document, the word "invariant" is used for inductive invariant.

```
(x1, x2) ∈ $\left\{x_1, x_2 \;\middle|\; x_1^2 + x_2^2 \leq 1.5^2\right\}$
while (1) {   // Find Inv. $p(x_1, x_2) \geq 0$
  x1 = x1 * x2;
  x2 = −x1;
}
```

$$p(x_1, x_2) := \begin{pmatrix} 1 + 2.46x_1^2 + 2.46x_2^2 - 5 \times 10^{-7}x_1^4 \\ -2.46x_1^2x_2^2 - 5 \times 10^{-7}x_2^4 \end{pmatrix}$$

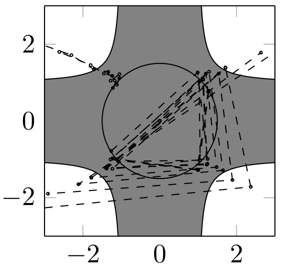


Figure 3.1: (**Left**) An example program, and "loop invariant" $p(x_1, x_2) \geq 0$ synthesized using numerical solvers. (**Right**) The claimed "invariant" and dashed lines showing violations.

```
(x1, x2) ∈ [0.9, 1.1] × [0, 0.2]
while (1) {
  pre_x1 = x1; pre_x2 = x2;
  if (x1^2 + x2^2 <= 1) {
    x1 = pre_x1^2 + pre_x2^3;
    x2 = pre_x1^3 + pre_x2^2;
  } else {
    x1 = 0.5 * pre_x1^3
              + 0.4 * pre_x2^2;
    x2 = -0.6 * pre_x1^2
              + 0.3 * pre_x2^2;
  }
}
```

$$\boxed{\begin{aligned}&2.510902467 + 0.0050x_1 + 0.0148x_2 - 3.0998x_1^2\\ &+ 0.8037x_2^3 + 3.0297x_1^3 - 2.5924x_2^2\\ &- 1.5266x_1x_2 + 1.9133x_1^2x_2 + 1.8122x_1x_2^2 - 1.6042x_1^4\\ &- 0.0512x_1^3x_2 + 4.4430x_1^2x_2^2 + 1.8926x_1x_2^3 - 0.5464x_2^4\\ &+ 0.2084x_1^5 - 0.5866x_1^4x_2 - 2.2410x_1^3x_2^2 - 1.5714x_1^2x_2^3\\ &+ 0.0890x_1x_2^4 + 0.9656x_2^5 - 0.0098x_1^6 + 0.0320x_1^5x_2\\ &+ 0.0232x_1^4x_2^2 - 0.2660x_1^3x_2^3 - 0.7746x_1^2x_2^4\\ &- 0.9200x_1x_2^5 - 0.6411x_2^6 \geq 0\end{aligned}}$$

Figure 3.2: (**Left**) An example program taken from from ADJÉ et al. [AGM15, Example 4]. (**Right**) Purported invariant at loop head synthesized using SDP solvers [AGM15].

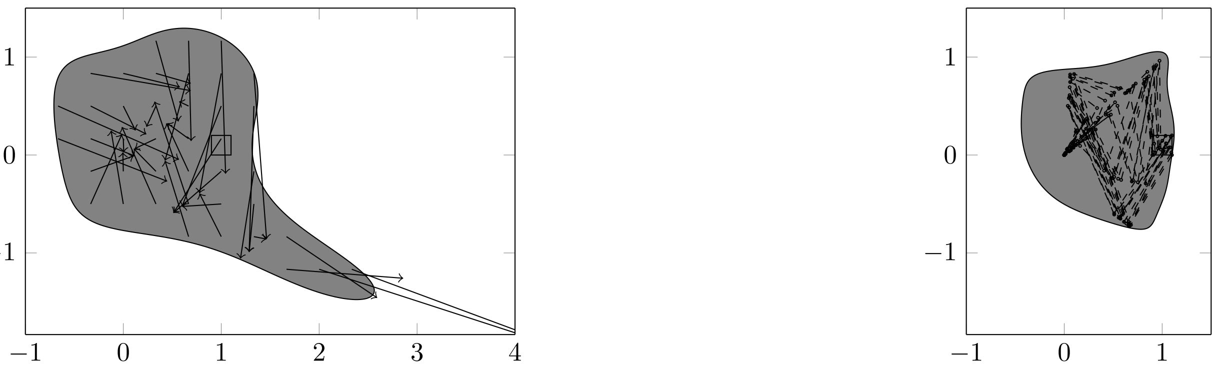


Figure 3.3: (**Left**) The candidate invariant from Figure 3.2 with arrows showing concrete transitions. The arrows leaving it are counterexamples to its inductiveness. (**Right**) The invariant of degree 8 whose soundness is proved using the approach in this chapter.

3.2 On paper

3.2.1 Sum of Squares (SOS) Programming

The sum of squares relaxation [Las01, Par03] is an incomplete but efficient way to numerically solve polynomial problems. This section aims at recalling its main ideas which are required to understand the main contribution of the chapter.

A multivariate polynomial $p \in \mathbb{R}[x]$ is said to be a sum of squares when there exist polynomials $h_i \in \mathbb{R}[x]$ such that, for all $x \in \mathbb{R}^n$,

$$p(x) = \sum_i h_i^2(x).$$

Although not all nonnegative polynomials are sum of squares, being a sum of squares is a sufficient condition to be nonnegative.

Example 8. *Considering $p(x_1, x_2) = 2x_1^4+2x_1^3x_2-x_1^2x_2^2+5x_2^4$, there exist $h_1(x_1, x_2) = \frac{1}{\sqrt{2}}\left(2x_1^2 - 3x_2^2 + x_1x_2\right)$ and $h_2(x_1, x_2) = \frac{1}{\sqrt{2}}\left(x_2^2 + 3x_1x_2\right)$ such that $p = h_1^2 + h_2^2$. This proves that for all $x_1, x_2 \in \mathbb{R}$, $p(x_1, x_2) \geq 0$.*

Any polynomial p of degree $2d$ (a nonnegative polynomial is necessarily of even degree) can be written as a quadratic form in the vector of all monomials of degree less or equal d:

$$p(x) = z^T Q\, z \tag{3.1}$$

where $z = \left[1, x_1, \dots, x_n, x_1x_2, \dots, x_n^d\right]$ and Q is a constant symmetric matrix.

Remark 7. *If the polynomial p is* homogeneous*, that is if all its monomials have the same degree $2d$ (as in Example 8), then only monomials of degree* exactly *d are required in the vector z.*

Example 9. *For $p(x_1, x_2) = 2x_1^4 + 2x_1^3x_2 - x_1^2x_2^2 + 5x_2^4$, according to the above Remark, we can use the vector* *of monomials $z = \left[x_1^2, x_2^2, x_1x_2\right]^T$. We then have*

$$\begin{aligned}
p(x_1,x_2) &= 2x_1^4 + 2x_1^3x_2 - x_1^2x_2^2 + 5x_2^4\\
&= \begin{bmatrix} x_1^2\\ x_2^2\\ x_1x_2\end{bmatrix}^T \begin{bmatrix} q_{11} & q_{12} & q_{13}\\ q_{12} & q_{22} & q_{23}\\ q_{13} & q_{23} & q_{33}\end{bmatrix} \begin{bmatrix} x_1^2\\ x_2^2\\ x_1x_2\end{bmatrix}\\
&= q_{11}x_1^4 + 2q_{13}x_1^3x_2 + (q_{33}+2q_{12})x_1^2x_2^2 + 2q_{23}x_1x_2^3 + q_{22}x_2^4.
\end{aligned}$$

Thus $q_{11} = 2$, $2q_{13} = 2$, $q_{33} + 2q_{12} = -1$, $2q_{23} = 0$ and $q_{22} = 5$. Two possible examples for the matrix Q are shown below:

$$Q = \begin{bmatrix} 2 & 1 & 1\\ 1 & 5 & 0\\ 1 & 0 & -3\end{bmatrix}, \qquad Q' = \begin{bmatrix} 2 & -3 & 1\\ -3 & 5 & 0\\ 1 & 0 & 5\end{bmatrix}.$$

The polynomial p is then a sum of squares if and only if there exists a positive semidefinite matrix Q satisfying (3.1). A matrix Q is said positive semidefinite when, for all vectors x, $x^T Q\, x \geq 0$. This will be denoted by $Q \succeq 0$.

Example 10. *In the previous example, the first matrix Q is not positive semidefinite (for $x = [0,0,1]^T$, $x^T Q\, x = -3$). In contrast, the second matrix Q' is positive semidefinite as it can be written $Q' = L^T L$ with*

$$L = \frac{1}{\sqrt{2}} \begin{bmatrix} 2 & -3 & 1\\ 0 & 1 & 3\end{bmatrix}$$

(then, for all x, $x^T Q x = (Lx)^T(Lx) = \|Lx\|_2^2 \geq 0$). This gives the sum of squares decomposition of Example 8: $p(x_1,x_2) = \frac{1}{2}(2x_1^2 - 3x_2^2 + x_1x_2)^2 + \frac{1}{2}(x_2^2 + 3x_1x_2)^2$.

3.2.2 Semidefinite Programming (SDP)

Given symmetric matrices $C, A_1, \ldots, A_m \in \mathbb{R}^{s\times s}$ and scalars $a_1, \ldots, a_m \in \mathbb{R}$, the following optimization problem is called *semidefinite programming*

$$\begin{array}{ll}
\text{minimize} & \operatorname{tr}(CQ)\\
\text{subject to} & \operatorname{tr}(A_1 Q) = a_1\\
 & \quad\vdots\\
 & \operatorname{tr}(A_m Q) = a_m\\
 & Q \succeq 0
\end{array} \tag{3.2}$$

with symmetric matrix $Q \in \mathbb{R}^{s\times s}$ as variable and where $\operatorname{tr}(M) = \sum_i M_{i,i}$ denotes the trace of the matrix M.

Remark 8. *Since the matrices are symmetric, $\operatorname{tr}(AQ) = \operatorname{tr}(A^TQ) = \sum_{i,j} A_{i,j}Q_{i,j}$. The constraints $\operatorname{tr}(AQ) = a$ are then affine constraints between the elements of the matrix Q, the variable $Q_{i,j}$ being assigned to the coefficient $A_{i,j}$.*

Semidefinite programming is a convex optimization problem for which there exist efficient numerical solvers [BV04, VB96].

As we have just seen in Section 3.2.1, existence of a sum of squares decomposition amounts to existence of a positive semidefinite matrix satisfying a set of affine constraints, that is a solution of a semidefinite program. Thus, semidefinite programming provides an efficient way to numerically solve problems involving polynomial inequalities, by relaxing them as sum of squares constraints.

Example 11. *The affine constraints computed in Example 9 can be encoded as a semidefinite program with the following constraints:*

$$\operatorname{tr}\left(\begin{bmatrix} 1 & 0 & 0\\ 0 & 0 & 0\\ 0 & 0 & 0\end{bmatrix} Q\right) = 2, \quad \operatorname{tr}\left(\begin{bmatrix} 0 & 0 & 1\\ 0 & 0 & 0\\ 1 & 0 & 0\end{bmatrix} Q\right) = 2, \quad \operatorname{tr}\left(\begin{bmatrix} 0 & 1 & 0\\ 1 & 0 & 0\\ 0 & 0 & 1\end{bmatrix} Q\right) = -1,$$

$$\operatorname{tr}\left(\begin{bmatrix} 0 & 0 & 0\\ 0 & 0 & 1\\ 0 & 1 & 0\end{bmatrix} Q\right) = 0, \quad \operatorname{tr}\left(\begin{bmatrix} 0 & 0 & 0\\ 0 & 1 & 0\\ 0 & 0 & 0\end{bmatrix} Q\right) = 5.$$

3.2.3 Parameterized Problems

Up to now, we have only seen how to check whether a given *fixed* polynomial p is sum of squares (which implies its nonnegativeness). One of the great strength of SOS programming is to enable to solve problems with unknown polynomials.

An unknown polynomial $p \in \mathbb{R}[x]$ with n variables and of degree d can be written

$$p = \sum_{\alpha_1+\dots+\alpha_n \leq d} p_\alpha x_1^{\alpha_1} \dots x_n^{\alpha_n}$$

where the $p_\alpha \in \mathbb{R}$ are scalar parameters. The p_α then just translate to additional variables in the resulting SDP problem. This enables to relax polynomial problems with polynomial constraints.

Example 12. *Given two polynomials p and q, to prove that $p(x) \geq 0$ whenever $q(x) \geq 0$ (said otherwise: $\forall x, q(x) \geq 0 \implies p(x) \geq 0$), one can exhibit a polynomial σ such that*

$$p - \sigma q \geq 0 \wedge \sigma \geq 0.$$

Indeed, for any x, if $q(x) \geq 0$ then $p(x) \geq \sigma(x)\, q(x) \geq 0$. Thus, the nonnegativity constraints can be relaxed to sum of squares constraints and the problem can be encoded as some SDP program (for some degree for σ).

The reader interested in more details is referred to the literature [Las09, Löf09].

3.2.4 Approximate Solutions from SDP Solvers

In practice, the matrix Q returned by SDP solvers upon solving an SDP problem (3.2) does not precisely satisfy the equality constraints. Therefore, although the SDP solver returns a positive answer for a SOS program, this does not translate to a valid proof that a given polynomial is SOS. This section details an incomplete but efficient validation method to address this issue.

Most SDP solvers start from some $Q \succeq 0$ not satisfying the equality constraints (for instance the identity matrix) and iteratively modify it in order to reduce the distance between $\mathrm{tr}(A_i Q)$ and a_i while keeping Q positive semidefinite. This process is stopped when this distance is deemed small enough.

Therefore, we do not obtain a Q satisfying $\mathrm{tr}(A_i Q) = a_i$ but rather $\mathrm{tr}(A_i Q) = a_i + d_i$ for some small[2] δ such that $|d_i| \leq \delta$ for all i. This has a simple translation in terms of our original SOS problem. The equality constraints $\mathrm{tr}(A_i Q) = a_i$ correspond to equality between corresponding monomials of the polynomials p and $z^T Q\, z$. As a result, there exists a matrix E such that for all i, j, $|E_{i,j}| \leq \delta$, with

$$p = z^T (Q + E)\, z. \tag{3.3}$$

Proof scheme (`soscheck_correct` in our Coq code). Just take the matrix E with coefficients

$$E_{i,j} := \frac{(p - z^T Q z)[z_i\, z_j]}{\#\{(i', j') \mid z_{i'}\, z_{j'} = z_i\, z_j\}}$$

where $p[m]$ stands for the coefficient of monomial m in polynomial p and $\#S$ denotes the cardinal of the set S. Then, each $(p - z^T Q z)[z_i\, z_j]$ corresponds to some d_i and $\#\{(i', j') \mid z_{i'}\, z_{j'} = z_i\, z_j\} \geq 1$, hence $|E_{i,j}| \leq \delta$. □

It is worth noting that we cannot trust the value of δ reported by the solver, as it is just computed in floating-point arithmetic. However, it is easy to recompute as $\delta := \max_i |\mathrm{tr}(A_i Q) - a_i|$.

3.2.5 A Validation Method

Our goal is now to check that $Q + E \succeq 0$ which would prove, along with (3.3) that p is SOS (and then nonnegative). The matrix Q is a floating-point matrix provided by the SDP solver and the above proof scheme provides a way to compute E, hence also $Q + E$. However, we don't want to exactly compute $Q + E$ as, unlike Q, there is no guarantee for it to contain only floating-point values. We will rather attempt to prove the stronger result that for any matrix M whose elements are bounded by ϵ, the inequality $Q + M \succeq 0$ holds. Figure 3.4 gives a geometrical intuition of this.

The following property then provides a sufficient condition.

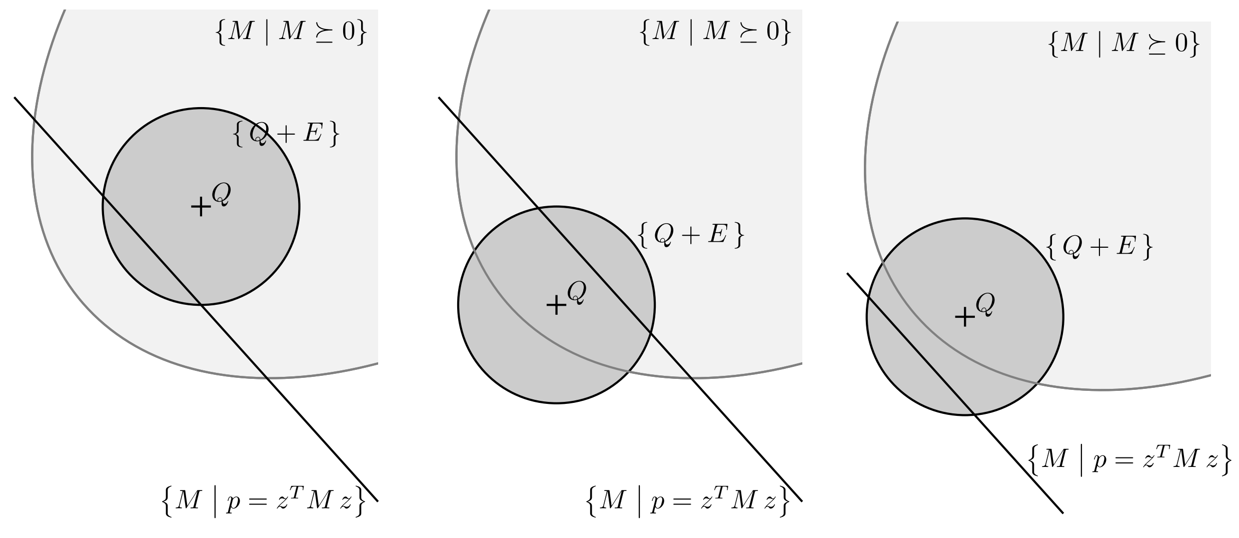
Property 1 (`posdef_check_itv_correct` in Coq). *For any $Q \in \mathbb{R}^{s \times s}$ and $\delta \in \mathbb{R}$, if*

$$Q - s\delta I \succeq 0,$$

then for any $E \in \{M \mid \forall i, j, |M_{i,j}| \leq \delta\}$,

$$Q + E \succeq 0.$$

[2]Typically, $\delta \sim 10^{-8}$ when the order of magnitude of the a_i is 1.



(a) p is proved SOS. (b) Cannot conclude. (c) Cannot conclude.

Figure 3.4: Given a polynomial p, all matrices M such that $p = z^T M\, z$ give an expression of p in the monomial basis z. Among these expressions, positive semidefinite matrices $M \succeq 0$ correspond to sum of squares. To prove that p is SOS, we then need to prove that the subspace $\{M \mid p = z^T M\, z\}$ (dark diagonal line on the figure) intersects the positive semidefinite cone $\{M \mid M \succeq 0\}$ (light grey rounded shape). The SDP solver returns a matrix Q close to the subspace, i.e., such that the ball $\{Q + E \mid \forall i, j, |E_{i,j}| \leq \delta\}$ (darker grey disc, denoted $\{\, Q + E\, \}$ on the figure) intersects it. Thus, proving that this ball is included in the cone, as on (a), enables to conclude. The proof can also fail, either because Q is too close to the border of the cone (b) or because p is simply not SOS (c).

```
$R := 0$;
for $j$ from $1$ to $n$ do
  for $i$ from $1$ to $j-1$ do
    $R_{i,j} := \left(M_{i,j} - \sum_{k=1}^{i-1} R_{k,i} R_{k,j}\right) / R_{i,i}$;
  od
  $R_{j,j} := \sqrt{M_{j,j} - \sum_{k=1}^{j-1} {R_{k,j}}^2}$;
od
```

Figure 3.5: From a matrix M, the Cholesky decomposition attempts to compute a matrix R such that $M = R^T R$. The algorithm succeeds if and only if $M \succ 0$, assuming exact real arithmetic.

Thus, we are left having to prove that a given matrix $Q - s\delta I$ is positive semidefinite instead of Q itself. We use a Cholesky decomposition to do so. Given a matrix M, if $M \succeq 0$, the Cholesky decomposition algorithm, given on Figure 3.5, computes a matrix R such that $M = R^T R$ which proves[3] that $M \succeq 0$. If M is not positive semidefinite, the Cholesky decomposition fails by attempting to take the square root of a negative value. The execution of the algorithm requires $\Theta(s^3)$ arithmetic operations.

However, for the sake of efficiency, a floating point Cholesky decomposition is used, which prevents the exact equality $M = R^T R$. The following theorem gives an incomplete but efficient method that allows us to conclude the positive definiteness of a matrix M using an unreliable floating point Cholesky decomposition of a slightly modified floating point matrix $\tilde{M}$.

Theorem 3. *[Rum06, Corollary 2.4] (`corollary_2_4_with_c_upper_bound` in Coq) Assume a floating-point format $\mathbb{F}$ with relative error ε and absolute error η. For $M \in \mathbb{R}^{s \times s}$, let c be*

$$\frac{(s+1)\varepsilon}{1-(s+1)\varepsilon}\operatorname{tr}(M) + 4s\left(2(s+1) + \max_i M_{i,i}\right)\eta.$$

If the floating-point Cholesky decomposition of $M - cI$ succeeds (i.e., without taking the square root of a negative value), then the Cholesky decomposition of M with real numbers would succeed as well, which implies that M is positive definite.

[3]For any x, $x^T M\, x = (Rx)^T (Rx) = \|Rx\|_2^2 \geq 0$.

So according to this theorem, the positive definiteness of $Q - s\delta I$ can be established by first computing an upper bound[4] of the constant c, subtracting it from the diagonal of our matrix and applying the floating-point Cholesky decomposition.

Remark 9. *ε and η are constants characterizing the errors due to normalized and denormalized numbers in the floating-point format $\mathbb{F}$. For instance, for the IEEE754 [IEE08] binary64 format with rounding to nearest[5], $\varepsilon = 2^{-53}$ ($\simeq 10^{-16}$) and $\eta = 2^{-1075}$ ($\simeq 10^{-323}$). Thus, for typical values ($s \leq 1000$ and elements of M of order of magnitude 1), $c \leq 10^{-10}$. This is negligible in front of $s\delta \sim 1000 \times 10^{-8} = 10^{-5}$ which means that the incompleteness of this positive-definiteness check is not an issue in practice.*

To sum up, we designed the following verification method to prove that a given polynomial p is SOS. Given the approximate solution Q returned by an SDP solver for the problem $p = z^T Q\, z, Q \succeq 0$:

1. Check that all monomials of the polynomial p are in the monomial base $z^T z$.
2. Bound the difference δ between the corresponding coefficients of p and $z^T Q\, z$.
3. Check that $Q - s\delta I \succeq 0$.

Step 1 is a purely symbolic computation, step 2 can be achieved using floating-point interval arithmetic[6] in $\Theta(s^2)$ operations (the size of Q) and step 3 can be done in $\Theta(s^3)$ floating-point operations thanks to the above theorem. Thus, the whole validation method takes $\Theta(s^3)$ floating-point operations which in practice constitutes a very small overhead compared to the time taken by the SDP solver to compute Q.

We formally proved all results in Sections 3.2.4 and 3.2.5 with the Coq proof assistant. The proofs can be found in the ValidSDP library[7], codeveloped with Érik Martin-Dorel. The proofs rely on the MathComp library [GMT08] for matrices, on the Flocq library [BM11] for the formalization of floating-point arithmetic and on MathComp-Multinomials [BBRS16] for multivariate polynomials. See Section 3.4 for more details on the ValidSDP library.

3.2.6 Making it Work in Practice

As seen on Figure 3.4, the validation method from Section 3.2.5 can easily fail when the matrix Q returned by the SDP solver is close to the border of the positive semidefinite cone. In practice, it is common to use some optimization objective function, which tends to push the solution Q to the border of the positive semidefinite cone. We thus need a solution to ensure that Q is well inside the cone, so that $Q - s\delta I \succeq 0$ holds. This can be obtained by asking the SDP solver for a matrix Q satisfying $Q - s\delta I \succeq 0$ rather than just $Q \succeq 0$ in the first place. An intuition is given on Figure 3.6.

However, according to the method in Section 3.2.5, it seems that δ is computed from the value of Q returned by the solver, so we don't know its value before running the solver. In fact, δ being the distance between Q and the subspace $\left\{M \mid p = z^T M\, z\right\}$ it is one of the measures of the quality of the numerical solution computed by the solver. More precisely, solvers implement interior-point algorithms that start from some matrix inside the cone and iterate steps to decrease such measures, while remaining inside the cone. They stop these iterations when the measures drop below a given predetermined threshold, called *stopping criterion*. Knowing the stopping criterion of the solver, we thus know beforehand an overapproximation of δ.

Finally, as noted in Remark 9, the constant c from Theorem 3 is orders of magnitude smaller than δ, hence in some sense already accounted for in our overapproximation of δ.

3.3 Inside the Alt-Ergo SMT Solver

This section mostly sums ups our TACAS 2018 paper [RIC18], that makes the method of the previous Section 3.2 available inside the Alt-Ergo SMT solver. This work was done in collaboration with Mohamed Iguernlala and Sylvain Conchon, authors of Alt-Ergo. The authors would also like to thank Rémi Delmas for insightful discussions and technical help, particularly with the dReal solver.

Systems of nonlinear polynomial constraints over the reals are known to be solvable since Tarski proved that the first-order theory of the real numbers is decidable, by providing a quantifier elimination procedure. This procedure has then been much improved, particularly with the cylindrical algebraic decomposition. Unfortunately, its doubly exponential complexity remains a serious limit to its scalability. It is now integrated into SMT solvers [JdM12]. Although it demonstrates very good practical results, symbolic quantifier

[4]For instance using floating-point arithmetic with directed rounding.

[5]Type `double` in C.

[6]Although we currently use rational arithmetic for this step in our Coq development, as it appeared cheap enough thanks to the reasonably small amount of computation involved.

[7]Available at `https://github.com/validsdp/validsdp`

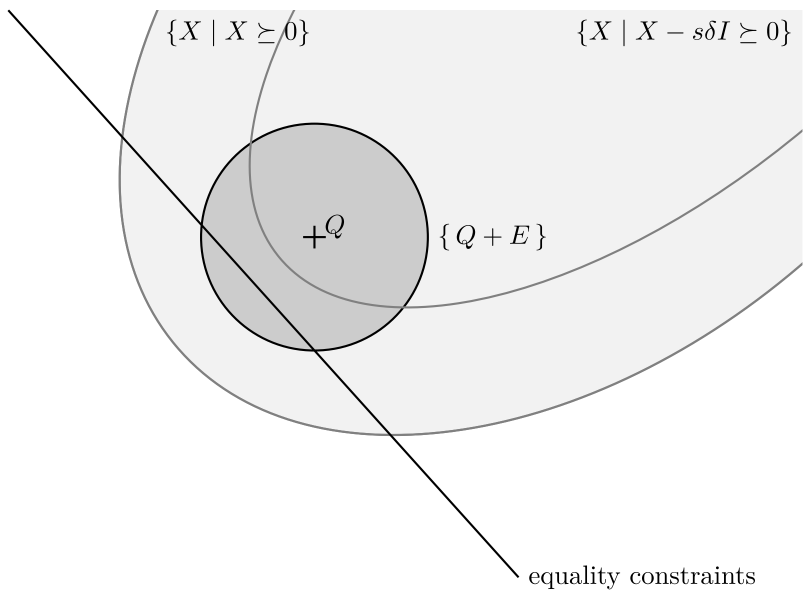


Figure 3.6: In order to ensure that the solution Q returned by the SDP solver is not too close to the border of the positive semidefinite cone, we ask the solver for a solution inside the padded cone $\{X \mid X - s\delta I \succeq 0\}$.

elimination seems to remain an obstacle to scalability on some problems. In some cases, branch and bound with interval arithmetic constitutes an interesting alternative [GAC12]. We show in this section how the method introduced in previous Section 3.2 can be used to design a sound semi-decision procedure that outperforms symbolic and interval-arithmetic methods on problems of practical interest.

A noticeable characteristic of the algorithms implemented in the SDP solvers is to only compute approximate solutions. We explain this by making a comparison with linear programming. There are two competitive methods to optimize a linear objective under linear constraints: the interior point and the simplex algorithms. The interior point algorithm starts from some initial point and performs steps towards an optimal value. These iterations converge to the optimum but not in finitely many steps and have to be stopped at some point, yielding an approximate answer. In contrast, the simplex algorithm exploits the fact that the feasible set is a polyhedron and that the optimum is achieved on one of its vertices. The number of vertices being finite, the optimum can be exactly reached after finitely many iterations. Unfortunately, this nice property does not hold for spectrahedra, the equivalent of polyhedra for semidefinite programming. Thus, all semidefinite programming solvers are based on the interior-point algorithm, or a variant thereof. This is illustrated on Figure 3.7.

To illustrate the consequences of these approximate solutions, consider the proof of $e \leq c$ with e a complicated ground expression and c a constant. $e \leq c$ can be proved by exactly computing e, giving a constant c', and checking that $c' \leq c$. However, if e is only approximately computed: $e \in [c' - \epsilon, c' + \epsilon]$, this is conclusive only when $c' + \epsilon \leq c$. In particular, if e is equal to c, an exact computation is required. In the SDP programming setting, this corresponds to problems whose feasible space has an empty interior, also called nonstrictly feasible problems, as illustrated on Figure 3.8. This inability to prove inequalities that are not satisfied with some margin is a well known property of numerical verification methods [Rum10] which can then be seen as a trade-off between completeness and computation cost.

The main point of this section is that, despite their incompleteness, numerical verification methods remain an interesting option when they enable to practically solve problems for which other methods offer an untractable complexity. Our contributions are:

(1) a comparison of two sound semi-decision procedures for systems of nonlinear constraints, which rely on off-the-shelf numerical optimization solvers,

(2) an integration of these procedures in the Alt-Ergo SMT solver,

(3) an experimental evaluation of our approach on a set of benchmarks coming from various application domains.

The rest of this section is organized as follows: Section 3.3.1 gives a practical example of a polynomial problem, coming from control-command program verification, better handled by numerical methods. In Section 3.3.3, we offer another method (after the one of Section 3.2.5) to derive sound solutions to polynomial problems from approximate answers of semidefinite programming solvers. Section 3.3.4 provides some

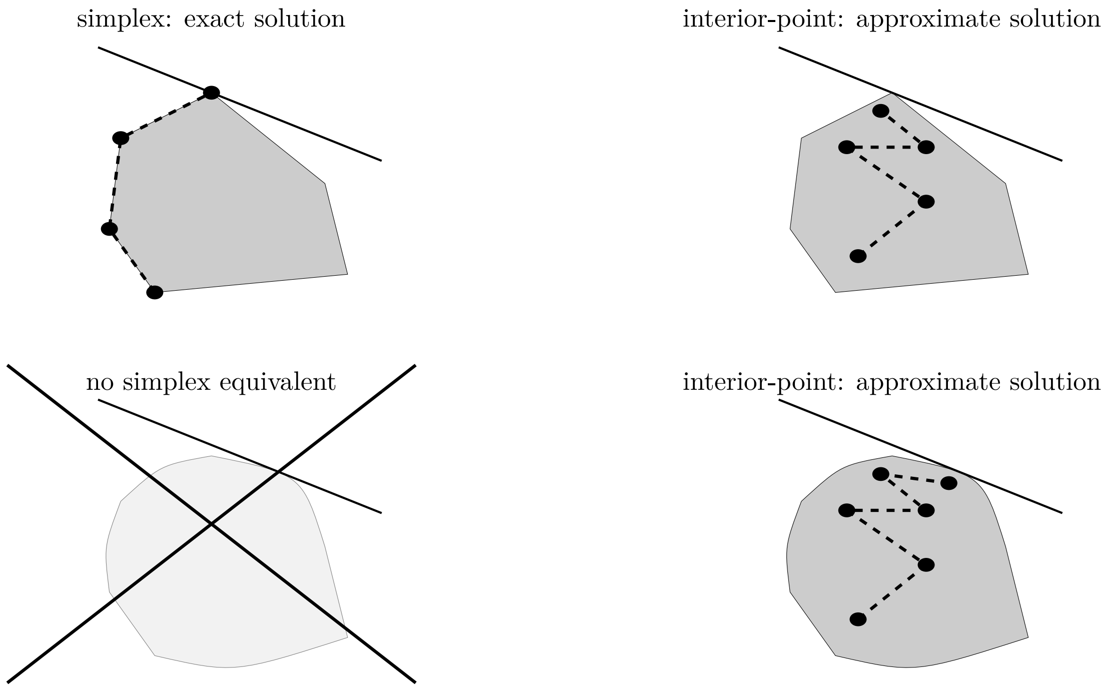

simplex: exact solution
interior-point: approximate solution
no simplex equivalent
interior-point: approximate solution

Figure 3.7: Comparison of SDP programming (bottom) with the linear case (top).

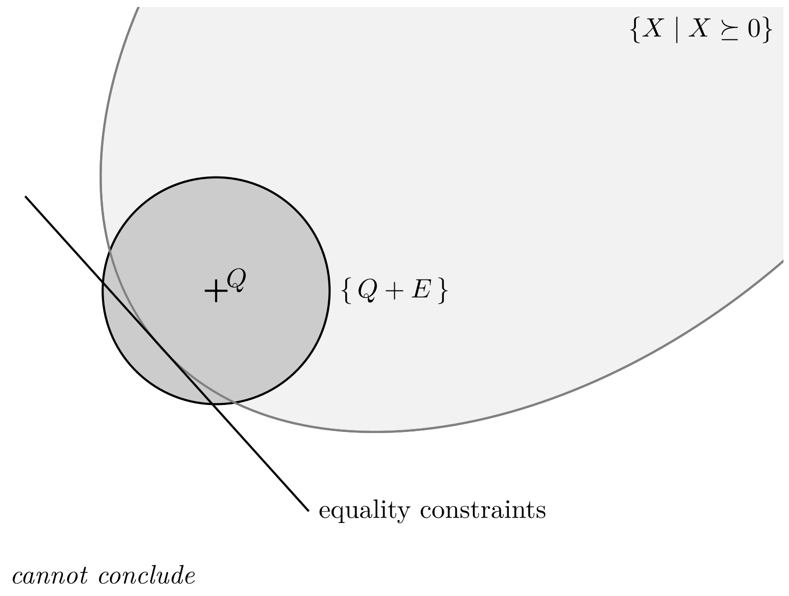


Figure 3.8: SDP problem with empty interior: the feasible space is limited to the intersection point between the equality constraints and the cone, which has an empty interior. Numerical methods then can’t conclude, as the $\{\, Q + E \,\}$ ball cannot be included in the positive semidefinite cone. Compare with Figure 3.4.

```
typedef struct { double x0, x1, x2; } state;
/*@ predicate inv(state *s) = 6.04 * s->x0 * s->x0 - 9.65 * s->x0 * s->x1
  @   - 2.26 * s->x0 * s->x2 + 11.36 * s->x1 * s->x1
  @   + 2.67 * s->x1 * s->x2 + 3.76 * s->x2 * s->x2 <= 1;
  @ requires \valid(s) && inv(s) && -1 <= in0 <= 1;
  @ ensures inv(s); */
void step(state *s, double in0) {
  double pre_x0 = s->x0, pre_x1 = s->x1, pre_x2 = s->x2;
  s->x0 = 0.9379 * pre_x0 - 0.0381 * pre_x1 - 0.0414 * pre_x2 + 0.0237 * in0;
  s->x1 = -0.0404 * pre_x0 + 0.968 * pre_x1 - 0.0179 * pre_x2 + 0.0143 * in0;
  s->x2 = 0.0142 * pre_x0 - 0.0197 * pre_x1 + 0.9823 * pre_x2 + 0.0077 * in0;
}
```

Figure 3.9: Example of a typical control-command code in C.

implementation details and Section 3.3.5 discusses experimental results. Finally, Section 3.3.6 presents related work and concludes.

3.3.1 Example: Control-Command Program Verification

Control-command programs usually iterate linear assignments periodically over time. These assignments take into account a measure (via some *sensor*) of the state of the physical system to control (called *plant* by control theorists) to update an internal state and eventually output orders back to the physical system (through some *actuator*). Figure 3.9 gives an example of such an update, `in0` being the input and `s` the internal state. The comments beginning by @ in the example are annotations in the ACSL language [CKK+12]. They specify that, before the execution of the function (`requires`), the pointer `s` must be valid, satisfying the predicate `inv` and $|\texttt{in0}| \leq 1$ must hold. Under these hypotheses, `s` still satisfies `inv` after executing the function (`ensures`).

To prove that the internal state remains bounded over any execution of the system, a quadratic polynomial[8] can be used as invariant[9]. Checking the validity of these invariants then leads to arithmetic verification conditions (VCs) involving quadratic polynomials. Such VCs can for instance be generated from the program of Figure 3.9 by the Frama-C/Why3 program verification toolchain [CKK+12, FP13]. Unfortunately, proving the validity of these VCs seem out of reach for current state-of-the-art SMT solvers. For instance, although Z3 [dMB08] can solve smaller examples with just two internal state variables in a matter of seconds, it ran for a few days on the three internal state variable example of Figure 3.9 without reaching a conclusion[10]. In contrast, our prototype can prove it in a fraction of second, as well as other examples with up to a dozen variables. This is illustrated on Figure 3.10.

Verification of control-command programs is a good candidate for numerical methods. These systems are designed to be robust to many small errors, which means that the verified properties are usually satisfied with some margin. Thus, the incompleteness of numerical methods is not an issue for this kind of problems.

3.3.2 Emptiness of Semi-algebraic Sets

Our goal is to prove that conjunctions of polynomial inequalities are unsatisfiable, that is, given some polynomials with real coefficients $p_1, \dots, p_m \in \mathbb{R}[x]$, we want to prove that there does not exist any assignment for the n variables $x_1, \dots, x_n \in \mathbb{R}^n$ such that all inequalities $p_1(x_1, \dots, x_n) \geq 0, \dots, p_m(x_1, \dots, x_n) \geq 0$ hold simultaneously. In the rest of this paper, the notation $p \geq 0$ (resp. $p > 0$) means that for all $x \in \mathbb{R}^n$, $p(x) \geq 0$ (resp. $p(x) > 0$).

Theorem 4. *If there exist polynomials $r_i \in \mathbb{R}[x]$ such that*

$$-\sum_i r_i\, p_i > 0 \quad \textit{and} \quad \forall i, r_i \geq 0 \tag{3.4}$$

then the conjunction $\bigwedge_i p_i \geq 0$ is unsatisfiable[11].

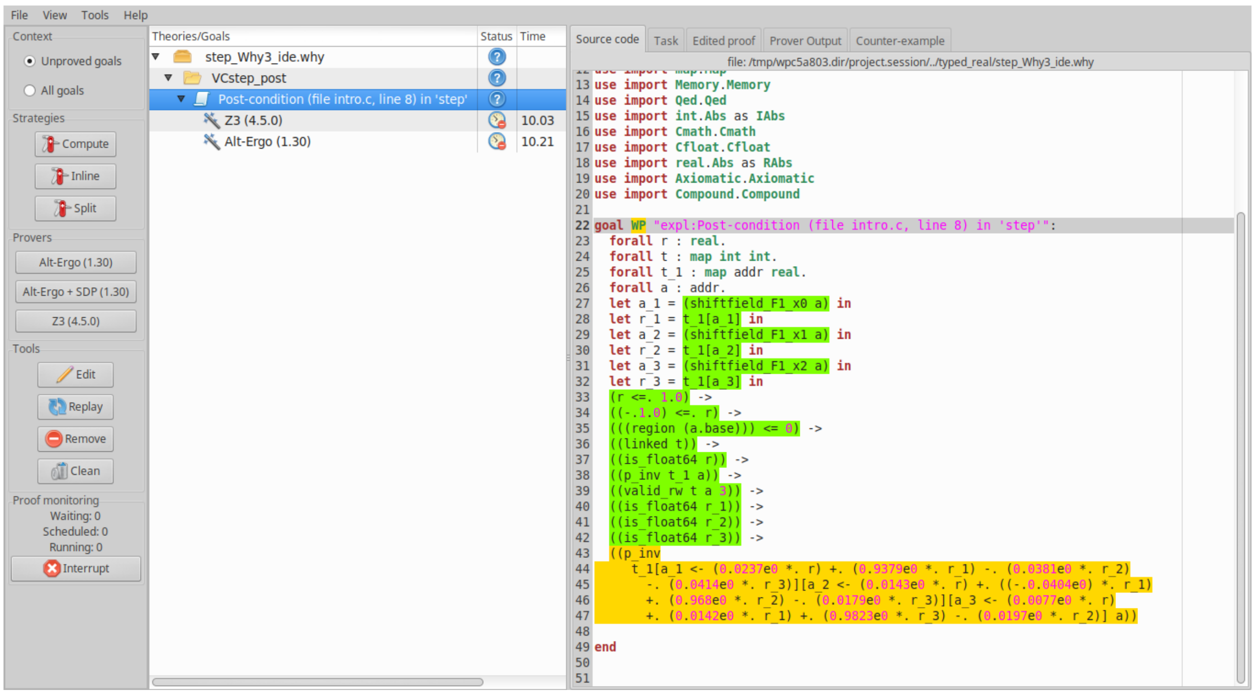
Proof. Assume there exist $x \in \mathbb{R}^n$ such that for all i, $p_i(x) \geq 0$. Then, since $r_i \geq 0$, we have $r_i(x)\, p_i(x) \geq 0$ hence $\left(\sum_i r_i\, p_i\right)(x) \geq 0$ which contradicts $-\sum_i r_i\, p_i > 0$. □

[8] For instance, the three variables polynomial in *inv* in Figure 3.9.

[9] Control theorists call these invariants sublevel sets of a quadratic Lyapunov function. Such functions exist for linear systems if and only if they do not diverge.

[10] This is the case even on a simplified version with just arithmetic constructs, i.e., expurgated of all the reasoning about pointers and the C memory model.

[11] Or, with different words, the semi-algebraic set $\{x \in \mathbb{R}^n \mid \forall i, p_i(x) \geq 0\}$ is empty.



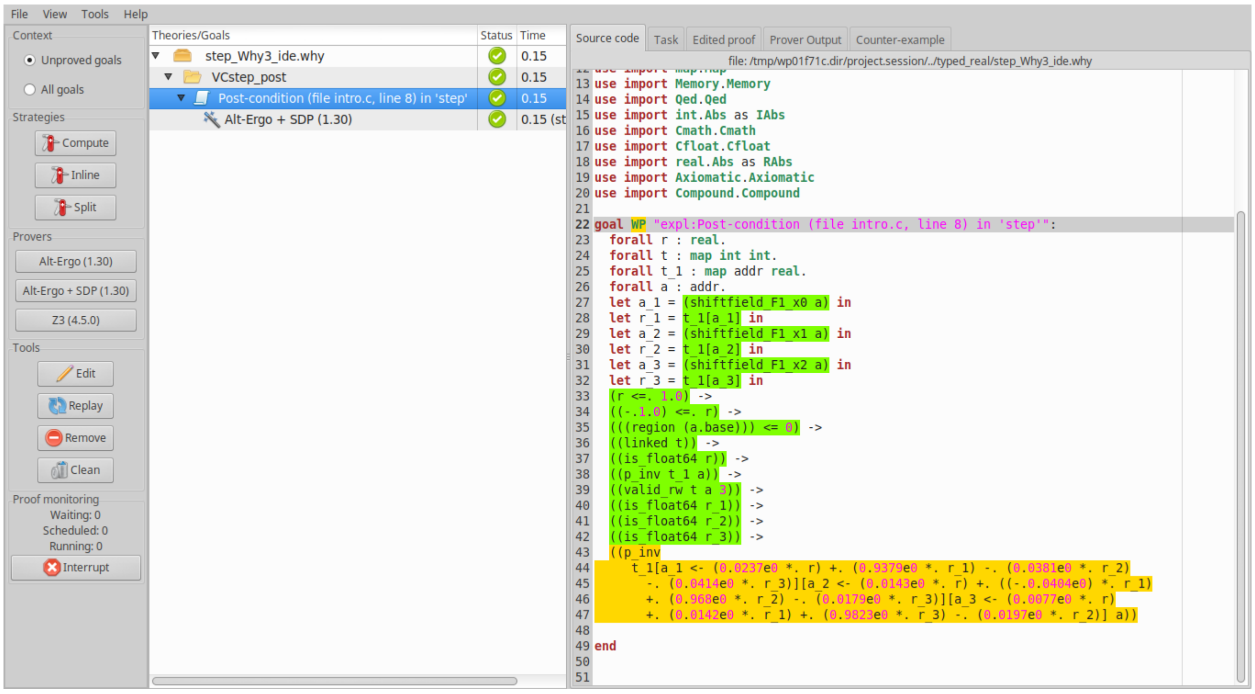
Theories/Goals	Status	Time
step_Why3_ide.why	?	
VCstep_post	?	
Post-condition (file intro.c, line 8) in 'step'	?	
Z3 (4.5.0)		10.03
Alt-Ergo (1.30)		10.21

file: /tmp/wpc5a803.dir/project.session/../typed_real/step_Why3_ide.why

```
use import Memory.Memory
use import Qed.Qed
use import int.Abs as IAbs
use import Cmath.Cmath
use import Cfloat.Cfloat
use import real.Abs as RAbs
use import Axiomatic.Axiomatic
use import Compound.Compound

goal WP "expl:Post-condition (file intro.c, line 8) in 'step'":
  forall r : real.
  forall t : map int int.
  forall t_1 : map addr real.
  forall a : addr.
  let a_1 = (shiftfield_F1_x0 a) in
  let r_1 = t_1[a_1] in
  let a_2 = (shiftfield_F1_x1 a) in
  let r_2 = t_1[a_2] in
  let a_3 = (shiftfield_F1_x2 a) in
  let r_3 = t_1[a_3] in
  (r <=. 1.0) ->
  ((-.1.0) <=. r) ->
  (((region (a.base))) <= 0) ->
  ((linked t)) ->
  ((is_float64 r)) ->
  ((p_inv t_1 a)) ->
  ((valid_rw t a 3)) ->
  ((is_float64 r_1)) ->
  ((is_float64 r_2)) ->
  ((is_float64 r_3)) ->
  ((p_inv
     t_1[a_1 <- (0.0237e0 *. r) +. (0.9379e0 *. r_1) -. (0.0381e0 *. r_2)
       -. (0.0414e0 *. r_3)][a_2 <- (0.0143e0 *. r) +. ((-.0.0404e0) *. r_1)
       +. (0.968e0 *. r_2) -. (0.0179e0 *. r_3)][a_3 <- (0.0077e0 *. r)
       +. (0.0142e0 *. r_1) +. (0.9823e0 *. r_3) -. (0.0197e0 *. r_2)] a))

end
```



Theories/Goals	Status	Time
step_Why3_ide.why	✓	0.15
VCstep_post	✓	0.15
Post-condition (file intro.c, line 8) in 'step'	✓	0.15
Alt-Ergo + SDP (1.30)	✓	0.15 (st

file: /tmp/wp01f71c.dir/project.session/../typed_real/step_Why3_ide.why

```
use import Memory.Memory
use import Qed.Qed
use import int.Abs as IAbs
use import Cmath.Cmath
use import Cfloat.Cfloat
use import real.Abs as RAbs
use import Axiomatic.Axiomatic
use import Compound.Compound

goal WP "expl:Post-condition (file intro.c, line 8) in 'step'":
  forall r : real.
  forall t : map int int.
  forall t_1 : map addr real.
  forall a : addr.
  let a_1 = (shiftfield_F1_x0 a) in
  let r_1 = t_1[a_1] in
  let a_2 = (shiftfield_F1_x1 a) in
  let r_2 = t_1[a_2] in
  let a_3 = (shiftfield_F1_x2 a) in
  let r_3 = t_1[a_3] in
  (r <=. 1.0) ->
  ((-.1.0) <=. r) ->
  (((region (a.base))) <= 0) ->
  ((linked t)) ->
  ((is_float64 r)) ->
  ((p_inv t_1 a)) ->
  ((valid_rw t a 3)) ->
  ((is_float64 r_1)) ->
  ((is_float64 r_2)) ->
  ((is_float64 r_3)) ->
  ((p_inv
     t_1[a_1 <- (0.0237e0 *. r) +. (0.9379e0 *. r_1) -. (0.0381e0 *. r_2)
       -. (0.0414e0 *. r_3)][a_2 <- (0.0143e0 *. r) +. ((-.0.0404e0) *. r_1)
       +. (0.968e0 *. r_2) -. (0.0179e0 *. r_3)][a_3 <- (0.0077e0 *. r)
       +. (0.0142e0 *. r_1) +. (0.9823e0 *. r_3) -. (0.0197e0 *. r_2)] a))

end
```

Figure 3.10: SMT solvers fail on main verification condition for the code of Figure 3.9 (top), while our modified solver succeeds in a fraction of second (bottom).

In fact, under some hypotheses[12] on the p_i, the condition (3.4) is not only sufficient but also necessary, as stated by the Putinar's Positivstellensatz [Las09, §2.5.1]. Unfortunately, no practical bound is known on the degrees of the polynomials r_i. In our prototype, we restrict the degrees of each r_i to[13] $d - \deg(p_i)$ where $d := \max_i(\deg(p_i))$, so that $\sum_i r_i\, p_i$ is a polynomial of degree d. This is a first source of incompleteness, although benchmarks show that it already enables to solve many interesting problems.

3.3.3 Rounding to an Exact Rational Solution

The most common solution to verify results of SOS programming is to round the output of the SDP solver to an exact rational solution [Har07, KLYZ12, MC11].

To sum up, the matrix Q returned by the SDP solver is first projected to the subspace $\left\{M \mid p = z^T M\, z\right\}$ then all its entries are rounded to rationals with small denominators (first integers, then multiples of $\frac{1}{2}, \frac{1}{3}, \ldots$)[14]. For each rounding, positive semidefiniteness of the resulting matrix Q is tested. The rationale behind this choice is that problems involving only simple rational coefficients can reasonably be expected to admit simple rational solutions[15].

Using exact solutions potentially enables to verify SDP problems with empty relative interiors. This means the ability to prove inequalities without margin, to distinguish strict and nonstrict inequalities and even to handle (dis)equalities. All of this nevertheless requires a different relaxation scheme than (3.4).

Example 13. *To prove $x_1 \geq 0 \wedge x_2 \geq 0 \wedge q_1 = 0 \wedge q_2 = 0 \wedge p > 0$ unsatisfiable, with $q_1 := x_1^2 + x_2^2 - x_3^2 - x_4^2 - 2$, $q_2 := x_1 x_3 + x_2 x_4$ and $p := x_3 x_4 - x_1 x_2$, one can look for polynomials l_1, l_2 and SOS polynomials $s_1, \ldots, s_8$ such that $l_1 q_1 + l_2 q_2 + s_1 + s_2 p + s_3 x_1 + s_4 x_1 p + s_5 x_2 + s_6 x_2 p + s_7 x_1 x_2 + s_8 x_1 x_2 p + p = 0$.*

Rounding the result of an SDP solver yields $l_1 = -\frac{1}{2}\left(x_1 x_2 - x_3 x_4\right)$, $l_2 = -\frac{1}{2}\left(x_2 x_3 + x_1 x_4\right)$, $s_2 = \frac{1}{2}\left(x_3^2 + x_4^2\right)$, $s_7 = \frac{1}{2}\left(x_1^2 + x_2^2 + x_3^2 + x_4^2\right)$ and $s_1 = s_3 = s_4 = s_5 = s_6 = s_8 = 0$. This problem has no margin, since when replacing $p > 0$ by $p \geq 0$, then $(x_1, x_2, x_3, x_4) = (0, \sqrt{2}, 0, 0)$ becomes a solution.

Under some hypotheses, this relaxation scheme is complete, as stated by a theorem from Stengle [Las09, Th. 2.11]. However, similarly to Section 3.3.2, no practical bound is known on the degrees of the relaxation polynomials.

Complexity

The relaxation scheme involves products of all polynomials appearing in the original problem constraints. The number of such products, being exponential in the number of constraints, limits the scalability of the approach.

Moreover, to actually enjoy the benefits of exact solutions, the floating-point Cholesky decomposition introduced in Section 3.2.5 cannot be used and has to be replaced by an exact rational decomposition[16]. Computing decompositions of large matrices can then become particularly costly as the size of the involved rationals can blow up exponentially during the computation.

Soundness

The exact solutions make for an easy verification. The method is thus implemented in the HOL Light [Har07] and Coq [Bes06] proof assistants.

Incompleteness

Although this verification methods can be applied to SDP problems with an empty interior, the rounding heuristic is not guaranteed to provide a solution. In practice, it tends to fail on large problems or problems whose coefficients are not rationals with small numerators and denominators.

3.3.4 Implementation

The OSDP Library

The SOS to SDP translation, as well as the validation methods, described in Section 3.2 have been implemented in our OCaml library OSDP (Ocaml SemiDefinite Programming library). This library offers a common

[12] For instance, when one of the sets $\{x \in \mathbb{R}^n \mid p_i(x) \geq 0\}$ is bounded.

[13] More precisely to $2\left\lceil \frac{d - \deg(p_i)}{2} \right\rceil$ as $deg(r_i)$ is necessarily even since $r_i \geq 0$.

[14] In practice, to ensure that the rounded matrix Q still satisfy the equality $p = z^T Q\, z$, a dual SDP encoding is used, that differs from the encoding introduced in Section 3.2.2. This dual encoding is also called image representation [Par03, §6.1].

[15] However, there exist rational SDP problems that do not admit any rational solution.

[16] The Cholesky decomposition, involving square roots, cannot be computed in rational arithmetic, its LDLT variant can.

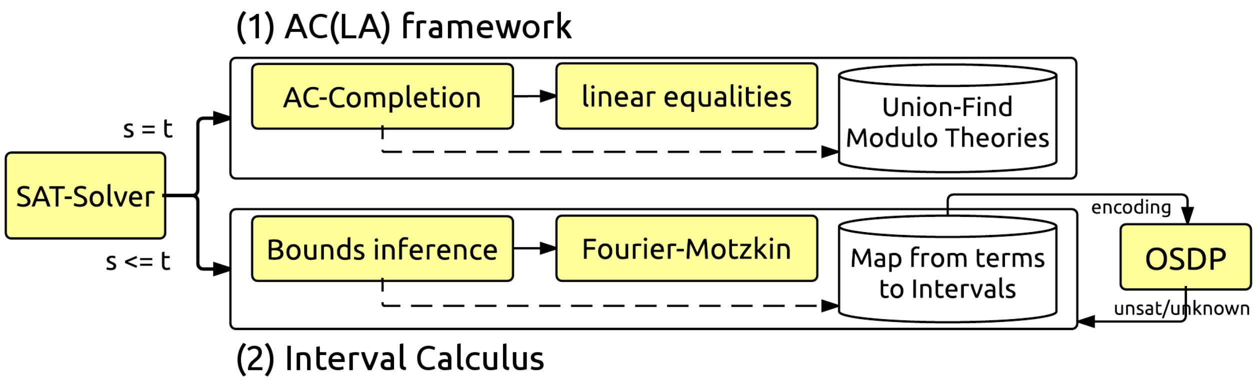


Figure 3.11: Alt-Ergo's arithmetic reasoning framework with OSDP integration.

interface to the SDP solvers Csdp [Bor99], Mosek [ApS15] and SDPA [YFN+10], giving simple access to SOS programming in contexts where soundness matters, such as SMT solvers or program static analyzers. It is composed of 5 kloc of OCaml and 1 kloc of C (interfaces with SDP solvers) and is available under LGPL license at `https://github.com/Embedded-SW-VnV/osdp`.

Integration of OSDP in Alt-Ergo

Alt-Ergo [BCC+14] is a very effective SMT solver for proving formulas generated by program verification frameworks. It is used as a back-end of different tools and in various settings, in particular via the Why3 [FP13] platform. For instance, the Frama-C [CKK+12] suite relies on it to prove formulas generated from C code, and the SPARK [HMWC15] toolset uses it to check formulas produced from Ada programs. It is also used by EasyCrypt [BDG+13] to prove formulas issued from cryptographic protocol verification, from the Cubicle [CGK+12] model-checker, and from Atelier-B [Abr05].

Alt-Ergo's native input language is a polymorphic first-order logic *à la ML* modulo theories, a very suitable language for expressing formulas generated in the context of program verification. Its reasoning engine is built on top of a SAT solver that interacts with a combination of decision procedures to look for a model for the input formula. Universally quantified formulas, that naturally arise in program verification, are handled via E-matching techniques. Currently, Alt-Ergo implements decision procedures for the free theory of equality with uninterpreted symbols, linear arithmetic over integers and rationals, fragments of nonlinear arithmetic, enumerated and record datatypes, and the theory of associative and commutative function symbols (hereafter AC).

Figure 3.11 shows the simplified architecture of arithmetic reasoning framework in Alt-Ergo, and the OSDP extension. The first component in the figure is a completion-like algorithm AC(LA) that reasons modulo associativity and commutativity properties of nonlinear multiplication, as well as its distributivity over addition[17]. AC(LA) is a modular extension of ground AC completion with a decision procedure for reasoning modulo equalities of linear integer and rational arithmetic [CCI12]. It builds and maintains a convergent term-rewriting system modulo arithmetic equalities and the AC properties of the nonlinear multiplication symbol. The rewriting system is used to update a union-find data-structure.

The second component is an Interval Calculus algorithm that computes bounds of (nonlinear) terms: the initial nonlinear problem is first relaxed by abstracting nonlinear parts, and a Fourier-Motzkin extension[18] is used to infer bounds on the abstracted linear problem. In a second step, axioms of nonlinear arithmetic are internally applied by interval propagation. These two steps allow to maintain a map associating the terms of the problems (that are normalized *w.r.t.* the union-find) to unions of intervals.

Finally, the last part is the SAT solver that dispatches equalities and inequalities to the right component and performs case-split analysis over finite domains. Of course, this presentation is very simplified and the exact architecture of Alt-Ergo is much more complicated.

The integration of OSDP in Alt-Ergo is achieved via the extension of the Interval Calculus component of the solver, as shown in Figure 3.11: terms that are polynomials, and their corresponding interval bounds, form the problem (3.4) which is given to OSDP. OSDP attempts to verify its result with the method of Section 3.2. When it succeeds, the original conjunction of constraints is proved `unsat`. Otherwise, (dis)equalities are added and OSDP attempts a new proof by the method of Section 3.3.3. In case of success, `unsat` is proved, otherwise satisfiability or unsatisfiability cannot be deduced. Outlines of the first algorithm are given in Figure 3.12 whereas the second one follows the original implementation [Har07].

[17]Addition and multiplication by a constant is directly handled by the LA module.
[18]We can also use a simplex-based algorithm [BCC+12] for bounds inference.

```
  $p'_1 := (p_1 - a_1)(b_1 - p_1), \ldots, p'_k := (p_k - a_k)(b_k - p_k)$
  // or $p'_i := p_i - a_i$ when $b_i = +\infty$ or $p'_i := b_i - p_i$ when $a_i = -\infty$
  $d := \max_i \{ \deg(p'_i) \}$
  encode $-\sum_{i=1}^{k} r_i p'_i$ is SOS, $r_1$ is SOS, $\ldots r_k$ is SOS
  as an SDP problem $-\sum r_i p'_i = z_0^T Q_0\, z_0,\ r_1 = z_1^T Q_1\, z_1, \ldots, r_k = z_k^T Q_k\, z_k$
  with $\deg(r_i) := 2 \left\lceil \frac{d - \deg(p'_i)}{2} \right\rceil$
  call an SDP solver and retrieve $r_1$, $r_k$ and $Q_0, Q_1, \ldots, Q_k$
  overapproximate $\delta_i := \max_\alpha \left\{ |c_\alpha| \;\middle|\; r_i - z_i^T Q_i\, z_i = \sum_\alpha c_\alpha x^\alpha \right\}$
  if $1 \in z_0 \wedge Q_0 - \#|z_0|\delta_0 I \succ 0 \wedge Q_1 - \#|z_1|\delta_1 I \succ 0 \wedge \ldots \wedge Q_k - \#|z_k|\delta_k I \succ 0$ then return Unsat
  else return Unknown
  end if
```

Figure 3.12: Semi-decision procedure to prove $\bigwedge_{i=1}^{k} p_i \in [a_i, b_i]$ unsat. $\#|z|$ is the size of the vector z and $\succ 0$ is tested with a floating-point Cholesky decomposition [Rum06].

Our modified version of Alt-Ergo is available under CeCILL-C license at `https://cavale.enseeiht.fr/osdp/aesdp/`.

Incrementality

In the SMT context, our theory solver is often successively called with the same problem with a few additional constraints each time. It would then be interesting to avoid doing the whole computation again when a constraint is added, as is usually done with the simplex algorithm for linear arithmetic.

Some SDP solvers do offer to provide an initial point. Our experiments however indicated that this significantly speeds up the computation only when the provided point is extremely close to the solution. A bad initial point could even slow down the computation or, worse, make it fail. This is due to the very different nature of the interior point algorithms, compared to the simplex, and their convergence properties [BV04, Part III]. Thus, speed ups could only be obtained when the previous set of constraints was already unsatisfiable, i.e., a useless case.

Small Conflict Sets

When a set of constraints is unsatisfiable, some of them may not play any role in this unsatisfiability. Returning a small subset of unsatisfiable constraints can help the underlying SAT solver. Such useless constraints can easily be identified in (3.4) when the relaxation polynomial r_i is 0. A common heuristic to maximize their number is to ask the SDP solver to minimize (the sum of) the traces of the matrices Q_i.

When using the exact method of Section 3.3.3, the appropriate r_i are exactly 0. Things are not so clear when using the approximate method of Section 3.2.5 since the r_i are only *close to* 0. A simple solution is to rank the r_i by decreasing trace of Q_i before performing a binary search for the smallest prefix of this sequence proved unsatisfiable. Thus, for n constraints, $\log(n)$ SDPs are solved.

3.3.5 Experimental Results

We compared our modified version of Alt-Ergo (v. 1.30) to the SMT solvers ran in both the QF_NIA (Quantifier Free Nonlinear Integer Arithmetic) and QF_NRA (Quantifier Free Nonlinear Real Arithmetic) sections of the last SMT-COMP (at the time of writing [RIC18]). We ran the solvers on two sets of benchmarks. The first set comes from the QF_NIA and QF_NRA benchmarks for the same SMT-COMP. The second set contains four subsets. The `C` problems are generated by Frama-C/Why3 [CKK+12, FP13] from control-command C programs such as the one from Section 3.3.1, with up to a dozen variables [CSC12, RJGF12]. To distinguish difficulties coming from the handling of the memory model of C, for which Alt-Ergo was particularly designed, and from the actual nonlinear arithmetic problem, the `quadratic` benchmarks contain simplified versions of the `C` problems with a purely arithmetic goal. To demonstrate that the interest of our approach is not limited to this initial target application, the `flyspeck` benchmarks come from the benchmark sets of dReal[19] [GKC13] and `global-opt` are global optimization benchmarks [MN13]. All these benchmarks

[19]Removing problems containing functions sin and cos, not handled by our tool.

	AE		AESDP		AESDPap		AESDPex		CVC4		Smtrat		Yices2		Z3	
	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time
AProVE (746)	103	7387	319	23968	359	7664	318	22701	586	10821	185	3879	**709**	**1982**	252	5156
calypto (97)	92	357	88	679	88	489	89	816	87	7	89	754	**97**	**409**	95	613
LassoRanker (102)	57	9	62	959	64	274	63	878	72	27	20	12	**84**	**595**	84	2538
LCTES (2)	0	0	0	0	0	0	0	0	**1**	**0**	0	0	0	0	0	0
leipzig (5)	0	0	0	0	0	0	0	0	0	0	0	0	**1**	**0**	0	0
mcm (161)	0	0	0	0	0	0	0	0	**4**	**2489**	0	0	0	0	4	2527
UltimateAutom (7)	1	0.35	7	0.73	7	0.62	7	0.69	6	0.03	1	7.22	**7**	**0.04**	7	0.31
UltimateLasso (26)	26	118	26	212	26	126	26	215	4	66	26	177	**26**	**6**	26	21
total (1146)	279	7872	502	25818	544	8553	503	24611	780	13411	321	4829	**924**	**2993**	468	10855

Table 3.1: Experimental results on benchmarks from QF_NIA.

	AE		AESDP		AESDPap		AESDPex		CVC4		Smtrat		Yices2		Z3	
	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time
Sturm-MBO (300)	155	12950	155	13075	155	13053	155	12973	285	1403	**285**	**620**	2	0	47	21
Sturm-MGC (7)	0	0	0	0	0	0	0	0	1	7	1	0	0	0	**7**	**0**
Heizmann (68)	0	0	0	0	0	0	0	0	1	16	1	0	**11**	**2083**	3	41
hong (20)	1	0	20	28	20	24	20	27	20	1	**20**	**0**	8	240	9	6
hycomp (2494)	1285	15351	1266	15857	1271	16080	1265	14909	2184	208	1588	13784	2182	1241	2201	4498
keymaera (320)	261	36	291	356	278	97	291	360	249	4	307	13	270	359	**318**	**2**
LassoRanker (627)	0	0	0	0	0	0	0	0	**441**	**32786**	0	0	236	30835	119	1733
meti-tarski (2615)	1882	10	2273	91	2267	65	2241	73	1643	804	2520	3345	2578	2027	**2611**	**337**
UltimateAutom (13)	0	0	0	0	0	0	0	0	5	0.52	0	0	12	57.19	**13**	**19.23**
zankl (85)	14	1.00	24	15.46	24	16.09	24	15.67	24	9.40	19	13.47	**32**	**7.22**	27	0.43
total (6549)	3571	28348	4029	29423	4015	29334	3996	28357	4853	35239	4740	17775	5331	36849	**5355**	**6658**

Table 3.2: Experimental results on benchmarks from QF_NRA.

are available at `https://cavale.enseeiht.fr/osdp/aesdp/`. Since our solver only targets unsat proofs, benchmarks known sat were removed from both sets.

All experiments were conducted on an Intel Xeon 2.30 GHz processor, with individual runs limited to 2GB of memory and 900 seconds. The results are presented in Tables 3.1, 3.2 and 3.3. For each subset of problems, the first column indicates the number of problems that each solver managed to prove unsat and the second presents the cumulative time (in seconds) for these problems. AE is the original Alt-Ergo, AESDP our new version, AESDPap the same but using only the approximate method of Section 3.2.5 and AESDPex using only the exact method of Section 3.3.3. All solvers were run with default options, except CVC4 which was run with all its `-nl-alg*` options.

As seen in Table 3.1 and 3.2, despite an improvement over Alt-Ergo alone, our development is not competitive with state-of-the-art solvers on the QF_NIA and QF_NRA benchmarks. The most commonly observed source of failure for AESDPap here comes from SDPs with empty relative interior. For instance, the `sqrt-circles-2-chunk-0008.smt2` file from the meti-tarski directory essentially asks for a proof of $\forall d\ y\ x \in \mathbb{R}^3, y(2 + 2d - y) \leq 1 + d(2 + d) + x^2$ whereas the equality is reached for $d = 1$, $y = 2$ and $x = 0$. Although AESDPex can handle such problems, it is impaired by its much higher complexity.

However good results are obtained on the more numerical[20] second set of benchmarks. In particular, control-command programs with up to a dozen variables are verified while other solvers remain limited to two variables. Playing a key point in this result, the inequalities in these benchmarks are satisfied with some margin. For control command programs, this comes from the fact that they are designed to be robust to many small errors.

Although solvers such as dReal, based on branch and bound with interval arithmetic, could be expected to perform well on these numerical benchmarks, dReal solves less benchmarks than most other solvers[21]. Geometrically speaking, the `C` benchmarks require to prove that an ellipsoid is included in a slightly larger one, i.e., the borders of both ellipsoids are close to one another. This requires to subdivide the space between the two borders into many small boxes so that none of them intersects both the interior of the first ellipsoid and the exterior of the second one. Whereas this can remain tractable for small dimensional ellipsoids, the number of required boxes grows exponentially with the dimension, which explains the poor results of dReal.

3.3.6 Related Work and Conclusion

Related work. Monniaux and Corbineau [MC11] replaced the rounding heuristic of Harrison [Har07] by a facial-reduction heuristic. This has unfortunately no impact on the complexity of the relaxation scheme. Platzer et al. [PQR09] compared their early versions with the symbolic methods based on quantifier elimination and Gröbner basis. An intermediate solution is offered by Magron et al. [MAGW15] but only handling a restricted class of parametric problems.

Branch-and-bound and interval arithmetic constitute another numerical approach to nonlinear arithmetic, as implemented in the SMT solver dReal by Gao et al. [GAC12, GKC13]. These methods easily handle

[20]Involving polynomials with a few dozen monomials or more and whose coefficients are not integers or rationals with small numerators and denominators.

[21]Except for its `flyspeck` benchmarks (16 out of 20 proved).

	AE		AESDP		AESDPap		AESDPex		CVC4		Smtrat		Yices2		Z3		dReal	
	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time	unsat	time
C (67)	11	0.05	**63**	**39.78**	63	40.01	13	1.18	0	0	0	0	0	0	0	0	0	0
quadratic (67)	13	0.06	**67**	**14.68**	67	15.44	15	0.08	14	2.46	18	1.26	25	357.20	25	257.39	13	23.36
flyspeck (20)	1	0.00	**19**	**26.35**	19	26.62	3	0.01	6	695.59	9	36.54	10	0.05	9	0.05	16	11.77
global-opt (14)	2	0.01	**14**	**8.72**	14	8.83	5	0.20	5	0.12	12	41.18	12	0.16	13	683.45	9	0.05
total (168)	27	0.12	**163**	**89.53**	163	90.90	36	1.47	25	698.17	39	78.98	47	357.41	47	940.89	38	35.18

Table 3.3: Experimental results on benchmarks from [CSC12, GKC13, MN13, RJGF12].

nonlinear functions such as the trigonometric functions sin or cos, not yet considered in our prototype[22]. In the case of polynomial inequalities Muñoz and Narkawicz [MN13] offer Bernstein polynomials as an improvement to simple interval arithmetic.

Finally, VSDP [HJL, JCK07] is a wrapper to SDP solvers offering a similar method to the one of Section 3.2.4. Moreover, an implementation is also offered by Löfberg [Löf09] in the popular Matlab interface Yalmip but remains unsound, since all computations are performed with floating-point arithmetic, ignoring rounding errors.

Using convex optimization into an SMT solver was already proposed by Nuzzo et al. [NPSS10, SNS$^+$17]. However, they intentionally made their solver unsound in order to lean toward completeness. While this can make sense in a bounded model checking context, soundness is required for many applications, such as program verification. Moreover, this proposal was limited to convex formulas. Although this enables to provide models for satisfiable formulas, while only unsat formulas are considered in this paper, and whereas this seems a perfect choice for bounded model checking applications, nonconvex formulas are pervasive in applications such as program verification[23].

The use of numerical off-the-shelf solvers in SMT tools has also been studied in the framework of linear arithmetic [FNOR08, Mon09]. Some comparison with state-of-the-art exact simplex procedures show mitigated results [dOM12] but better results can be obtained by combining both approaches [KBT14].

Conclusion. We presented a semi-decision procedure for nonlinear polynomial constraints over the reals, based on numerical optimization solvers. Since these solvers only compute approximate solutions, a-posteriori soundness checks were investigated. Our first prototype implemented in the Alt-Ergo SMT solver shows that, although the new numerical method does not strictly outperform state-of-the-art symbolic methods, it enables to solve practical problems that are out of reach for other methods. In particular, this is demonstrated on the verification of functional properties of control-command programs. Such properties are of significant importance for critical cyber-physical systems.

3.4 Inside the Coq Proof Assistant

This section mostly sums ups our CPP 2017 paper [MR17], that makes the method of the previous Section 3.2 available as an automatic tactic inside the Coq proof assistant. This work was done in close collaboration with Érik Martin-Dorel.

The main contribution of this section is to demonstrate that the rigorous proofs introduced in section 3.2.5 can be mechanized in a proof assistant with nice computation capabilities such as Coq [Coq24]. We also identify which mechanisms could be added to Coq in order to make this kind of rigorous proofs based on floating-point computations much more efficient.

After the next section describing related work, Section 3.4.2 presents the data refinements that were needed in order to obtain an executable version of the algorithms described in Section 3.2.5. Then, Section 3.4.3 details our tactic that is able to automatically discharge polynomial inequality goals thanks to the previous programs. Finally, Section 3.4.4 comments the results on some benchmarks and compares our tactic to other tools while Section 3.4.5 concludes.

3.4.1 Related Work

There have been several works for developing (semi)decision procedures for nonlinear inequalities in formal proof assistants. Some of them follow the so-called *autarkic* approach and perform all the required computations within the prover itself. Others follow the so-called *skeptical* approach and delegate most of the computations to some external, possibly unsound yet efficient, oracle. In this latter case the prover only needs to verify the witnesses generated by the oracle, by using a dedicated algorithm that has been formally verified.

First, let us focus on the HOL Light proof assistant. The REAL_SOS decision procedure [Har07] relies on the CSDP library [Bor99] for semidefinite programming. It generates a semidefinite programming problem

[22]Polynomial approximations such as Taylor expansions should be investigated.

[23]Typically, to prove a convex loop invariant I for a loop body f, one need to prove $I \Rightarrow I(f)$, that is $\neg I \vee I(f)$ which is likely nonconvex ($\neg I$ being concave).

from the user's goal, calls CSDP and tries to infer an "exact" solution. This is done by rounding the approximation solution returned by the SDP solver to "rational numbers with moderate coefficients", as discussed in Section 3.3.3.

Another decision procedure for the HOL Light proof assistant has been developed as part of the Flyspeck project[24]: verify_ineq [SH13]. It does not involve SDP computations but relies on the computation of Taylor polynomials, using interval arithmetic and bisection. To be more specific, it uses an external search procedure to precompute a "solution certificate" that is useful to speed-up the *a posteriori* verification. These certificates notably describe how the input domain should be split. Then, the verification procedure computes order-1 Taylor-Lagrange enclosures (i.e., with quadratic remainders) using interval arithmetic and floating-point numbers. Contrarily to REAL_SOS, verify_ineq supports transcendental functions such as cos and arctan.

Focusing on the Coq proof assistant, Laurent Théry has isolated the HOL-independent code of REAL_SOS that Frédéric Besson has then integrated in several decision procedures within the Micromega Coq library [Bes06]. The approach of Micromega is the same as that of REAL_SOS: it depends on CSDP and relies on the Positivstellensatz.

Another decision procedure that is amenable to Coq formal proof has been developed with a special focus on certifying SDP problems with empty interior [MC11]. This package has been written in Sage and relies on DSDP [BY08] and fpLLL[25]: it can be viewed as an enhancement of [Har07]'s approach, with the simple rounding heuristic replaced by an actual facial-reduction, that takes advantage of the LLL algorithm [LLL82].

The CoqInterval library provides a decision procedure for nonlinear inequalities with transcendental functions. It has been developed with a special focus on verifying approximation errors for univariate expressions, but is also able to solve quasi-multivariate problems [MDM16]. It follows the *autarkic* approach and involves floating-point and interval arithmetic, bisection, and computation of univariate order-n Taylor polynomials.

Within the Flyspeck project, a decision procedure named NLCertify has been devised for the Coq proof assistant [Mag14]. It relies on the SDPA solver, on an OCaml backend that generates positivity certificates, and on a Coq verification procedure that mainly relies on the `ring` tactic. NLCertify supports nonlinear multivariate inequalities with transcendental functions over a hyperbox, but only the polynomial goals are formally certified currently.

Two decision procedures are available for the PVS proof assistant: `interval` [NM13] and `bernstein` [MN13]. Both rely on a branch-and-bound algorithm and follow the *autarkic* approach. The `interval` PVS strategy supports transcendental functions and uses interval arithmetic with rational bounds. The `bernstein` PVS strategy relies on a representation of multivariate polynomials in Bernstein form to easily infer bounds on them.

It is worth noting that all the above work performing proofs in exact rational arithmetic may be able to prove "sharp" inequalities. In contrast, the tactic presented in this paper will usually only be able to prove inequalities satisfied within some margin (i.e., inequalities $p < q$ such that $p + \varepsilon < q$ holds for some $\varepsilon > 0$). This is inherent in the floating-point arithmetic roundings.

3.4.2 Verification of Effective Computation using Data Refinement

The Coq proofs mentioned in section 3.2.5 deal with abstract versions of the algorithms, which are not suitable for computation. For instance, Mathcomp's matrices have been specifically designed to ease reasoning on them, but one cannot compute with them as most functions about matrices do not permit evaluation. So we rely on the CoqEAL library [CDM13] which has been devised to facilitate the design and the proof of effective computation, notably for linear algebra. We thus apply the following methodology that is typical when using CoqEAL:

- We implement the algorithms in a general way (using polymorphic functions and Type Classes).
- We specialize the algorithms with proof-oriented datatypes and prove these algorithms correct. This step often requires doing some "program refinement", i.e., proving that the considered algorithm is extensionally equal to a simpler algorithm.
- We specialize the algorithms with effective datatypes and prove them correct with respect to the proof-oriented datatypes. This amounts to doing some "data refinement".

We elaborate on the last step in the rest of this section.

To obtain an effective version of our verification algorithm, we need to compute with rational numbers, floating-point numbers, matrices, vectors of monomials and multivariate polynomials. To this aim, we rely on the following Coq libraries : (i) BigQ from the bignums [GT06] library[26] for efficient arbitrary precision

[24]`https://github.com/flyspeck/flyspeck`
[25]`https://github.com/fplll/fplll`
[26]`https://github.com/coq-community/bignums/`

arithmetic (based on machine integers); (*ii*) The floating-point kernel of CoqInterval [MDM16] for emulating Binary64 floating-point numbers; (*iii*) The CoqEAL library for effective matrices based on lists of lists; (*iv*) Lists of binary integers (`seq N`) for vectors of monomials; (*v*) The FMapAVL standard library for efficient maps (based on AVL trees).

Relying on these datatypes, a major contribution of our work was to formalize effective multivariate polynomials using FMaps, and prove them correct with respect to MathComp Multinomials[27], by taking advantage of CoqEAL's framework. This formalization is now integrated in CoqEAL (file `multipoly.v`).

We also needed to refine proof-oriented rational numbers (of type `rat`, which is endowed with a `realFieldType` Mathcomp algebraic structure) to effective rationals, and provide functions to go back and forth between rationals and floating-point numbers.

3.4.3 An Automated Tactic for Verifying Positivity Witnesses

Calling SDP Solvers Through an OCaml Module

The main ingredients of the positivity proofs using SOS polynomials, as described in Section 3.2.5, are the vector of monomial z and the matrix Q computed by the numerical SDP solvers.

We developed an OCaml library called OSDP, already described in Section 3.3.4. A Coq module written in OCaml then uses this library. This module, called `soswitness`, only requires 0.3 kloc of OCaml to read a polynomial from Coq, call OSDP and translate the resulting z and Q back to Coq terms. It is worth noting that all this OCaml code and the underlying SDP solvers are only used as untrusted oracles, just making the main proof ingredients available to the formally verified Coq tactic discussed in the remainder of this section. The main advantage of this skeptical approach is to enable the use of any off-the-shelf solver and easy implementation of arbitrary optimizations in the SOS to SDP translation, without any risk of jeopardizing the eventual proof soundness.

Verification of the Witness

The verification of witnesses is performed by a computational boolean function `soscheck_eff_wrapup` whose type is as follows:

```
seq R → p_abstr_poly → seq (seq N) * seq (seq (s_float bigZ bigZ)) → bool
```

The first two arguments (the list of real variables involved in the user's goal and the abstract syntax tree corresponding to the polynomial under study) will be obtained after the reification of the user's goal (c.f., the upcoming section). The third argument is the witness `(z, Q)` obtained after calling the external SDP solver, i.e., a list (`seq`) of monomials (list of degree for each variable `seq N`) and a matrix of floating-point values (`seq (seq (s_float bigZ bigZ))`). It can be noted that we relied on CoqEAL's approach for data refinement (see Section 3.4.2) on every building block involved in the definition of `soscheck_eff_wrapup`.

We then prove the main correctness claim associated to this function by relying on the previously presented material:

```
Theorem soscheck_eff_wrapup_correct :
  ∀ (vm : seq R) (pap : p_abstr_poly) (zQ : seq(seq N) * seq(seq(s_float bigZ bigZ))),
    soscheck_eff_wrapup vm pap zQ = true → (0 ≤ interp_p_abstr_poly vm pap)%R.
```

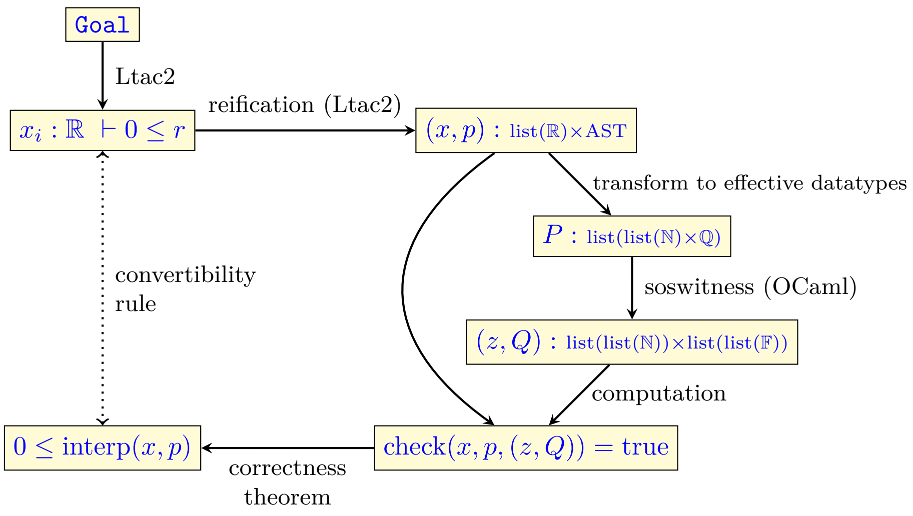
Overview of the Tactic

Finally, we developed some dedicated Ltac2 code to reify the user's goal and obtain an element of the `p_abstr_poly` abstract syntax tree. This syntax involves rational constants (defined as a separate inductive `p_real_cst` built from nonnegative integers, opposites and divisions thereof), real variables, and the usual polynomial operations including exponentiation (with positive, constant exponents). As usual with a Coq reflexive tactic, the Ltac2 code then applies the correctness claim `soscheck_eff_wrapup_correct` to the user's goal and discharges the obtained subgoals by computation. One can find an overview of the tactic in Figure 3.13.

As a result, our `validsdp` tactic recognizes goals that are inequalities between polynomial expressions, involving atomic real variables as well as rational constants.

The complete development is available at `https://github.com/validsdp/validsdp/`. It amounts to 13.6 kloc, that can be decomposed as 5.1 kloc for the proof of the Cholesky decomposition algorithm, 3.0 kloc for the refinement of the algorithm to an executable program (relying on an arithmetic parameterized with Type Classes), 1.3 kloc for the refinement of CoqInterval floating-point arithmetic, 2.2 kloc for the refinement of multivariate polynomials and finally 2.0 kloc for the main tactic and the related correctness proofs.

[27]`https://github.com/math-comp/multinomials/`



Goal
Ltac2
$x_i : \mathbb{R} \vdash 0 \le r$
reification (Ltac2)
(x, p) : list($\mathbb{R}$)×AST
transform to effective datatypes
P : list(list($\mathbb{N}$)×$\mathbb{Q}$)
soswitness (OCaml)
(z, Q) : list(list($\mathbb{N}$))×list(list($\mathbb{F}$))
computation
convertibility rule
$0 \le \mathrm{interp}(x, p)$
$\mathrm{check}(x, p, (z, Q)) = \mathrm{true}$
correctness theorem

Figure 3.13: Overview of the `validsdp` reflexive tactic.

3.4.4 Benchmarks

We evaluated the performances of our tactic with respect to the related works mentioned in Section 3.4.1 on a set of benchmarks. These benchmarks are composed of three subsets:

- A set of global optimization problems consisting in proving bounds for polynomials on given hyperboxes. These problems are taken from [MN13] and correspond to the `global-opt` category in previous Section 3.3.5.
- A set of problems on 6 variable polynomials coming from some inequalities of the Flyspeck project. Most of them require to prove that a given polynomial is positive on a given hyperbox. Some require to prove that at least one of two or three polynomials is positive on any point of a given hyperbox (two polynomials in `fs868`, `fs884` and `fs890`, three polynomials in `fs260` and `fs491`). They mostly correspond to the `flyspeck` category in previous Section 3.3.5.
- A set of problems on loop invariants of programs [AGM15, RVS16]. They require to prove that a given polynomial p is nonnegative when some initial conditions are met and that whenever p is nonnegative, then $p \circ f$ also is, for given polynomials f (representing the loop body assignments).

Note that some benchmarks are known to be false[28] (viz. `ex4_d4`, `ex4_d6` and `ex7_d4`), so the absence of proof is the expected result. Others are known to be correct[29] (`fs884`, `ex4_d8` and `ex7_d8`). Finally, the status of the remaining unproved benchmarks is unknown (`fs859`, `ex4_d10`, `ex7_d10`, `ex8_d6`, `ex8_d8` and `ex8_d10`).

Table 3.4 presents the results on a Core i5-4460S CPU clocked at 2.9 GHz. Unfortunately, some of the tools (PVS/Bernstein, NLCertify and HOL Light/Taylor) were not applicable on some benchmarks as they require a hyperbox. It is also worth noting that for methods based on SDP solvers (i.e., all methods but PVS/Bernstein and HOL Light/Taylor), its choice can have a slight influence on the results. As a matter of fact, NLCertify uses SDPA. Monniaux and Corbineau 2011 can use either CSDP, DSDP or SDPA and we chose CSDP as it gave the best results. Finally, OSDP (hence ValidSDP) can use either CSDP, Mosek or SDPA. Although Mosek tends to give the best results on large problems and is about ten times faster, we chose SDPA for the sake of fairness of the comparison.

These results indicate that while our tactic, based on floating-point arithmetic, is competitive with other tools on the simplest benchmarks it, more noticeably, enables to perform proofs that seem out of reach for other methods, mostly based on rational arithmetic. It may be noted that those benchmarks could be seen as quite unfair to other tools, implementing rounding to exact solutions, as described in Section 3.3.3. Indeed, other tools could thus solve some nonstrictly feasible problems on which our tool will always fail, but all benchmarks are strictly feasible (at least all of those that are known to be feasible).

The OSDP column in the table is included to give some insight on what is lost, in terms of performance, by running the verification procedure inside the Coq virtual machine (ValidSDP column). Indeed, most of the time in the OSDP column is spent running SDP solvers. The table then indicates that our Coq tactic commonly incurs a factor ten to fifty overhead. This is somewhat disappointing as the verification of results

[28]Counter examples can easily be found.
[29]Another SDP solver and/or some symbolic preprocessing enables the proof to go through with OSDP.

Problem	n	d	OSDP (not verified)	ValidSDP	PVS/Bernstein	Monniaux and Corbineau 11 (not verified)[**]	NLCertify (not verified)	NLCertify	HOL Light/ Taylor[*]
adaptativeLV	4	4	0.70	8.73	14.93	2.67	1.12	2.61	12.31
butcher	6	4	0.84	21.14	48.44	—	1.05	8.36	15.62
caprasse	4	4	0.47	8.91	25.89	1.82	0.88	2.63	17.68
heart	8	4	0.99	37.99	131.13	268.75	—	—	26.15
magnetism	7	2	0.24	4.68	245.52	2.04	1.64	14.50	16.07
reaction	3	2	1.02	4.05	11.48	1.56	0.24	1.96	12.41
schwefel	3	4	0.88	5.67	14.72	2.45	2.76	56.13	17.46
fs260	6	4	0.26	12.32	—	—	—	—	—
fs461	6	4	0.26	11.24	621.06	11.18	0.87	7.46	22.70
fs491	6	4	1.02	14.93	—	21.81	—	—	—
fs745	6	4	0.29	12.20	623.17	11.74	0.94	6.90	22.48
fs752	6	2	0.43	3.88	54.52	1.81	0.90	7.88	13.34
fs859	6	8	—	—	—	—	—	—	—
fs860	6	4	0.97	11.09	73.65	10.53	1.11	7.34	14.28
fs861	6	4	0.25	11.13	69.74	10.48	1.20	7.87	14.28
fs862	6	4	0.79	10.81	73.54	79.25	1.25	7.58	14.14
fs863	6	2	—	—	—	1.50	—	—	13.85
fs864	6	2	0.97	4.50	—	2.05	—	—	13.28
fs865	6	2	0.93	4.56	—	2.11	—	—	13.76
fs867	6	2	0.47	3.90	—	2.09	1.74	8.04	—
fs868	6	4	0.72	12.65	—	—	—	—	—
fs884	6	4	—	—	—	—	—	—	—
fs890	6	4	—	—	—	7.78	—	—	—
fs8	6	2	0.65	4.58	52.63	1.53	1.48	6.62	13.40
ex4_d4	2	12	—	—	—	—	—	—	—
ex4_d6	2	18	—	—	—	—	—	—	—
ex4_d8	2	24	—	—	—	—	—	—	—
ex4_d10	2	30	—	—	—	—	—	—	—
ex5_d4	3	8	0.73	22.67	—	—	—	—	—
ex5_d6	3	12	3.41	85.34	—	—	—	—	—
ex5_d8	3	16	20.54	324.29	—	—	—	—	—
ex5_d10	3	20	150.86	—	—	—	—	—	—
ex6_d4	4	8	2.44	57.97	—	—	—	—	—
ex6_d6	4	12	56.19	502.17	—	—	—	—	—
ex7_d4	2	12	—	—	—	—	—	—	—
ex7_d6	2	18	0.84	43.87	—	—	—	—	—
ex7_d8	2	24	—	—	—	—	—	—	—
ex7_d10	2	30	—	—	—	—	—	—	—
ex8_d4	2	8	0.13	10.53	—	15.72	—	—	—
ex8_d6	2	12	—	—	—	—	—	—	—
ex8_d8	2	16	—	—	—	—	—	—	—
ex8_d10	2	20	—	—	—	—	—	—	—

Table 3.4: Running time (elapsed real time, in s) for various tools on a set of benchmarks. "—" indicates either that a tool is not applicable or that it failed to produce a proof within the time limit (900 s). n is the number of variables of the polynomials and d is their degree. "OSDP" is our OCaml implementation of our proof procedure and "ValidSDP" our Coq tactic, "PVS/Bernstein" corresponds to [MN13], "Monniaux and Corbineau 11" corresponds to [MC11], "NLCertify" corresponds to [Mag14], and "HOL Light/Taylor" corresponds to [SH13].

[*]Remark: it should be noted that each running time in the last column includes the time (around 11s) for loading the image of the HOL Light libraries, checkpointed beforehand using DMTCP.

[**]Remark: although this column doesn't include verification time by a proof assistant, it can ben done via the `ring` tactic of Coq, just like the further NLCertify column.

from SDP solvers should ideally imply a negligible overhead. However, most of this overhead comes from the emulation of floating-point arithmetic that is typically three orders of magnitude slower than native operations performed by the hardware floating-point unit. This means that a native implementation of floating-point arithmetic in the Coq kernel, as done with 31 bit integers [AGST10], could yield a speedup of our tactic of one or two orders of magnitude. Such an implementation will be presented in next Chapter 4. It is also worth noting that the memory footprint of the Coq implementation appears to be about ten times the memory footprint of the OCaml implementation.

Finally, one can notice that all computations in our tactic are performed with the `vm_compute` instead of the more recent `native_compute` mechanism [BDG11]. We did experiment with this mechanism[30]. Unfortunately, at the time of writing, the large terms, corresponding to the Q matrices, produced by our OCaml module led to huge OCaml source codes[31] whose compilation time made `native_compute` a few times slower than `vm_compute`. This has since been fixed and next chapter will show that `native_compute` can indeed be a few times faster on large problem instances.

3.4.5 Conclusion and Future Work

We developed a reflexive Coq tactic for proving multivariate polynomials positivity. This tactic, relying on intensive floating-point computations, demonstrates that performing rigorous proofs inside a proof assistant using floating-point implementations of nontrivial algorithms is tractable. The fact that our tactic is able to discharge proof obligations that seem untractable with other state of the art methods even indicates that such proof methods can be profitable. This, more generally, opens to proof assistants a wide range of rigorous proofs based on floating-point computations [Rum10].

Our experiments also indicate that this kind of proof methodology would greatly benefit from having native floating-point operations available in the proof assistant, rather than having to emulate them, which will be the focus of next Chapter 4.

[30] `https://coq.inria.fr/bugs/show_bug.cgi?id=4714`

[31] For instance, 10MB, 100 kloc OCaml source code for medium size benchmarks.

Chapter 4

Hardware Floats in Coq

This chapter mostly sums ups our JAR 2023 paper [MMR23], that makes hardware floating-point operators available in the Coq proof assistant. This work was done in close collaboration with Érik Martin-Dorel. The initial implementation was done by Guillaume Bertholon in summer 2018 during his excellent L3 internship from ENS Paris, coadvised by Érik and myself. We then completed the implementation and got the feature integrated in Coq 8.11, released in November 2019. We also collaborated with Guillaume Melquiond, particularly for the integration of the feature into the Flocq and CoqInterval libraries.

4.1 Introduction

Efficient and reproducible floating-point computations are widely available nowadays, from embedded processors to supercomputers, thanks to the internationally recognized IEEE-754 standard [IEE08]. The primary use of floating-point arithmetic is to perform approximate computations over real numbers. Due to the limited precision and range of floating-point numbers, various numerical issues arise: rounding errors, overflows, underflows, loss of algebraic properties, and so on.

These issues have not stopped people from using floating-point arithmetic for serious applications. In fact, given sufficient care in the implementation, intensive floating-point computations can even be used in proofs of mathematical theorems, e.g., the existence of a strange attractor for Lorenz' equations [Tuc02]. To do so, the algorithms not only compute a floating-point approximation of the ideal real result but also a bound on the approximation error. This bound might be computed dynamically, as is the case with interval arithmetic [Moo63]. This approach is easy to use and correct by construction but it is more expensive than traditional floating-point arithmetic. Another approach consists in mathematically bounding the approximation error by performing a comprehensive floating-point analysis. Correctness becomes much more tedious to guarantee (though tools for static analysis might help in some specific cases), but this offers a wide range of efficient yet rigorous algorithms [Rum10].

Both approaches have been used to formally prove theorems with the Coq proof assistant. On the one hand, the use of interval arithmetic can be illustrated by the CoqInterval library [MDM16], which automatically and formally proves properties on real expressions by computing their floating-point enclosures. To do so, it specifies, implements, and verifies an ad-hoc floating-point arithmetic in Coq: arbitrary radix, arbitrary precision, unbounded exponent range, neither infinities nor signed zeros. On the other hand, the use of precomputed error bounds can be illustrated by the formal verification of the rigorous variant of Cholesky decomposition [Rou16]. This time, the underlying arithmetic complies with the IEEE-754 standard, so one can use the reference implementation provided by the Flocq library to perform the floating-point computations [BM11]. In either case, computations on floating-point numbers are emulated inside the logic of Coq using integer arithmetic, and thus they do not benefit from the highly efficient floating-point unit of the processor running the proof assistant.

To improve on this unfortunate state of affairs, support for hardware floating-point numbers was added to Coq [BMDR19], in a way reminiscent of the support for machine integers [AGST10, Dén13]. In both cases, the process was as follows. One first declares an abstract type as well as some operations over it: addition, multiplication, and so on. Then, one provides some dedicated reduction rules, which delegate the operations to the arithmetic unit of the processor. This makes it possible to formally prove an equality such as $1.0 + 2.0 = 2.0 + 1.0$ by reducing it to $3.0 = 3.0$. But short of enumerating all the finitely-many pairs of floating-point numbers (which is practically impossible), this is not sufficient to formally prove $\forall x, y,\ x + y = y + x$. So, one also has to relate these abstract operations to some concrete definitions that are much slower but exhibit suitable mathematical properties.

Section 4.2 focuses on the latter part, that is, how these concrete definitions are formalized in Coq. A peculiarity of this specification of floating-point arithmetic is that it is complete but hardly useful in isolation, as it explains how the numbers are computed but not what their properties are. We thus extended this specification using the Flocq library to obtain a meaningful formalization that bridges the gap between the hardware floating-point numbers provided by Coq and the real numbers.

Section 4.3 then focuses on the actual implementation. It presents how the various conversion and reduction engines of Coq were modified to support hardware floating-point computations [BMDR19], as well as various improvements to make this support more useful and usable. In particular, it explains some design choices related to rounding modes (which are critical for interval arithmetic) and to parsing and printing. Finally, it discusses the issue of trust, as both the specification and implementation amount to adding a large numbers of axioms. In particular, we analyze all the soundness bugs that were found (and fixed) during the four years that followed the original implementation.

Even once a consistent system has been recovered, relying on the hardware floating-point unit in a proof assistant would be pointless if it required too much of a proof effort or if the performance gain was too small. So, we have converted two preexisting, representative, Coq developments, so as to evaluate the costs and the benefits. Section 4.4 describes what kind of work this conversion entailed. It also benchmarks how proofs relying on hardware floating-point arithmetic compare to those based on emulated computations, performance-wise.

4.2 Specification: Coq and Flocq

From the point of view of the logic of Coq, the type `float` of floating-point numbers is completely abstract. Similarly, basic operations on these numbers are declared as axioms. Except for some hardcoded reduction rules for these operations (which are delegated to the floating-point unit of the processor), no property is known. Thus, a specification describing this arithmetic is needed. Its role is twofold.

First, it should precisely characterize what the floating-point operations compute. In other words, one should be able to prove properties about what is being computed without performing the computation, *e.g.*, commutativity of addition. The axioms describing this part of the specification are shipped with Coq's standard library. It provides an inductive data type `spec_float` representing floating-point numbers, as well as conversion functions from and to the abstract type `float`. It also provides some naive implementation of the arithmetic operations over `spec_float`, and then states as axioms that the conversion function is a morphism from `float` to `spec_float`. Section 4.2.1 gives more details about this specification.

This first specification is complete but useless in practice, as it is purely operational. It is no different from a software floating-point emulator such as SoftFloat.[1] One should not have to look at the implementation to make use of floating-point numbers inside proofs. So, we need higher-level properties about floating-point operations. In particular, the IEEE-754 standard states that an arithmetic operation shall be performed as if it first computed the result with infinite precision and then rounded it to the target floating-point format [IEE08]. By "infinite precision", the standard simply means that, except for the exceptional values, floating-point numbers are just real numbers, and operations behave the same, rounding notwithstanding. With such a specification, it becomes possible to perform floating-point computations to prove properties about real numbers. This higher-level specification is provided by the Flocq library [BM11]. If not for the large and intricate proofs that relate both specifications, the Flocq specification could also have been shipped with Coq's standard library. Section 4.2.2 gives more details about it.

4.2.1 On Coq's side

Support for hardware floating-point arithmetic in Coq was inspired by two libraries: the Flocq library [BM11], which offers an IEEE-754-compliant executable formalization[2], and the formalization of primitive 63-bit integers [Dén13], now part of the standard library of Coq.[3] The former provides a precise specification in the form of a reference implementation of floating-point operators, while the latter guided the implementation methodology.

Flocq defines an inductive type `full_float` that represents unbounded floating-point numbers. Our aim was to extract a subset of this Flocq theory sufficient to completely specify floating-point numbers and operators, so that Coq does not depend on Flocq at compilation time. We started by porting this definition, just removing the NaN payloads because they are not fully specified by the IEEE-754 standard, which can lead to implementation discrepancies between hardware vendors. Thus, ignoring these payloads in Coq is paramount to guarantee the portability of computations and proofs performed with the same Coq script on different processors.

[1] `http://www.jhauser.us/arithmetic/SoftFloat.html`

[2] Former `IEEE754.Binary` module `https://gitlab.inria.fr/flocq/flocq/-/blob/flocq-3.2.1/src/IEEE754/Binary.v`

[3] In the `UInt63` module `https://coq.github.io/doc/V8.17.0/stdlib/Coq.Numbers.Cyclic.Int63.Uint63.html`

```
Variant spec_float : Set :=
  | S754_zero (s : bool)  (* +0 and -0 *)
  | S754_infinity (s : bool)  (* +oo and -oo *)
  | S754_nan  (* no payload *)
  | S754_finite : (s : bool) (m : positive) (e : Z).  (* m*2^e and -m*2^e *)
```

The types `positive` and `Z` are offered by Coq standard library to represent unbounded integers, respectively positive and signed.

Values of type `spec_float` are neither normalized nor bounded, since no bound is enforced on mantissas and exponents. But in practice, all the operators will make sure that the exponent `e` is bounded, and that the mantissa `m` contains 53 bits for normalized numbers, and at most 52 for subnormal numbers, thus ensuring that the number belongs to the `binary64` format of the IEEE-754 standard.

Note that this definition actually matches Flocq's original formalization of the IEEE-754 standard [BJLM13]. But the set of NaN values was later extended to make it suitable for the semantics used in the CompCert C compiler [BJLM15]. Indeed, even if nothing can be specified about NaN payloads, they can still be distinguished by a C program and thus need to exist. As part of this work, the original formalization was added back to Flocq, in the module `IEEE754.BinarySingleNaN`.

Next, following the same methodology as that of the primitive 63-bit integers formalization, we have declared an abstract type and some arithmetic operations:

```
Primitive float := #float64_type.
Primitive add := #float64_add. (* and so on *)
```

Here, the `Primitive` vernacular amounts to introducing `Parameters` (*i.e.*, axiomatic symbols) that the kernel maps to hardware operations whenever they are fully applied to concrete floating-point values, as explained in Section 4.3.

After declaring various axiomatic symbols to manipulate `float` values (*e.g.*, `is_nan`, `of_uint63`), the types `float` and `spec_float` are linked through two mappings `Prim2SF` and `SF2Prim`, implemented as regular definitions (using the axiomatic symbols):

```
Definition Prim2SF (f : float) : spec_float := (* body omitted *).
Definition SF2Prim (f : spec_float) : float := (* body omitted *).
```

Unlike `Prim2SF` which is injective, the function `SF2Prim` will typically map many `spec_float` to the same `float`, since `spec_float` enforces no constraint at all on the range of the mantissa and exponents. We thus have the following axioms:

```
Axiom SF2Prim_Prim2SF : ∀ p : float, SF2Prim (Prim2SF p) = p.
Axiom Prim2SF_valid : ∀ p : float, valid_binary (Prim2SF p).
Axiom Prim2SF_SF2Prim : ∀ s : spec_float, valid_binary s → Prim2SF (SF2Prim s) = s.
```

where `valid_binary` expresses bounds on the mantissa and exponent.

Finally, the computational content of all the operators is axiomatized with respect to reference implementations of the algorithms over the `spec_float` type. For example, in the case of the addition, we first define a naive[4] implementation `SFadd` for any precision `prec` and maximal exponent `emax`:

```
Definition SFadd (prec emax : Z) (x y : spec_float) :=
  match x, y with
  | S754_nan, _ | _, S754_nan => S754_nan
  | S754_infinity sx, S754_infinity sy => if Bool.eqb sx sy then x else S754_nan
  | S754_infinity _, _ => x | _, S754_infinity _ => y
  | S754_zero sx, S754_zero sy => if Bool.eqb sx sy then x else S754_zero false
  | S754_zero _, _ => y | _, S754_zero _ => x
  | S754_finite sx mx ex, S754_finite sy my ey =>
    (* let ez be the minimum of ex and ey                               *)
    (* shift mx and my to the left, i.e., mx := mx * 2^(ex - ez)        *)
    (* take signs into account:  mx := (-1)^sx * mx; idem for my        *)
    (* round the result (mx + my) * 2^ez to nearest (according to prec and emax) *)
  end.
```

Then, an axiom relates the axiomatic symbol `add` to this reference implementation instantiated for the `binary64` format (`prec` = 53 and `emax` = 1024):

```
Axiom add_spec : ∀ x y, Prim2SF (add x y) = SFadd 53 1024 (Prim2SF x) (Prim2SF y).
```

[4]The naivety of the approach can be seen in that both mantissas `mx` and `my` are aligned by shifting them left according to the smallest exponent `min ex ey`. A fast implementation would instead shift them right according to the largest exponent, as done in hardware. This is much trickier to get right and the improved performance is not required for our specification purpose.

Users wanting to load all floating-point operations and axioms just need to `Require Import Floats.` In addition, to enjoy decimal literals and notations without explicit quoting, users can `Open Scope float_scope.`

The implementation provides the following floating-point primitives. First, come some arithmetic operations: "`+`", "`-`" (opposite and subtraction), "`*`", "`/`", `abs`, and `sqrt`. All these operations are rounded to nearest, ties breaking to even.

Second, some comparison functions return a boolean result: "`=?`", "`<?`", "`<=?`". These comparison functions comply with the IEEE-754 standard, that is, comparing to a NaN is always false. In addition, there is a generic comparison function "`?=`" that returns a four-valued result: `FEq`, `FLt`, `FGt`, or `FNotComparable` (in case one or both inputs are NaN). We also provide a `classify` function that tells whether a number is NaN, zero, infinite, normal, or subnormal and, except for NaN, what its sign is. Note that there is a lot of redundancy between the comparison functions "`=?`", "`<?`", "`<=?`" and "`?=`" as most could be emulated using the others. It is difficult to choose which ones are more useful than others, so we ended up providing hardcoded reduction rules for all of them.

Finally, there are some dedicated operations to manipulate numbers. The function `of_int63` rounds a 63-bit unsigned integer to the nearest floating-point number, ties breaking to even. Conversely, `normfr_mantissa` takes a number in $[0.5, 1.0)$ and returns its integral mantissa. There are also functions `frshiftexp` and `ldshiftexp` that behave like `frexp` and `ldexp` from the standard library of the C language. The first one decomposes a floating-point number into a mantissa (*i.e.*, a floating-point number f such that $|f| \in [0.5, 1)$) and an integer exponent e. The second one is the converse operation which computes $f \cdot 2^e$. There is a small peculiarity though, as Coq's hardware integers were unsigned at the time hardware floating-point numbers were implemented (signed integers have since been added). So, the exponent e is represented with a bias, to make it nonnegative. The functions `next_up` and `next_down` return the successor and predecessor of a given floating-point value.

The reference implementation amounts to about 500 lines of Coq code in the standard library. But let us recall that the material provided in the `Floats` module is purely algorithmic. It does not axiomatize nor prove any meaningful floating-point properties and is thus basically useless in isolation.

4.2.2 On Flocq's side

Comprehensive formalizations of floating-point arithmetic exist for several proof assistants, *e.g.*, HOL Light [Har99] and PVS [Min95, BM06]. In the case of Coq, the largest formalization is provided by the Flocq library [BM11]. A whole hierarchy of formats is provided, ranging from real numbers with bounded mantissas but unbounded exponents to computable numbers with all the floating-point special values: signed zeros, infinities, and NaNs. Along with these formats and the links between them, the library contains many classical results about roundings and error-free transformations.

When verifying properties of floating-point algorithms, two families of formats are commonly encountered:

- Numbers with an unbounded exponent range, *i.e.*, without underflow nor overflow. Although unrealistic, this model is attractive for its simplicity and commonly used for error bounds [Hig96].
- Numbers with an exponent range only lower bounded, *i.e.*, with underflow but without overflow. This is slightly more realistic, since overflows can often be studied separately, while this is usually much harder for underflows [Rou16].

To make Coq's basic floating-point specification useful, we need to establish a link with one of Flocq's formats, namely the `binary_float` type. This is basically a dependent product of `spec_float` and a proof that the mantissa and exponent are bounded:

```
Inductive binary_float :=
  | B754_zero (s : bool)
  | B754_infinity (s : bool)
  | B754_nan : binary_float
  | B754_finite (s : bool) (m : positive) (e : Z) :
    valid_binary (S754_finite s m e) → binary_float.
```

The Coq theory `Flocq.IEEE754.PrimFloat`[5] provides two functions `Prim2B : float -> binary_float` and `B2Prim : binary_float -> float` that convert back and forth between values of Coq's abstract type `float` and Flocq's concrete type `binary_float`. These conversion functions act as morphisms, as illustrated by the following theorem.

```
Theorem add_equiv : ∀ x y, Prim2B (x + y) = Bplus mode_NE (Prim2B x) (Prim2B y).
```

[5]`https://gitlab.inria.fr/flocq/flocq/-/blob/flocq-4.1.1/src/IEEE754/PrimFloat.v`

In the above theorem, the `+` operator on the left stands for Coq's hardware `float` addition whereas `Bplus mode_NE` on the right stands for Flocq's addition rounded to nearest, with ties breaking to even, which is the default rounding mode of the IEEE-754 standard.

The main purpose of these morphisms is to give access to Flocq's theorems which state that floating-point operations amount to rounding the corresponding operation in the real field $\mathbb{R}$, as mandated by the IEEE-754 standard. For instance, Flocq provides a formal proof of the following theorem:

```
Theorem Bplus_correct : ∀ m x y, is_finite x → is_finite y →
  let z : R := round radix2 fexp (round_mode m) (B2R x + B2R y) in
  if Rlt_bool (Rabs z) (bpow radix2 emax) then B2R (Bplus m x y) = z  (* no overflow *)
  else B2SF (Bplus m x y) = binary_overflow m (Bsign x)
```

This theorem mainly states that, if `x` and `y` are finite floating-point numbers, the real value of their floating-point sum `(Bplus m x y)` is exactly the rounding `z` of the mathematical addition `(B2R x + B2R y)` of `x` and `y` seen as real numbers, assuming the addition did not overflow. The actual theorem in Flocq also gives the sign of the result, which is useful to distinguish $+0$ and -0, but it is omitted here for the sake of conciseness.

By relating the addition of real numbers with the addition of floating-point numbers, this theorem brings confidence in the correctness of the nontrivial bit-level specification of floating-point operations described in Section 4.2.1, at least for finite inputs. For infinite and NaN inputs, exhaustive testing is achievable [BJLM15]. Moreover, this theorem gives access to the extensive set of results proved in the Flocq library. This includes cases where floating-point operations are exact or where the round-off error is represented as a floating-point number or bounded. The latter enables the use of the standard model of floating-point arithmetic to derive bounds on errors of elaborate expressions or algorithms [JR18, Rou16].

Building this link between Flocq's formalization and Coq's specification of hardware floating-point numbers has provided the opportunity to add to the `IEEE754` layer of Flocq several new functions, such as the Boolean comparisons `Beqb`, `Bltb`, and `Bleb`, which complement the already available and more general `Bcompare` function. Some other added functions provide ways to precisely craft or destruct floating-point values from their integer mantissa and exponent: `Bnormfr_mantissa`, `Bldexp`, and `Bfrexp`. Of particular interest for interval arithmetic are the predecessor and successor functions `Bpred` and `Bsucc` as well as the unit in the last place `Bulp`. Finally, the two constants `Bone` and `Bmax_float` are provided for convenience.

In terms of implementation, changes to Flocq required adding 4900 lines and removing 1600 lines.

4.3 Implementation

While the reference implementations described in Section 4.2.1 can effectively perform floating-point operations, they are excruciatingly slow. So, we want to delegate them to hardware units instead. Section 4.3.1 shows how the kernel of Coq was extended to make it possible.

To minimize the trusted computing base, only the default rounding direction of the IEEE-754 standard is supported by Coq, as it remains the most portable one. Unfortunately, applications such as interval arithmetic depend on the availability of directed rounding modes. Section 4.3.2 explains how the functions predecessor and successor can be used instead, and how the performance overhead was mitigated.

While floating-point numbers are usually hidden deep inside proofs of theorems about real numbers, it might happen that the user wants to directly manipulate floating-point numbers. To cover this use case, our implementation also provides some facilities for parsing and printing numbers (Section 4.3.3).

Finally, this whole work would be moot if the implementation did not match the specification described in Section 4.2.1. So, Section 4.3.4 discusses the issue of trust and what has been done to offer the highest level of guarantee.

4.3.1 Reduction engines

As explained in Section 4.2.1, Coq provides an abstract type `float` as well as operations over it, and Coq's version of λ-calculus is extended with dedicated reduction rules for these operations. This means that these rules have to be implemented in the software. Unfortunately, Coq supports various engines, each one with its own implementation of the rules of Coq's calculus.

First, comes the conversion engine, which is responsible for checking that two λ-terms are equivalent according to the rules of the calculus. The conversion engine works great in general, but it falls short when performing a proof by computational reflection. Indeed, in that case, the archetypal goal to prove is $f(x) = \mathit{true}$ for some given x. This is a proof by reflexivity, which means that Coq has to check that the term $f(x) = \mathit{true}$ is equivalent to the term $\mathit{true} = \mathit{true}$. Since true is already in normal form, this amounts to computing the normal form of $f(x)$. But the conversion engine tries hard to never normalize terms, as the size of the normal form explodes in the general case. So, it is unable to handle this use case efficiently.

That is why Coq provides two reduction engines that are solely designed to compute normal forms. The first one, invoked by the `vm_compute` tactic, compiles the λ-term to bytecode and then evaluates it using an interpreter derived from the bytecode interpreter for the OCaml language [GL02]. The second one, invoked by the `native_compute` tactic, follows a similar approach. It turns the λ-term into an OCaml function, compiles it to machine code using the OCaml compiler, and then loads it and executes it [BDG11].

For all three engines, the underlying runtime comes from the OCaml language. As a consequence, we chose to represent floating-point numbers the same way as OCaml, if only to please the garbage collector. Thus, floating-point numbers in Coq are boxed, that is, they are represented by a pointer to an allocated memory cell containing a single floating-point number.

The implementation of the arithmetic operations, however, is not as straightforward, as we cannot just follow OCaml's guidance. Indeed, contrarily to standard OCaml functions, Coq λ-terms can contain free variables, which are irreducible. In turn, even if the type of a λ-term is `float`, it does not mean that its reduced value is a floating-point number, it might be an arbitrary irreducible expression. So, we follow the same approach as in previous works for hardware integer support [AGST10, Dén13]. The implementation first checks if the inputs are actual floating-point numbers. Thanks to the boxing format, this information is readily available. If the inputs are numbers, the floating-point operation is performed. Otherwise, an irreducible term is built from the inputs and the operation.

This causes a discrepancy with the OCaml runtime. Indeed, while floating-point numbers are usually boxed, the runtime optimizes the representation of floating-point arrays, so that they directly store numbers rather than pointers to boxes. This is problematic for Coq, since irreducible terms of floating-point types are not numbers and thus cannot be stored as unboxed values. The original implementation failed to fully circumvent this runtime optimization, which led to an inconsistency, as explained in Section 4.3.4.

4.3.2 Rounding directions

As mentioned above, only rounding to nearest is supported by our implementation. Yet, the IEEE-754 standard specifies some other rounding directions, some of which are especially useful for proofs by computation. Let us illustrate this with interval arithmetic. This arithmetic reliably approximates a real expression x with a pair $(\underline{x}, \overline{x})$ of floating-point numbers that enclose it [Moo63]. Given some enclosures of u and v, an enclosure of $u + v$ is then represented by the pair $(\underline{u} + \underline{v}, \overline{u} + \overline{v})$. To make sure this is an actual enclosure, the lower bound $\underline{u} + \underline{v}$ is computed using a floating-point addition rounded toward $-\infty$, while the upper bound is rounded toward $+\infty$. Indeed, simply rounding to nearest would be unsound; for instance, $1 \in [0; 1]$ and $2^{-80} \in \left[0; 2^{-80}\right]$ whereas the real sum $1 + 2^{-80}$ does not lie in $\left[0 \oplus 0; 1 \oplus 2^{-80}\right] = [0; 1]$, denoting by $\oplus$ the floating-point addition rounded to nearest. A similar approach is used for multiplication, division, and so on.

Unfortunately, while rounding to nearest is readily available in floating-point units, this is not the case for directed rounding. And even if it was, there is no foolproof way of performing floating-point operations with directed rounding in programming languages such as OCaml and C. For instance, as of writing this, the GCC compiler still does not handle dynamically changing the rounding mode in a safe way, let alone an efficient one.[6] So, to ensure portability, we had to stop at rounding to nearest.

But since interval arithmetic is such an ubiquitous paradigm when it comes to proving properties about real numbers, we have provided two more primitives to ease its implementation: predecessor and successor. Indeed, interval arithmetic does not really care whether the bounds of the enclosures are correctly rounded toward $-\infty$ and $+\infty$. If the lower bound is even smaller (or the upper bound larger), the enclosure is still valid, though less tight. In other words, this might cause some proofs to fail, but cannot lead to unsound results. Thus, rather than rounding $\underline{u} + \underline{v}$ toward $-\infty$ to compute the lower bound, an interval library can instead round $\underline{u} + \underline{v}$ to nearest and then take its floating-point predecessor.

The original implementation of these two primitives was a trivial wrapper over the `nextafter` function of the C standard library [BMDR19]. Unfortunately, neither LLVM nor GCC properly optimize it, even if the second operand is an infinite constant. So, we replaced `nextafter` by our own specific implementation,[7] which brought a noticeable speedup.

Still, performance of the bytecode interpreter (`vm_compute`) was poor, due to the boxing of floating-point numbers. Indeed, any arithmetic operation would cause two memory allocations in a row, one for the result of the floating-point operation rounded to nearest, and another one for the predecessor or successor. A simple way to fix it would have been to provide variants of all the arithmetic operations composed either with the predecessor or the successor, that is, ten more floating-point primitives. The C implementation of these new primitives would then be able to elide the first allocation.

But adding ten more primitives to Coq just for the sake of one single library felt wasteful. So, we explored a different approach. Instead of eliding the first allocation, we elide the second one in some specific cases. Indeed, if the first allocation is no longer needed once the predecessor/successor has been computed (and thus

[6] https://gcc.gnu.org/bugzilla/show_bug.cgi?id=34678
[7] https://github.com/coq/coq/pull/12959/commits/ef3ec53e4f74f32a705489b332b037569680d28e

would be collected by the garbage collector), these functions can reuse it to store their own result. This case can easily be detected by the peephole optimizer of the bytecode compiler.[8] As a consequence, immediately computing the predecessor/successor of a floating-point result is now so cheap that removing the calls to these functions from CoqInterval (which would be unsound) does not bring any noticeable speedup, and thus neither would having some dedicated primitives.

4.3.3 Parsing and printing

Parsing and particularly printing the floating-point constants in Coq appeared to be a nontrivial point. Coq basically offers two levels for printing terms. The most commonly used level applies library- and user-defined notations to display concise terms that are as readable as possible. At the lower level, terms are displayed in a raw form to ensure that no details are hidden by some notation. The user can switch between the two levels with the `Set` and `Unset Printing All` commands. Printing of floating-point constants supports both levels.

As suggested by its name, the `binary64` floating-point format, implemented by processors and wired into Coq's hardware floating-point numbers, is a binary format. This means that its finite values are the rational numbers $m \times 2^e$ for bounded integers m and e. In the `binary64` format, the mantissa m is encoded in 53 bits while the exponent e is encoded in 11 bits. Thus, all finite values can be exactly encoded in the standard hexadecimal format `[+-]0x⟨m⟩p[+-]⟨e⟩` where `⟨m⟩` is an hexadecimal encoding of m, spreading on at most 14 characters, and `⟨e⟩` a decimal encoding of e, taking at most 4 characters. This is the way floating-point values are displayed as raw terms, offering an exact and compact display, with at most 23 characters.

Unfortunately, this hexadecimal printing lacks readability for humans, used to work with decimal representations. It is worth noting that 10 being a multiple of 2, any binary value can be exactly represented as a decimal one. Indeed, when $e \geq 0$, the number $m \times 2^e$ is an integer and when $e < 0$, we have $m \times 2^e = (m \times 5^{-e}) \times 10^e$ that is a decimal value since $m \times 5^{-e}$ is an integer. For the `binary64` format, $|m| \leq 2^{53} - 1$ and $-1074 \leq e \leq 971$, which means that, in the first case, the integer $m \times 2^e$ can be represented with at most 309 digits and in the second case, the integer $m \times 5^{-e}$ can be represented with at most 767 digits. Thus, although an exact decimal representation exists, using up to nearly 800 characters to display numeric constants is utterly unpractical. However, it is known that printing values in a decimal format with at least 17 significant digits and implementing parsing as a rounding to nearest guarantees that no information is lost [MBdD+18, Table 3.16]. Printing with 17 decimal digits is thus the choice made in the default printing mode, see [MMR23, §3.3] for more details (including the statement of the Coq proof). This means that one can verify the following script:

```
Goal (0.99999999999999999 = 1)%float.
Proof. reflexivity. Qed.
```

Indeed, the constants `0.99999999999999999` and `1` are the same, as clearly seen when displaying the goal under its raw form: `@eq float 0x1p+0%float 0x1p+0%float`. To avoid any surprise to the user, a warning is displayed by Coq: *The constant 0.99999999999999999 is not a binary64 floating-point value. A closest value 0x1p+0 will be used and unambiguously printed 1.*

4.3.4 Soundness

When we originally reported on the implementation of hardware floating-point numbers in Coq [BMDR19], we had identified three main potential threats to soundness. We recall them below, along with a discussion of the few soundness bugs that were discovered since the merge of the feature in Coq in November 2019. All known bugs are now fixed. It is worth noting that all these bugs pertain more to the usual implementation mishaps than fundamental issues and that barring such implementation bugs, the approach is theoretically sound.

Specification issues A mismatch with respect to the implementation would break the soundness. Of course, such errors in the specification can only be harmful when the offending axioms are used (we recall that all the axioms used in a proved theorem explicitly appear in the result of the Coq command `Print Assumptions`). So far, two such bugs have been reported and fixed:

- incorrect specification of `PrimFloat.leb`,[9]
- inconsistent classification of zeros.[10]

[8] https://github.com/coq/coq/pull/12959/commits/5c7f63fa7a88cf2cb9b6837eb2797268c5843030
[9] https://github.com/coq/coq/issues/12483
[10] https://github.com/coq/coq/issues/16096

Both bugs were due to some typo in the specification, and it should be noted that the former would now be automatically spotted, thanks to a new warning raised by Coq for unused variables in pattern-matching.

As seen in Section 4.2.1, our Coq specification happens to be executable (although this can be pretty slow). This allowed us to add to Coq's test-suite some consistency checks between the specification and the implementation. These checks, however, can obviously not be exhaustive.

Incompatible implementations If evaluation tactics (`native_compute`, `vm_compute`, `compute`, `simpl`, `hnf`, and so on) were to evaluate a same term to different results depending on the hardware, this would lead to a proof of `False`. In particular, it happens that the payload of NaNs is not fully specified by the IEEE-754 standard (different hardwares can produce different NaNs for a same computation), so we have chosen to consider all NaNs as equal and not distinguish them. Thus incompatible bit-level implementations remain compatible at the logical level.

Double roundings due to the legacy x87 unit on old 32-bit architectures could also be harmful [Mon08]. The OCaml compiler systematically relies on it when it is available, which led us to implement all floating-point operators in C for 32-bit architectures, and use the appropriate compiler flags. To double-check the absence of double roundings, we have also added a runtime test[11] to prevent Coq from running in case of miscompilation.

As with the specification, some tests of consistency between the various evaluation mechanisms have been added to Coq's test-suite.

Incorrect convertibility test Distinguishing two values that should not be distinguished, or vice versa, would also be a threat. In particular, implementing the convertibility test using the equality test on floating-point values (as defined in the IEEE-754 standard) would be wrong, as not only NaNs cause issues, but also signed zeroes. Indeed, the standard equates -0 and $+0$, while $1 \div (-0) = -\infty \neq 1 \div (+0) = +\infty$. Fortunately enough, a very simple implementation is feasible; it amounts to the following OCaml code:

```
let equal f1 f2 =
  if f1 = f2 then f1 <> 0. || sign f1 = sign f2 else is_nan f1 && is_nan f2
```

When `f1` and `f2` compare equal in the guard, they are either nonzero, and are then the same floating-point value, or they can be $+0$ or -0 which are distinguished by the `then` branch. Otherwise, `f1` and `f2` are different floating-point values unless they are both NaNs, since `nan = nan` is false by the IEEE-754 equality. The `else` branch thus checks for that case.

Interaction with other primitive types in Coq This can also be a source of soundness issues. To be more precise, unsigned hardware integers were completely reworked in February 2019 (9 months before floating-point numbers); primitive persistent arrays were added in July 2020; finally, signed hardware integers were added in February 2021. First, the fact that the hardware integers used in the specification of hardware floating-point numbers were unsigned did require some care. Next, the way OCaml optimizes arrays of floating-point values did raise a few issues during the development, although it seemed unlikely that such bugs could lead to a proof of `False`. This nonetheless happened with the introduction of primitive persistent arrays that were developed concurrently to hardware floating-point numbers, and whose interactions had not been properly tested before then.[12] Lastly, a similar bug involving the OCaml binary representation of floating-point arrays and the `native_compute` mechanism, but independent from primitive arrays, was discovered and fixed.[13]

4.4 Applications

A few Coq libraries rely on intensive floating-point computations to formally prove properties involving real numbers. We have adapted two of them to take advantage of the hardware support for floating-point arithmetic. The first one is ValidSDP [MR17], which uses floating-point numbers to verify positivity certificates. The adaptation was rather straightforward, since ValidSDP was already formalized in a way that made it robust against underflow and overflow. See [MMR23, §4.1] for more details.

This was not the case for the second library, CoqInterval [MDM16], which uses pairs of floating-point numbers to enclose real numbers. Indeed, CoqInterval was performing computations using a floating-point format with unbounded exponents and, in numerous places, the proofs were implicitly relying on some of its nonstandard properties. Making this formalization compatible with IEEE-754 floating-point numbers required some large and intrusive changes. Since version 4.0, the CoqInterval library uses hardware floating-point numbers by default. This implementation of interval arithmetic lives alongside the original implementation,

[11] `https://github.com/coq/coq/blob/V8.17.0/kernel/float64_63.ml#L28-L35`
[12] `https://github.com/coq/coq/issues/15070`
[13] `https://github.com/coq/coq/issues/17871`

which emulates floating-point numbers using arbitrary-precision integers `bigZ` from the bignums library [GT06]. See [MMR23, §4.2] for details of the adaptation.

The use of hardware floating-point numbers provides a speedup of about one order of magnitude, as illustrated by the following proof of

$$\left|\int_0^8 \sin(t + \exp t)dt - 0.3474\right| \leq 0.1.$$

```
From Coq Require Import Reals.
From Coquelicot Require Import Coquelicot.
From Interval Require Import Tactic.
Open Scope R_scope.

Goal Rabs ((RInt (fun t => sin (t + exp t)) 0 8) - 0.3474) ≤ 0.1.
Proof.
Time integral with (i_fuel 600, i_prec 30).
(* emulated 30-bit FP computations: 32.7s *)
Undo.
Time integral with (i_fuel 600).
(* hardware FP computations: 3.7s; 9x faster! *)
Qed.
```

The main drawback stems from the fact that we do not use directed rounding modes but instead approximate them by composing rounding to nearest with successor or predecessor, as explained in Section 4.3.2. Thus, proofs that rely on correct directed rounding cannot be performed with hardware operations, as illustrated by the following example.

```
Goal 1 + 1 ≤ 2.
interval with (i_prec 1).
(* exact rounding with emulated FP works (even with a single bit) *)
Undo.
Fail interval.
(* but hardware FP fails as next_up(1 + 1) is larger than 2 *)
```

This happened to yield a disastrous effect on proofs involving square roots such as follows.

```
Goal ∀ x, -1 ≤ x ≤ 1 → sqrt (1 - x) ≤ 3/2.
Proof. intros; interval. Qed.
```

Indeed, since the interval computed for `1 - x` now contains negative values (e.g., the predecessor of zero), the interval version of `sqrt` started returning `Inan`, which is the interval containing all the real numbers as well as `Xnan`. We have alleviated this issue by modifying the interval square root so that it ignores negative inputs. This is made possible because the `sqrt` function in Coq's standard library defines `sqrt x = 0` for any negative `x`, rather than some arbitrary result.

Many proofs, however, do not rely on exact rounding and work just as well with hardware floating-point numbers as with emulated ones. For the few cases where exact rounding is required, users can tell the tactics to fall back to emulated numbers by providing an explicit precision with the `i_prec` argument.

4.5 Benchmarks

We have evaluated the benefit of implementing reduction rules using hardware floating-point arithmetic for the applications ValidSDPand CoqInterval libraries, presented in Section 4.4.

The experimental results of the upcoming sections have been obtained using a Debian GNU/Linux workstation based on an Intel Core i7-7700 CPU clocked at 3.60 GHz, with 16 GB of RAM. All the benchmarks have been executed sequentially (namely, without the `-j` option of `make`).

Our benchmarks rely on a large set of dependencies that can take about an hour to compile. For greater convenience, we devised some Docker images containing the benchmark environments, based on Debian 11, opam 2 (the OCaml package manager), and OCaml 4.07.1. The source code of all the benchmarks as well as guidelines to run them are gathered at the following URL: `https://github.com/validsdp/benchs-primitive-floats`.

Source	Emulated	Hardware	Speedup
mat0050	0.228s ±9.5%	0.013s ±16.7%	17.3×
mat0100	1.451s ±2.4%	0.113s ±8.2%	12.9×
mat0150	4.572s ±6.8%	0.276s ±13.0%	16.6×
mat0200	10.724s ±3.4%	0.557s ±9.8%	19.2×
mat0250	21.839s ±1.2%	1.032s ±3.5%	21.2×
mat0300	37.706s ±1.6%	1.810s ±4.7%	20.8×
mat0350	60.616s ±1.5%	2.802s ±4.1%	21.6×
mat0400	89.343s ±1.5%	4.110s ±0.9%	21.7×

Table 4.1: Proof time for the reflexive tactic `posdef_check` using `vm_compute`. Every test is run 5 times. The table indicates the average and relative variability among the timings of 5 runs.

Source	Emulated	Hardware	Speedup
mat0050	0.702s ±1.4%	0.602s ±4.2%	1.2×
mat0100	1.034s ±2.3%	0.674s ±2.5%	1.5×
mat0150	1.735s ±2.7%	0.694s ±2.6%	2.5×
mat0200	3.207s ±5.6%	0.836s ±4.5%	3.8×
mat0250	5.624s ±4.2%	0.924s ±2.2%	6.1×
mat0300	9.359s ±4.1%	1.080s ±3.3%	8.7×
mat0350	14.524s ±3.6%	1.307s ±3.8%	11.1×
mat0400	21.650s ±3.0%	1.641s ±5.3%	13.2×

Table 4.2: Proof time for the reflexive tactic `posdef_check` using `native_compute`. Every test is run 5 times. The table indicates the average and relative variability among the timings of the 5 runs.

4.5.1 Benchmark with ValidSDP 1.0.1 and Coq 8.15

We first run the `posdef_check` tactic on ValidSDP's test-suite and compare its execution time between emulated and hardware floating-point numbers. The results are displayed in Table 4.1 for `vm_compute` and Table 4.2 for `native_compute`.

Notice that the measured speedups are far from the three order of magnitudes separating emulated floating-point operations from equivalent OCaml implementations. From the above results, it appears that arithmetic operators constitute most of the computation time when using emulated numbers (at least 95% with `vm_compute`) but nothing tells us this is still the case with hardware numbers. In fact, with hardware numbers, most of the computation time is spent on manipulating lists as our matrices are implemented using them [CDM13]. This could be improved using primitive "persistent arrays" now that they have been integrated in Coq.

To get an idea of the time actually devoted to floating-point arithmetic in the total proof time of our reflexive tactic, we use the following simple methodology: replace every arithmetic operator over emulated floating-point numbers with a version that uselessly performs the computation twice, then measure the execution time of both the original program (denoted "Op" in Table 4.3) and the modified program (denoted "Op×2"). The difference between both timings gives the cost of performing only the arithmetic operations. In the case of hardware operations, they are performed 1001 times instead, as the difference would otherwise be of the same order of magnitude as the variability of the measured timings. The results are given in Table 4.3 for `vm_compute` and `native_compute`. The tested operations are addition and multiplication, as they constitute the vast majority of the arithmetic computations performed during a Cholesky decomposition. It is worth noting that these speedups should be taken as coarse orders of magnitude rather than precise measurements. Indeed, the time difference "Op×N−Op" also includes the cost of the duplication machinery itself. As a consequence, the speedups are presumably largely underestimated, as this extra cost is far from negligible in the case of hardware floating-point arithmetic ("Op×1001−Op").

4.5.2 Benchmark with CoqInterval 4.5.2 and Coq 8.15

The second benchmark comprises 101 mathematical properties verified using the CoqInterval library. Figure 4.1 shows that, except for the shortest examples, hardware floating-point numbers usually offer a 10× to 20× speedup over emulated computations when the latter are performed at the same precision, i.e., 53 bits.

Earlier versions of CoqInterval, however, were using a default precision of 30 bits, which is sufficient for a large number of examples from the benchmark. As shown in Figure 4.2, this made proofs up to twice faster

Op	compute	Emulated CPU times (Op×2−Op)	Hardware CPU times (Op×1001−Op)	Speedup
add	vm	101.54±1.6% – 77.91±1.2%	163.50±0.5% – 4.12±0.9%	148×
mul	vm	116.68±1.5% – 77.91±1.2%	163.54±0.5% – 4.12±0.9%	243×
add	native	25.08±2.0% – 20.10±4.8%	88.67±2.2% – 1.66±0.9%	57×
mul	native	29.13±1.2% – 20.10±4.8%	92.79±1.7% – 1.66±0.9%	99×

Table 4.3: Computation time for individual operations obtained by subtracting the CPU time of a normal execution from that of a modified execution where the specified operation is performed twice (resp. 1001 times). Every test is run 5 times. The table indicates the average and relative variability among the timings of the 5 runs.

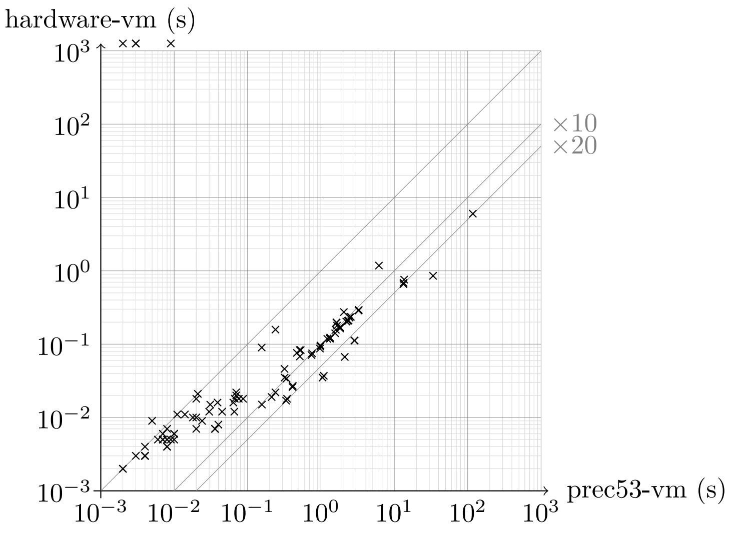


Figure 4.1: Comparison of proof times between hardware and emulated 53-bit floating-point arithmetic using `vm_compute`. The graph uses a log-log scale, so diagonals represent equivalent speedups. Out of the 101 examples, 4 proofs fail with hardware numbers due to the pessimistic outward rounding, as explained in Section 4.4. The corresponding points appear at the top of the graph.

in that case, but nowhere near the speedup achieved using hardware floating-point numbers.

Figure 4.3 shows that using the `native_compute` reduction mechanism instead of `vm_compute` can bring a 3× speedup, but only for the longer examples. Indeed, `native_compute` performs an invocation of the OCaml compiler, which incurs a systematic latency. In particular, hardware floating-point numbers bring such a large speedup over emulated ones that using `native_compute` is often detrimental, except for a handful of examples, as shown in Figure 4.4.

Finally, Figure 4.5 shows that the 10× to 20× speedup from hardware floating-point numbers observed in Figure 4.1 also holds when using `native_compute` instead of `vm_compute`, but only for the longest examples. For shorter examples, the native OCaml compilation dominates the timings, so any speedup brought by hardware floating-point numbers goes unnoticed.

4.6 Conclusion

The work described in this chapter explains how we added support for hardware floating-point numbers to the Coq proof assistant. Formally proving properties of real numbers using floating-point computations is nothing new, but up to now, these computations were slowly emulated in the logic of Coq [MDM16]. Given that modern processors come with a floating-point unit whose semantics are specified by the IEEE-754 standard [IEE08], such an emulation is a waste of computational resources. The same motivation had already led to delegating arithmetic on 31-bit integers (and later 63-bit integers) to hardware units [AGST10]. This work follows a similar approach for floating-point computations: the three conversion/reduction engines of Coq have been extended, so as to use the processor whenever floating-point inputs are not open terms.

While the approach is similar on the implementation side, there is a large difference on the specification side. Indeed, while both integer and floating-point computations are axiomatized using their operational semantics, floating-point arithmetic is so peculiar that one should not blindly believe that the semantics expressed in the logic of Coq matches the behavior of the floating-point hardware. To restore the trust in the formal system, this operational semantics has been proved equivalent to that of the Flocq library, which had already been proved to comply with the IEEE-754 standard [BM11, BJLM15]. But as usual with any software implementation, a few bugs were introduced along the way. Those are now fixed and, barring such implementation bugs, the approach is theoretically sound and does not allow any incorrect proof.

Since the IEEE-754 standard relates floating-point computations to infinitely precise ones, i.e., real numbers, the theorems from Flocq make it easy to use hardware floating-point numbers to formally prove properties on real numbers. There are two main ways to do so. One is to formalize a careful error analysis of floating-point computations, as in the ValidSDP library [Rou16]. The other is to use directed rounding, as in the CoqInterval library [MDM16]. Our work accommodates both approaches. But in the latter case, directed rounding is only approximate, which has forced us to rewrite large parts of CoqInterval to enable the use of hardware floating-point numbers.

Thanks to this work, some proofs by computational reflection have been sped up by a factor 10. While they would have necessarily been run offline before, some of them can now be performed in an interactive setting. This makes for a much friendlier user experience when tactics fail earlier. This speedup comes at the expense of a larger trusted computing base, but the opposite could also be said of this work. Indeed, the speedup is so large that the point of the `native_compute` mechanism becomes largely moot for this use case, thus making it possible to greatly reduce the trusted computing base.

Hardware floating-point arithmetic is not a panacea though, as the numbers are constrained to a bounded range of exponents and, more importantly, a precision of 53 bits. Whenever a higher precision is needed, the tactics provided by, e.g., CoqInterval have to fall back to emulating floating-point arithmetic. But this problem is nothing new, and whichever solution the scientific computing community comes with, we hope it can be adapted to a proof assistant.

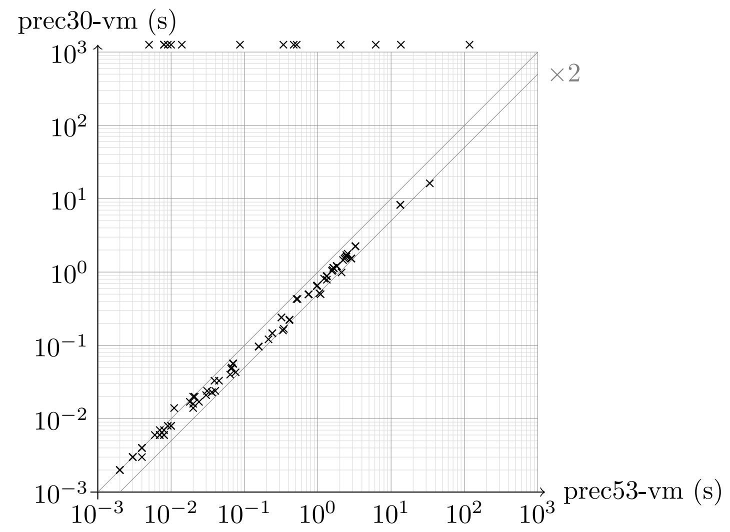


Figure 4.2: Comparison of proof times between emulated 53-bit and 30-bit floating-point arithmetic using `vm_compute`. Out of the 101 examples, 14 proofs fail with the reduced precision.

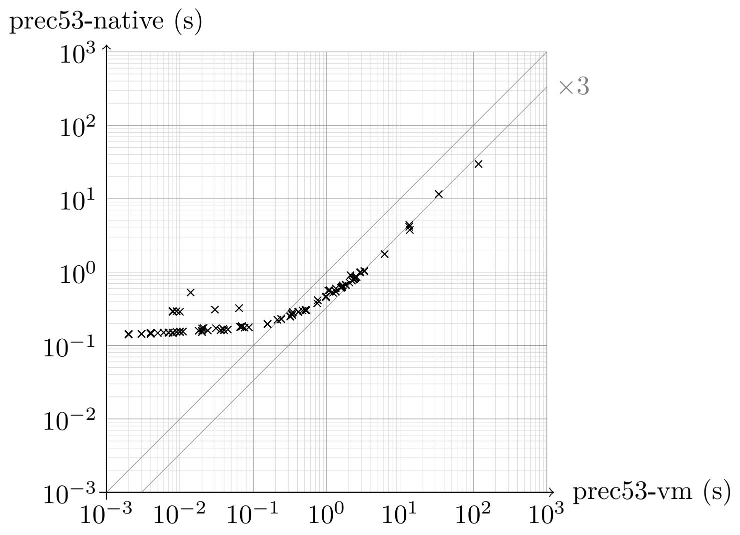


Figure 4.3: Comparison of proof times for emulated 53-bit floating-point arithmetic between `vm_compute` and `native_compute`.

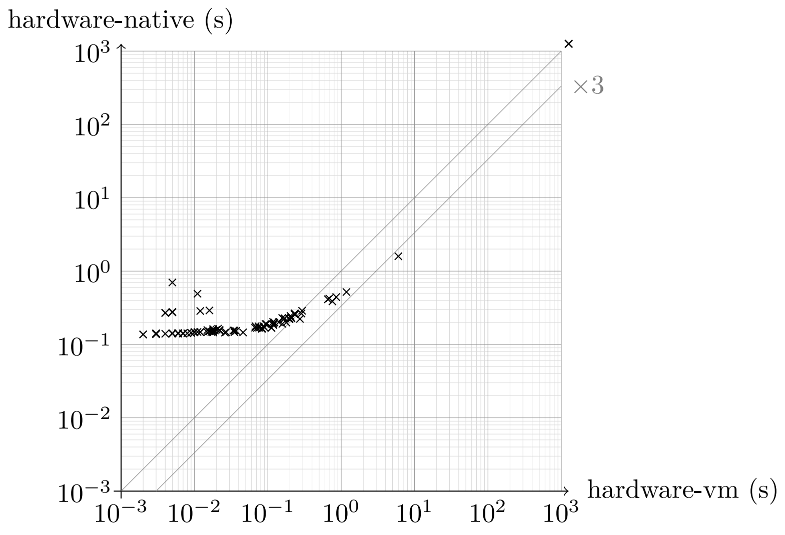


Figure 4.4: Comparison of proof times for hardware floating-point arithmetic between `vm_compute` and `native_compute`. The 4 proofs that fail in both cases appear as a cluster at the top right of the graph.

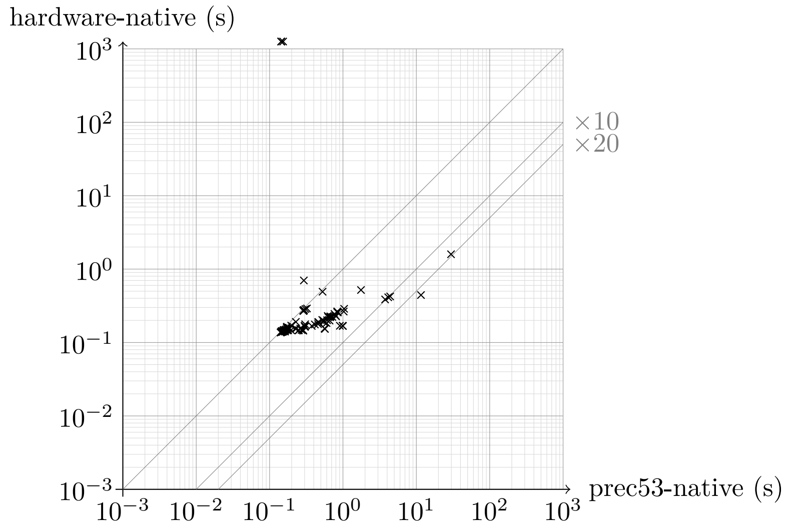


Figure 4.5: Comparison of proof times between hardware and 53-bit emulated floating-point arithmetic using `native_compute`. The 4 proofs that fail with hardware numbers appear as a cluster at the top of the graph.

Chapter 5

Verifying Network Calculus

This chapter presents work done in collaboration with Marc Boyer and Lucien Rakotomalala. Marc is a colleague at ONERA and expert in Network Calculus who coadvised with me the PhD thesis of Lucien. Contrary to the three previous chapter that offered some logical progress, this chapter is more orthogonal. Nevertheless, we'll see that this new subject was an opportunity to reuse some of the technologies introduced in the previous chapters.

5.1 Real-Time Networks

For some critical applications, such as in the aerospace industry, it is required to guarantee that embedded networks meet some constraints on traversal time, as well as the absence of buffer overflow. That is, we want to ensure that a packet emitted on the network will reach its destination within a given time bound and won't be dropped due to some buffer overflow in any intermediate switch. Figure 5.1 shows a basic network example.

There are two main approaches to real-time networks:

time-triggered networks are networks in which the emission date of each packet is statically precomputed[1] in order to ensure that no pair of packets will ever be sent at a time that could yield a conflict between them. Computing this schedule can be a complex task but is done offline and the network then only needs to apply the resulting schedule. However, for the precomputed schedule to work flawlessly, all the elements of the network must share a common clock. Some clock synchronization protocol is thus required to adjust each element clock, that tends to naturally drift.

rate constrained networks are networks in which emission rates of each emitter are bounded. Some static analysis is then used to ensure that, even when packets conflict, this doesn't result in excessive delays. No global clock is thus required as it is enough for the static analysis to accommodate some bounded clock-drift that can have a small impact on emission rates.

Network calculus [LBT01, BBLC18] is one such static analysis. The method is used to certify networks such as AFDX networks (Avionics Full DupleX, a kind of extension of ethernet) used in large modern commercial aircrafts [BNF12]. The remaining of the current chapter will focus on this method. First Section 5.2 will give an overview of how we formalized network calculus in Coq, then Section 5.3 will explain how we automatically verified in Coq the specific computations performed during network-calculus analyses.

5.2 Formalizing Network Calculus in Coq

The first part of Lucien's thesis was dedicated to formalizing in Coq the network calculus theory. All the basic concepts were formalized and a few archetypal theorems were formally proved. This mostly proved uneventful besides a few statements that needed to be made more precise or fixed. The current section will presents such a theorem fix, along with the basic definitions required to state it.

5.2.1 Concrete Model

Network calculus models the amount of data that goes through each network point by a function from time $\mathbb{R}_+ := \{x \in \mathbb{R} \mid x \geq 0\}$ to amount of data $\mathbb{R}_+$. The delay experienced by a flow between two network points will then be the maximum horizontal deviation between the two functions. This is illustrated on Figure 5.2.

[1]Networks can usually run for an unbounded amount of time, but do this on a periodic schedule, so it's enough to plan for a single period.

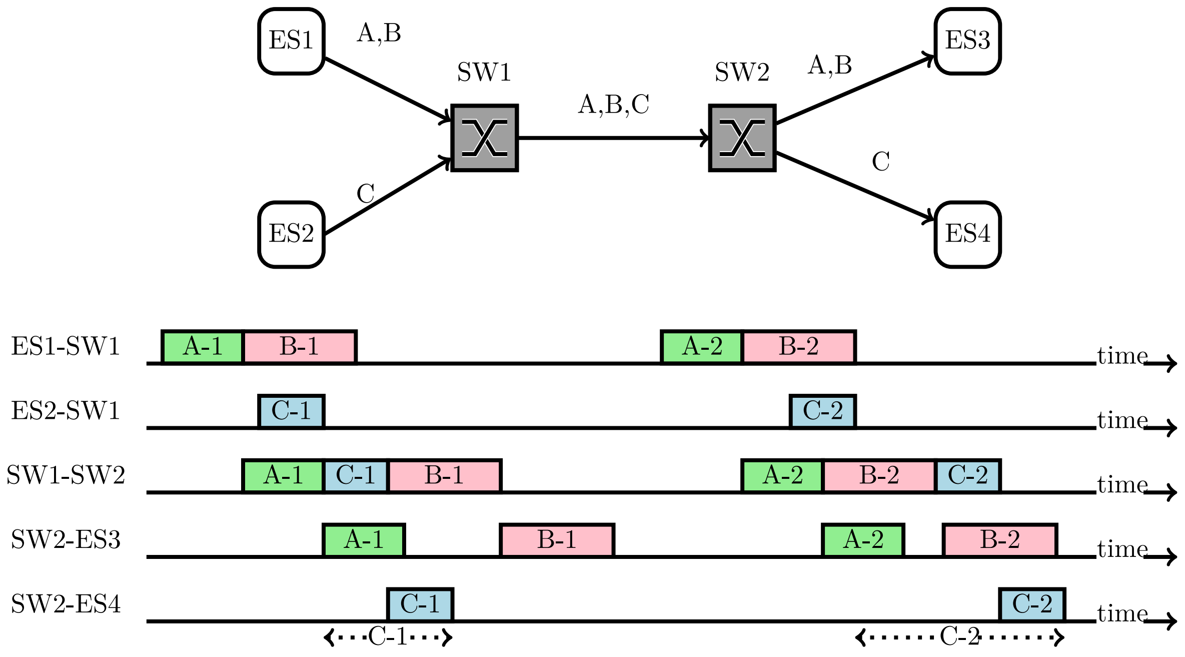


Figure 5.1: Example of a basic network with four end-systems (white rounded squares) connected via two switches (grey squares). There is a conflict between packets B-1 and C-1 that want to cross the SW1-SW2 link at the same time. The switch SW1 has to send one of the two first and keep the other in some internal buffer until the link is available again. Here C-1 goes before B-1 whereas later B-2 will go before C-2. This leads to different delays (represented as dotted arrow on the bottom of the figure) for packets C-1 and C-2. Our goal is to guarantee such delays remain below a given bound. (Figure by Marc Boyer)

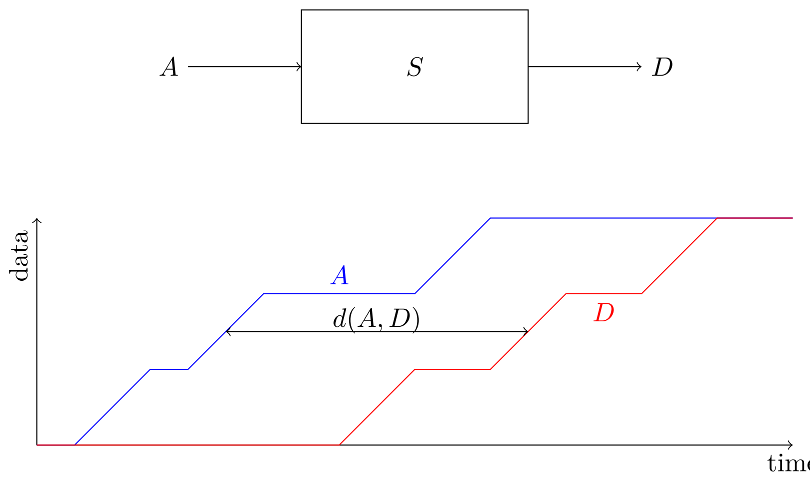


Figure 5.2: Example of arrival flow A and departure flow D for a server S. The delay between A and D is the horizontal deviation between the two curves.

Definition 1. (`F` in our Coq development) $\mathcal{F} := \mathbb{R}_+ \to \overline{\mathbb{R}}$ *is the set of functions from* $\mathbb{R}_+$ *to* $\overline{\mathbb{R}}$.

With $\overline{\mathbb{R}} := \mathbb{R} \cup \{ -\infty, +\infty \}$. Our Coq development can be found at `https://gitlab.mpi-sws.org/proux/nc-coq`. The real numbers are taken from the MathComp Analysis library [ACK+20]. An initial version used the Coquelicot library [BLM15] but since we were already using the MathComp library, for instances for the Σ summations or for algebra, using Analysis proved much more consistent and convenient, with nicer and more uniform definitions and lemmas, as well as some nice automation, for instance for nonnegative numbers, enabling shorter proofs overall.

Definition 2. (`Fplus`) $\mathcal{F}_+ := \{f \in \mathcal{F} \mid 0 \leq f\}$ *is the subset of functions from* $\mathcal{F}$ *that are nonnegative.*

Definition 3. (`Fup`) $\mathcal{F}^\uparrow := \{f \in \mathcal{F}_+ \mid \forall xy, x \leq y \Rightarrow f(x) \leq f(y)\}$ *is the subset of nondecreasing functions from* $\mathcal{F}_+$.

Now equipped with these basic definitions, we can define the main object of network calculus: data flow cumulative curves. These are the A and D functions in the example of Figure 5.2.

Definition 4. (`flow_cc`) *A* cumulative curve *is a function* $f \in \mathcal{F}^\uparrow$ *satisfying*

- $f(0) = 0$
- f *is left continuous*
- f *only takes finite values:* $\forall t, f(t) \in \mathbb{R}_+$

We denote $\mathcal{C}$ *the set of cumulative curves.*

Network calculus defines delays using the notion of horizontal deviation between two cumulative curves.

Definition 5. (`hDev_at`, `hDev`) *For* $f, g \in \mathbb{F}$ *and* $t \in \mathbb{R}_+$, *the horizontal deviation* $hDev(f, g, t) \in \overline{\mathbb{R}}_+$ *between* f *and* g *at* t *is defined as*

$$hDev(f, g, t) := \inf \{d \in \mathbb{R}_+ \mid f(t) \leq g(t + d)\}$$

and the horizontal deviation $hDev(f, g) \in \overline{\mathbb{R}}_+$ *between* f *and* g *is defined as*

$$hDev(f, g) := \sup \{hDev(f, g, t) \mid t \in \mathbb{R}_+\}.$$

Finally, *servers* constitute the last main notion of network calculus to model concrete behaviors.

Definition 6. (`partial_server`) *A* partial server S *is a relation on* $\mathcal{C}$ *(i.e.,* $S \subseteq \mathcal{C} \times \mathcal{C}$*) satisfying the following constraint*

- $\forall A\, D, (A, D) \in S \Rightarrow D \leq A$

The constraint expresses the fact that, at any time, there cannot be more data that departed the server than data that arrived in it.

Definition 7. (`server`) *A* server S *is a partial server satisfying the additional constraint*

- $\forall A, \exists D, (A, D) \in S$

This second constraint means that for any input, there is, at least one, output. Servers crossed by n flows are defined similarly.

Definition 8. (`nserver`) *Given* $n \in \mathbb{N} \setminus \{ 0 \}$, *a* n*-server* S *is a relation on* $\mathcal{C}^n$ *(i.e.,* $S \subseteq \mathcal{C}^n \times \mathcal{C}^n$*) satisfying the following two constraints*

- $\forall A\, D, (A, D) \in S, \forall i, \Rightarrow D_i \leq A_i$
- $\forall A, \exists D, (A, D) \in S$

It will sometime be convenient to see a n-server as a simple server for the sum of the flows that cross it.

Definition 9. (`aggregate_server`) *Given a* n*-server* S, *its aggregate server is the server*

$$\left\{ (A', D') \;\middle|\; \exists A\, D, (A, D) \in S \wedge A' = \sum_i A_i \wedge D' = \sum_i D_i \right\}.$$

Conversely, we will sometime need to "extract" a server for one of the flows of a n-server.

Definition 10. (`residual_server`) *Given a* n*-server* S, *the* i*th* residual server *of* S *is*

$$\{(A', D') \mid \exists A\, D, (A, D) \in S \wedge A' = A_i \wedge D' = D_i\}.$$

5.2.2 Contracts

Whereas cumulative curves $\mathcal{C}$ model the concrete behaviors of the studied networks, there are usually infinitely many such behaviors and it is not practical to directly manipulate them. Thus, instead of dealing with sets of concrete behaviors in $\mathcal{C}$, network calculus abstract them using *arrival curves*.

Definition 11. (`is_maximal_arrival`) *A function $\alpha \in \mathcal{F}$ is an* arrival curve *for some cumulative curve $A \in \mathcal{C}$ when*

$$\forall t, d \in \mathbb{R}_+,\ A(t+d) - A(t) \le \alpha(d).$$

This will later be denoted arrival (A, α).

While arrival curves are used as contract for emissions of the end systems, we also need a way to express the minimal performances of the network switches. We will use *service curves* to that end.

Definition 12. (`backlog_itv`) *Given $A, D \in \mathcal{F}$, an interval $I \subseteq \mathbb{R}_+$ is a* backlog interval *for A and D when: $\forall x \in I, D(x) < A(x)$. We will denote this backlog(A, D, I).*

Definition 13. (`strict_min_service`) *Given $S \subseteq \mathcal{C} \times \mathcal{C}$, a function $\beta \in \mathcal{F}$ is a* strict service curve *for S when*

$$\forall A\, D, (A, D) \in S \Rightarrow \forall u, v \in \mathbb{R}_+, u \le v \Rightarrow \mathit{backlog}\,(A, D, (u, v]) \Rightarrow \beta(v - u) \le D(v) - D(u).$$

Intuitively, on any backlog interval $(u, v]$ (i.e., any interval on which more data has been received than emitted by the server), the server must emit at least $\beta(v - u)$.

Given those definitions, the network calculus method will then manipulate arrival and service curves according to various theorems to propagate constraints through the analyzed network. Then, for a given flow, another theorem will state that the horizontal deviation between the computed arrival curves at initial and end point of the flow is an upper bound for the horizontal deviation between any cumulative curves that satisfies the considered contracts.

Remark 10. *This looks a lot like static analysis of programs by abstract interpretation where the cumulative curves A, D are the concrete world and the arrival curves α are the abstract world. In practice, most embedded networks are such that there is no need for any actual fixpoint computation, propagating constraints from emission points of the flows to their destinations (e.g., from left to right on the example of Figure 5.1) is enough. That's called "absence of circular dependency". I'd still like to write a paper "network calculus is abstract interpretation", not just to be pedantic, not just because it would be fun (networks make for quite an unusual "programming language" to analyze) but also because I believe this could open the door to some crossfeeding between the two approaches, particularly in the case of circular dependencies where network calculus might benefit from the many widening strategies developed in abstract interpretation. For instance, it would be interesting to study the applicability of the policy iteration techniques mentioned in Chapter 2. For real-time calculus, an approach similar to network calculus, the paper [JPTY08] identifies many of the elements of abstract interpretation but falls short of naming abstract interpretation. In particular, they miss the widening method to overapproximate fixpoints and only deal with convergence in the mathematical analysis understanding of the term, without explaining how to actually compute sound postfixpoints.*

For instance, here is one of those theorems for the case of static priority servers.
Let's first define static priority servers.

Definition 14. (`preemptive_SP`) *Let $S \subseteq \mathcal{C}^n \times \mathcal{C}^n$ be a n-server and prior : $\{1, \ldots, n\} \to \mathbb{N}$, the server S is a* preemptive static-priority server *when*

$$\forall A\, D, (A, D) \in S \Rightarrow \forall i, \forall s, t \in \mathbb{R}_+, s \le t \Rightarrow \mathit{backlog}\left(\sum_{j \prec i} A_j, \sum_{j \prec i} D_j, [s, t]\right) \Rightarrow D_i(s) = D_i(t)$$

noting $j \prec i$ for prior$(j) <$ prior(i).

Intuitively, for each flow i, whenever there is some backlog for the (sum of) higher-priority flows, nothing gets out on i (i.e., D_i remains constant on the interval).

Theorem 5. (Theorem 7.6 in [BBLC18]) *Let prior : $\{1, \ldots, n\} \to \mathbb{N}$ be an injective function and S be a preemptive static priority n-server offering an aggregate strict service curve β. Suppose that for all $i \in \{1, \ldots, n\}, \alpha_i \in \mathcal{F}$ is an arrival curve for each flow i. Then, β_i is a strict service curve offered to flow i, with*

$$\beta_i := \left[\beta - \sum_{j \prec i} \alpha_j\right]_{\uparrow}^{+}.$$

First, when trying to formalize the statement, we notice that the scope of the universal quantification of the flows A, in the definition of service curve, makes it impossible to express that the service curve β_i is only valid when the A_i satisfy some arrival curve α_i. For that, we first need a new definition of residual server under constraints.

Definition 15. (`residual_server_constr`) *Given a n-server S and a n-tuple $\alpha \in \mathcal{F}^n$, the ith residual server of S constrained by α is the partial server*

$$\{(A', D') \mid \exists A\, D, (A, D) \in S \wedge A' = A_i \wedge D' = D_i \wedge \forall j, arrival\,(A_j, \alpha_j)\}.$$

Let's try again

Theorem 6. (next attempt at static priority) *Let $prior : \{1, \dots, n\} \to \mathbb{N}$ be an injective function and S be a preemptive static priority n-server whose aggregate server satisfies a strict service curve $\beta \in \mathcal{F}_+$, for $\alpha \in \mathcal{F}_+^n$, the function*

$$\left[\beta - \sum_{j \prec i} \alpha_j\right]_{\uparrow}^{+}$$

is a strict service curve for the ith residual server of S constrained by α.

Now, we can write that statement in Coq. Unfortunately, the proof in the book looks perfectly fine on paper but when trying to replicate it in Coq, this appeared impossible and made us discover that the theorem is in fact not exactly a theorem, as shown by the following counterexample.

Example 14. *Consider the 2-server $S := \{((A_1, A_2), (D_1, D_2))\}$ with $A_1 := step(2)$, $D_1 := step(4)$, $A_2 := step(1)$ and $D_2 := step(2)$ with $step(x)$ defined as $t \mapsto 0$ when $t \leq x$ and $t \mapsto 1$ otherwise. One can check that this is indeed a preemptive static priority server with the priority $1 \prec 2$. Let's also consider β defined as*

$$d \mapsto \begin{cases} 0 & \textit{when}\ d < 2 \\ 1 & \textit{otherwise.} \end{cases}$$

Note that β is right-continuous. One can check that β is a strict service curve for the aggregate server of S. Now, for α constant functions equal to $+\infty$ (we don't really need them here), the theorem states (for $i = 1$) that β should be a service curve for the first residual server (i.e., $\{(A_1, D_1)\}$) which is not the case: the interval $(2, 4]$ is a backlog interval but $D_1(4) - D_1(2) = 0 < 1 = \beta(4 - 2)$.

Fortunately enough, assuming that β is left-continuous is enough to fix the theorem. This assumption is in practice almost always satisfied by all service curves considered when applying the network calculus method to real case studies, which is probably why the error went unnoticed for so many years. Here is finally the theorem that we managed to prove in Coq.

Theorem 7. (`SP_residual_service_curve`) *Let $prior : \{1, \dots, n\} \to \mathbb{N}$ be an injective function and S be a preemptive static priority n-server whose aggregate server satisfies a strict service curve $\beta \in \mathcal{C}$, for $\alpha \in \mathcal{F}_+^n$, the function*

$$\left[\beta - \sum_{j \prec i} \alpha_j\right]_{\uparrow}^{+}$$

is a strict service curve for the ith residual server of S constrained by α.

5.3 Verifying min-plus Computations

It's not obvious in Theorem 7, which involves mostly additions and subtraction, but network calculus is mostly based on tropical algebra, more precisely the min-plus dioid of functions on real numbers (used to represent both time and amounts of data). Thus, as an intermediate step in any analysis, the method produces algebraic formulas in this dioid, whose computation eventually gives actual numerical bounds. Soundness of the bounds then crucially relies on both the soundness of the network calculus theory and of those computations. Soundness of the theory was adressed in the previous Section 5, we will focus here on verification of computations of algebraic operators in the min-plus dioid of functions.

Efficient algorithms are known for these computations and a few effective implementations do exist [BCG$^+$09, BT08, BMF11]. However, these algorithms are rather tricky, hence the interest in formal proofs to greatly increase the level of confidence in their results. We use the proof assistant Coq [Coq24] to provide formal proofs of correctness of such results. This section essentially sums up our NFM 2021 paper [RRB21].

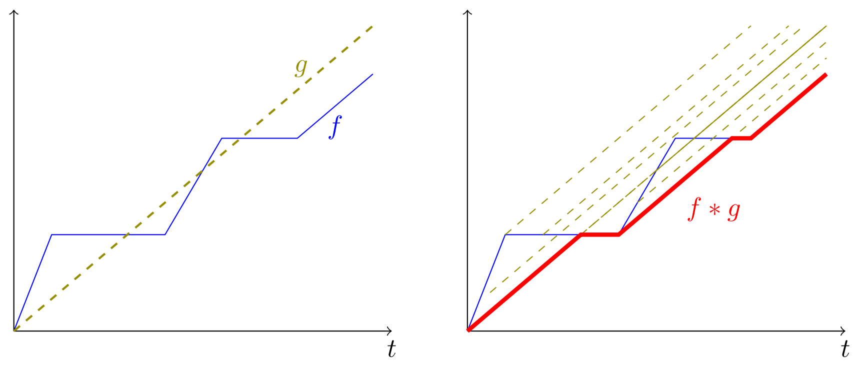


Figure 5.3: Two functions f, g (on the left) and their convolution $f * g$ (on the right). Intuitively, the convolution of two functions can be obtained by sliding one function along the other and taking the minimum hull.

Sections 5.3.1 and 5.3.2 introduce a few notations and give an overview of the objects and operations manipulated throughout the remaining of the current chapter. Then, Section 5.3.3 recalls the state of the art. Sections 5.3.4 and 5.3.5 detail the formalization of these objects, while Sections 5.3.6 and 5.3.7 prove some of their fundamental properties. Finally Section 5.3.8 proves the core soundness arguments of the expected verifiers, Section 5.3.9 discusses the implementation and Section 5.3.10 concludes.

5.3.1 Notations

As already used above, $\mathbb{R}$ will denote the real numbers, $\mathbb{R}_+ := \mathbb{R} \cap [0; +\infty[$ and $\overline{\mathbb{R}} := \mathbb{R} \cup \{-\infty, +\infty\}$. Similarly, let $\mathbb{Q}$ denote the rational numbers, $\mathbb{Q}_+ := \mathbb{Q} \cap [0; +\infty[$ and $\mathbb{Q}_+^* := \mathbb{Q}_+ \setminus \{0\}$. Let $\mathbb{N}$ denote the natural integers, $\mathbb{N}^* := \mathbb{N} \setminus \{0\}$. For any finite set S, let $\#S \in \mathbb{N}$ denote its cardinal and for any sequence s, $\mathit{last}(s)$ denote its last element.

In Coq code appearing below, `nat` will stand for $\mathbb{N}$, `R` for $\mathbb{R}$, `\bar R` for $\overline{\mathbb{R}}$, `{nonneg R}` for $\mathbb{R}_+$, `rat` for $\mathbb{Q}$, `{nonneg rat}` for $\mathbb{Q}_+$, `{posnum rat}` for $\mathbb{Q}_+^*$ and `&&` for logical conjunction $\wedge$.

We also use some list manipulating functions of Coq: `nth`, `head` and `last`. `nth x0 l i` returns the element of index `i` (starting at 0) of the list `l` or `x0` if `l` contains less than `i` elements. `n.+1` and `n.-1` are the successor and the predecessor of any natural number `n` (the predecessor of 0 is 0). The notation `%/` is used for euclidean division. To ease readability of the Coq code, we omit scope annotations below. For each result, we give the name of its Coq implementation: for instance `F_UPP` for Definition 16 below. The code is available at `https://gitlab.mpi-sws.org/proux/nc-coq/-/tree/master/minerve`.

5.3.2 *(min, plus)* Operators on Functions

Network calculus handles functions in $\mathcal{F}$ and uses *(min, plus)* operations over this set: addition, minimum, convolution and deconvolution. We assume[2] that $+\infty + -\infty = +\infty$. We first present these operators. Then, we introduce sub-classes of $\mathcal{F}$ stable for these operators and amenable for effective computations.

(min, plus) Operators

The addition $f + g$ and the minimum $\min(f, g)$ of two functions f and g of $\mathcal{F}$ are pointwise extensions of the corresponding operators on $\overline{\mathbb{R}}$, that is $f + g \ := \ t \mapsto f(t) + g(t)$ and $\min(f, g) \ := \ t \mapsto \min(f(t), g(t))$. We also use two operators, the convolution $f * g$ and the deconvolution $f \oslash g$ that are not pointwise operators, defined as:

$$f * g \ := \ t \mapsto \inf_{\substack{u,v \geq 0 \\ u+v=t}} (f(u) + g(v)), \qquad f \oslash g \ := \ \inf \{h \mid f \leq h * g\}. \tag{5.1}$$

where inf on a set $S \subseteq \mathcal{F}$ is defined as $\inf\{S\} \ := \ t \mapsto \inf\{f(t) \mid f \in S\}$. On Figure 5.3, we plot an example of convolution. More details can be found in [BBLC18, §2].

Sub-classes of Functions for Effective Computation

network calculus tools do not manipulate the complete $\mathcal{F}$ set of functions but only subclasses with good stability properties and effective computations [BT08].

[2] In the min-plus semiring, min is the first (additive) operator and + the second (multiplicative) one, the neutral element of the first operator (the zero), here $+\infty$, should then be absorbing for the second.

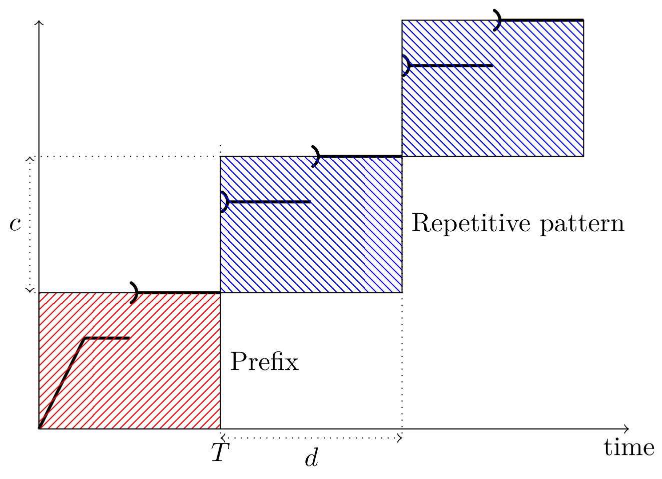


Figure 5.4: Example of function in $\mathcal{F}_{\text{UPP-PA}}$. An initial part appears in the interval $[0, T]$, then a periodic part of length d is repeated every period d while being translated vertically by some increment c. To record such a function, it is then enough to keep its representation on $[0, T + d]$.

In network calculus, it is quite common to have periodic behaviors. To describe them, we use functions that are ultimately pseudo-periodic (UPP), denoted $\mathcal{F}_{\text{UPP}}$. A function $f \in \mathcal{F}$ belongs to the subset $\mathcal{F}_{\text{UPP}}$ if, given a point $T \in \mathbb{Q}_+$ (called initial segment), a period $d \in \mathbb{Q}_+^*$ and an increasing element $c \in \mathbb{Q}$, it holds, for all $t > T$ that $f(t + d) = f(t) + c$. To store a concrete description of such a function, it is then enough to store the values of T, d and c and the description of the function on the initial segment plus one period.

We consider the sub-class of $\mathcal{F}$ made of the Piecewise Affine (PA) functions, denoted $\mathcal{F}_{\text{PA}}$. For these functions, it is sufficient to give, for each segment, the point of discontinuity on one side of the segment, the slope and the offset on the segment. These parameters can be recorded in a list, although this list can be infinite.

We define $\mathcal{F}_{\text{UPP-PA}} := \mathcal{F}_{\text{UPP}} \cap \mathcal{F}_{\text{PA}}$. Its elements can be finitely represented by giving T, d and c from $\mathcal{F}_{\text{UPP}}$ and the initial segment of the list from $\mathcal{F}_{\text{PA}}$ representing the function on $[0; T + d]$. This is illustrated on Figure 5.4.

Our initial idea, for instance with the addition, was to let some external tool provide three functions f, g and h while claiming that $f + g = h$, and then check this relation with a finite number of tests. To this end, we would prove that checking the equality $f(t_i) + g(t_i) = h(t_i)$ on a set of points $t_1, \dots, t_n$, plus some compatibility tests on initial segments, periods and increments, is enough to ensure the equality on $\mathbb{R}_+$. However, in practice, it is just as easy to recompute the sum $f + g$ as to compute the set of points $t_1, \dots, t_n$, so we simply implement and prove the addition operator in Coq. We similarly implemented minimum and convolution.

Regarding the deconvolution, in practice network calculus only requires, given two functions f and g, a function h such that $h \geq f \oslash g$. It is then enough to check that $h * g \geq f$, that is $\min(f, h * g) = f$ which only involves already implemented minimum and convolution.

5.3.3 State of the Art

Two main classes of curves are used in network calculus: the set of concave or convex piecewise linear functions, C[x]PL [SCP99], and the, strictly larger, set of ultimately pseudo-periodic piecewise linear functions UPP-PA, commonly known as UPP [BT08].

The class of the CPL linear functions has nice mathematical properties: it is stable under the addition and the minimum, and moreover, the convolution can be implemented as a minimum plus a constant. The data structure and related algorithms are so simple that they, to our knowledge, have never been published. The class of convex piecewise linear functions has very similar properties, replacing minimum by maximum, and its (min,plus) convolution can also be implemented very efficiently [BBLC18, Sect. 4.2]. Nevertheless, they cannot accurately model packetized traffic, whereas the UPP-PA class gives better results at the expense of higher computation times [BMF11].

An open implementation of the operators on the C[x]PL class can be found in the DISCO network calculus tool [BS14].

The algorithms of the operators on the UPP-PA class are given in [BT08]. An open implementation has been developed but is no longer maintained [BCG$^+$09] to our knowledge. An industrial implementation exists, which is the core of the network calculus tool PEGASE [BNOT10]. The UPP-PA implementation can be accessed through an on-line console [MPC].

The Real-Time Calculus toolbox (RTC) does performance analysis of distributed real-time systems [Wan06, WT06]. Its kernel implements minimum, sum, and convolution on Variability Characterization Curves (VCC's), a class very close to UPP-PA, but no explicit comparison of those two classes has been done up to now.

None of these implementations were formally proved correct.

The first works on the formal verification of network calculus computation were presented in [MBFM13]. The aim was to verify that a tool was correctly using the network calculus theory. An Isabelle/HOL library was developed, providing the main objects of network calculus (flows and servers, arrival and service curves) and the statement of the main theorems, but not their proofs. They were assumed to be correct, since they had been long established in the literature. Then, the tool was extended to provide not only a result, but also a proof on how that network calculus has been used to produce this result. Then, Isabelle/HOL was in charge of checking the correctness of this proof.

Another piece of work, presented in [RBR19] and summarized in previous Section 5.2, consists in proving, in Coq, the network calculus results themselves: building the min-plus dioid of functions, the main objects of network calculus and the main theorems (statements and proofs).

The PROSA library also provides proofs of correctness for the response time of real-time systems, but focuses on scheduling tasks for processors [CSB16].

5.3.4 Ultimately Pseudo Periodic Functions

We now present the formal definition of the set of UPP functions.

Definition 16. (Ultimately Pseudo Periodic Functions, `F_UPP`) *$\mathcal{F}_{UPP}$ is the set of functions $f \in \mathcal{F}$ such that there exists $T \in \mathbb{Q}_+$, $d \in \mathbb{Q}_+^*$ and $c \in \mathbb{Q}$ for which*

$$\forall t \in \mathbb{R}_+, t > T \Rightarrow f(t+d) = f(t) + c. \tag{5.2}$$

Remark 11. *The values of T, d and c could have been in $\mathbb{R}$. However, we know from [BT08] that $\mathcal{F}_{UPP}$ is stable over more operators if T, d and c are rationals. It is not a practical restriction since $\mathbb{Q}$ is the set used in computation.*

We represent $\mathcal{F}_{\text{UPP}}$ in Coq as follows.

```
1 Record F_UPP := {
2   F_UPP_val :> F;
3   F_UPP_T : {nonneg rat};  F_UPP_d : {posnum rat};  F_UPP_c : rat;
4   _ : ∀ t : {nonneg R}, ratr F_UPP_T < t →
5       F_UPP_val (t%:num + ratr F_UPP_d)%:nng = F_UPP_val t + (ratr F_UPP_c)%:E }.
```

This code means that a value of type `F_UPP` is:

line 2 a function `F_UPP_val` of type `F`, i.e., from `{nonneg R}` to `\bar R`. The notation `:>` is a Coq notation for coercion: Coq introduces automatically `F_UPP_val` whenever we give a value of type `F_UPP` whereas a function from `{nonneg R}` to `\bar R` is expected.

line 3 `F_UPP_T`, `F_UPP_d` and `F_UPP_c`, the three parameters T, d and c of (5.2).

lines 4 and 5 the property (5.2). We use `ratr` to cast a rational as a real. The other `%:` things are merely casts too: `%:num` casts from `{nonneg R}` to `R` while `%:nng` is the reverse cast[3] and `%:E` is a cast from `R` to `\bar R`.

The command `Record` creates a constructor of `F_UPP` named `Build_F_UPP`. To declare a value in `F_UPP`, Coq will then require a function of type `F`, three parameters and a proof of (5.2).

5.3.5 UPP and Piecewise Affine Functions

We briefly presented in section 5.3.2 the set $\mathcal{F}_{\text{UPP-PA}}$ of functions that are both UPP and PA. We give in this section a formal definition.

In [BT08], this set was introduced as the intersection of two sets of functions: $\mathcal{F}_{\text{UPP}}$ and $\mathcal{F}_{\text{PA}}$, the set of PA functions. Here, we rather choose to formalize the subset of functions in $\mathcal{F}_{\text{UPP}}$ that are PA, as this greatly simplifies the formalization, by avoiding any requirement for infinite lists (a.k.a., streams).

To define PA functions, we need to record points of discontinuities and change of slopes, we call this data structure *jump sequences*.

[3]It automatically infers a proof of nonnegativity, here because the two terms of the sum are nonnegative themselves.

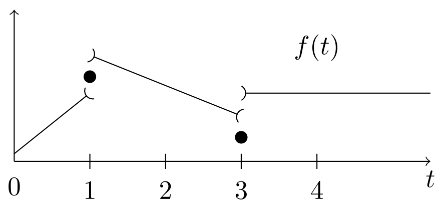


Figure 5.5: The function f is piecewise affine. $a := \{0, 1, 3\}$ and $b := \{0, 1, 2, 3\}$ are JS of this function: $a \in JS(f)$ and $b \in JS(f)$. We notice that $c := \{0, 2, 4\} \in JS$ but $c \notin JS(f)$.

Definition 17. (Jump Sequence, `JS`) *For any $n \in \mathbb{N}^*$, we call* Jump Sequence *(JS) a tuple $a \in \mathbb{Q}_+^n$ such that $a_0 = 0$ and: $\forall i \in \{0, \ldots, n-2\}, a_i < a_{i+1}$. We call n the* size *of the JS and the set of JS of size n is denoted JS_n.*

Each piece is linear on an interval with a slope and an offset.

Definition 18. ((ρ, σ)-*affine* on, `affine_on`) *Given $\rho, \sigma \in \mathbb{Q} \times \overline{\mathbb{Q}}$ and $x, y \in Q$, a function $f \in \mathcal{F}$ is called (ρ, σ)-affine on $]x; y[$ when, for all $t \in]x; y[$:*

$$f(t) = \rho(t - x) + \sigma. \tag{5.3}$$

PA are then functions that are *affine on* all intervals of a JS.

Definition 19. (JS of a Function, `JS_of`) *Let $n \in \mathbb{N}^*$, $a \in JS_n$ and $f \in \mathcal{F}$. We say that a is a JS of f, denoted $a \in JS(f)$, when for all $i < n - 1$, f is* affine on *$]a_i; a_{i+1}[$.*

So, according to the previous definition, each PA function is associated to a JS but it is not unique. We illustrate this in Figure 5.5. Also notice that a function $f \in \mathcal{F}$ with $a \in JS(f)$ is a PA function at least up to the last point of a.

Definition 20. (UPP-PA Functions, `F_UPP_PA`) *The set $\mathcal{F}_{UPP\text{-}PA}$ of UPP-PA functions is the set of functions $f \in \mathcal{F}_{UPP}$ with T for initial segment and d for period, such that there exists $a \in JS(f)$ and $last(a) = T + d$.*

The functions presented in Figure 5.4 belong to $\mathcal{F}_{\text{UPP-PA}}$. The list of abscissas of discontinuities given in the caption are jump sequences of the functions.

A UPP-PA function with initial segment T and period d is PA in $[0; T + d]$ by construction, and also PA after $T + d$ by periodicity. This point is developed in the following property.

Lemma 1. (`F_UPP_PA_JS_upto_spec`) *Let $f \in \mathcal{F}_{UPP\text{-}PA}$ with $a \in JS(f)$. For any $l \in \mathbb{Q}_+$ such that $last(a) \leq l$, there exists $a' \in JS$ such that $a' \in JS(f)$ and $last(a') = l$.*

5.3.6 Stability of UPP Functions by *(min, plus)* Operators

We now want to prove stability of $\mathcal{F}_{\text{UPP}}$ by *(min, plus)* operators: addition, minimum and convolution. These operators have been presented in Section 5.3.2. We need another operator on rational numbers: a notion of least common integer multiple such that, for any $d, d' \in \mathbb{Q}$, there exists $k, k' \in \mathbb{N}$ satisfying $kd = k'd' = lcm_Q(d, d')$.

Definition 21 ($lcm_{\mathbb{Q}_+^*}$, `lcm_pos_rat`). *For all $d, d' \in \mathbb{Q}_+^*$, for all $a, a' \in \mathbb{Z}$ and $b, b' \in \mathbb{N}^*$ such that $d = \frac{a}{b}$ and $d' = \frac{a'}{b'}$, we define*

$$lcm_{\mathbb{Q}_+^*}(d, d') := \frac{\mathrm{lcm}\left(a\frac{\mathrm{lcm}(b,b')}{b}, a'\frac{\mathrm{lcm}(b,b')}{b'}\right)}{\mathrm{lcm}(b, b')} \tag{5.4}$$

where lcm *is the least common multiple on $\mathbb{Z}$.*

We developed an expansive theory of $lcm_{\mathbb{Q}_+^*}$ in file `ratdiv.v`, including for instance the following lemma.

Lemma 2. (`dvdq_lcml`) *For $d, d' \in \mathbb{Q}_+^*$, there is $k \in \mathbb{N}$ s.t. $lcm_{\mathbb{Q}_+^*}(d, d') = k\,d$.*

To ease notations, we want to transform this binary operator, into a set operator such as $\sum_{i=1}^{3} i = (1+2)+3$. There exists a library in Coq designed with this objective: the `bigop` theory of *Mathcomp* [BGOBP08]. To fully use this library, at the time of writing, we needed to prove that $lcm_{\mathbb{Q}_+^*}$ satisfies the monoid laws. In other words, we needed to prove that $lcm_{\mathbb{Q}_+^*}$ is associative and has a neutral element. However, $lcm_{\mathbb{Q}_+^*}$ does not have a neutral element. The lcm on $\mathbb{N}^*$ has a neutral element 1. It is not the case for $lcm_{\mathbb{Q}_+^*}$: for instance $lcm_{\mathbb{Q}_+^*}\left(1, \frac{2}{3}\right) = 2$. To get out of it, a common trick extends the definition of the binary law on option types. The `option` type is used to extend the type of `{posnum rat}` with a `None` element. Then, this element is the neutral element for this optional definition of $lcm_{\mathbb{Q}_+^*}$. Since then, we extended MathComp's bigops with a theory of (idempotent) semi-groups, making the trick now mostly useless. This was made possible thanks to the switch of MathComp to Hierarchy-builder, see Section 6.1.2 for more details.

The following lemmas prove stability of $\mathcal{F}_{\text{UPP}}$ by addition and minimum.

Lemma 3. (`F_UPP_add`) *Given $f, f' \in \mathcal{F}_{UPP}$ with initial segments $T, T' \in \mathbb{Q}_+$, periods $d, d' \in \mathbb{Q}_+^*$ and increments $c, c' \in \mathbb{Q}$ respectively, the sum $f + f'$ is a UPP function with an initial segment* $\max(T, T')$*, a period $lcm_{\mathbb{Q}_+^*}(d, d')$ and an increment $lcm_{\mathbb{Q}_+^*}(d, d')\left(\frac{c}{d} + \frac{c'}{d'}\right)$.*

Lemma 4. (`F_UPP_min`) *Given $f, f' \in \mathcal{F}_{UPP}$ with initial segments $T, T' \in \mathbb{Q}_+$, periods $d, d' \in \mathbb{Q}_+^*$ and increments $c, c' \in \mathbb{Q}$ respectively, and assuming there exists $M, m \in \mathbb{Q}$ such that:*

$$\begin{aligned}
&\tfrac{c}{d} = \tfrac{c'}{d'} \\
&\vee \left(\tfrac{c}{d} < \tfrac{c'}{d'} \wedge \left(\forall t \in (T, T+d], f(t) \le M + \tfrac{c}{d} t\right) \wedge \left(\forall t \in (T, T+d], m + \tfrac{c'}{d'} t \le f'(t)\right)\right) \\
&\vee \left(\tfrac{c}{d} > \tfrac{c'}{d'} \wedge \left(\forall t \in (T, T+d], f'(t) \le M + \tfrac{c'}{d'} t\right) \wedge \left(\forall t \in (T, T+d], m + \tfrac{c}{d} t \le f(t)\right)\right)
\end{aligned} \tag{5.5}$$

the function $\min(f, f')$ *is UPP with an initial segment $\tilde{T}$, a period $\tilde{d}$ and an increment $\tilde{c}$ with*

$$\tilde{T} := \begin{cases} \max(T, T') & \textit{when } \frac{c}{d} = \frac{c'}{d'} \\ \max\left(\max(T, T'), \frac{M-m}{\frac{c'}{d'} - \frac{c}{d}}\right) & \textit{when } \frac{c}{d} < \frac{c'}{d'} \\ \max\left(\max(T, T'), \frac{M-m}{\frac{c}{d} - \frac{c'}{d'}}\right) & \textit{when } \frac{c}{d} > \frac{c'}{d'} \end{cases}$$

$$\tilde{d} := \begin{cases} lcm_{\mathbb{Q}_+^*}(d, d') & \textit{when } \frac{c}{d} = \frac{c'}{d'} \\ d & \textit{when } \frac{c}{d} < \frac{c'}{d'} \\ d' & \textit{when } \frac{c}{d} > \frac{c'}{d'} \end{cases}$$

$$\tilde{c} := \begin{cases} lcm_{\mathbb{Q}_+^*}(d, d')\frac{c}{d} & \textit{when } \frac{c}{d} = \frac{c'}{d'} \\ c & \textit{when } \frac{c}{d} < \frac{c'}{d'} \\ c' & \textit{when } \frac{c}{d} > \frac{c'}{d'}. \end{cases}$$

Remark 12. *In the case of PA functions, it is easy to find values for M and m_i satisfying* (5.5) *by computing bounds like $\sup_{t \in (T, T+d]} \left\{f(t) - \frac{c}{d} t\right\}$ and $\inf_{t \in (T, T+d)} \left\{f'(t) - \frac{c'}{d'} t\right\}$. That's what our implementation does.*

Regarding the convolution, considering $f, f' \in \mathcal{F}_{\text{UPP}}$, we first decompose f and f' into their initial and periodic part f_1, f_2 and f'_1, f'_2 such that $f = \min(f_1, f_2)$ and $f' \min(f'_1, f'_2)$ respectively.[4] We then use the algebraic properties of the *(min, plus)* semi-ring to derive $f * f' = \min(f_1 * f'_1, f_1 * f'_2, f_2 * f'_1, f_2 * f'_1)$ (c.f., `F_UPP_conv_aux`), and prove that each of the four convolutions is UPP (c.f., `F_UPP_conv_f1_f1'` and the two following lemmas).

5.3.7 Stability of UPP-PA Functions by *(min, plus)* Operators

We are now focusing on stability of $\mathcal{F}_{\text{UPP-PA}}$ by *(min, plus)* operators. Let us first define the union of two jump sequences.

Definition 22. (Union of two JS, `JS_merge`) *For any $n, m \in \mathbb{N}^*, a \in JS_n, b \in JS_m$, the tuple of size $\#\left(\{a_i \mid 0 \le i < n\} \cup \{b_j \mid 0 \le j < m\}\right)$ containing the elements of $\{a_i \mid 0 \le i < n\} \cup \{b_j \mid 0 \le j < m\}$ sorted by increasing order, is called* union *of the jump sequences a and b. This union is denoted $a \cup b$.*

If jump sequences are implemented by lists, the union can be implemented similarly to the merge part of a merge sort.

The following Lemma states the stability of $\mathcal{F}_{\text{UPP-PA}}$ by addition.

Lemma 5. (`F_UPP_PA_add`) *Given two functions $f, f' \in \mathcal{F}_{UPP\text{-}PA}$, their sum $f + f'$ is also UPP-PA.*

[4]For instance, f_1 is equal to f on its initial segment $[0, T]$ and to $+\infty$ beyond.

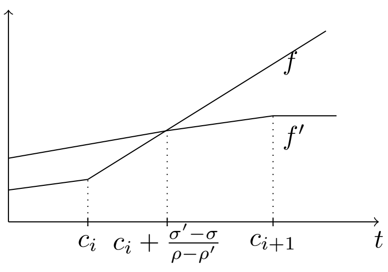


Figure 5.6: Example of point added by the min operator in a JS. f and f' are respectively $(\rho,\ \sigma)$-*affine* and $(\rho',\ \sigma')$-*affine* on $]c_i; c_{i+1}[$ with different slopes ρ and ρ'. Since we have $c_i + \frac{\sigma'-\sigma}{\rho-\rho'} \in]c_i; c_{i+1}[$, this point must be added to the jump sequence.

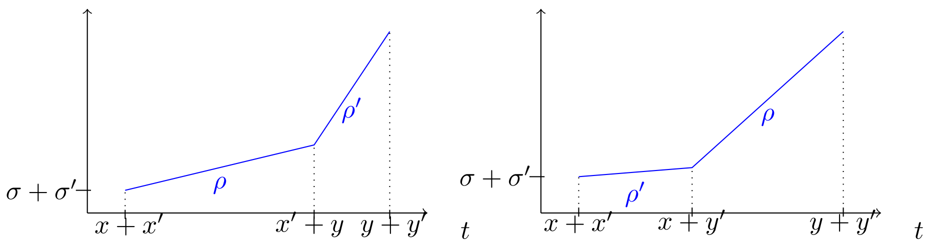


Figure 5.7: Convolution of two segments. Let f and f' be two functions that are respectively $(\rho,\ \sigma)$-*affine* on $[x; y[$ and $(\rho',\ \sigma')$-*affine* on $[x'; y'[$ and $+\infty$ elsewhere. We plot the two cases of $f * f'$ on $[x + x', y + y'[$: left is for $\rho < \rho'$ and right is $\rho' < \rho$.

The sum can be computed by merging the JS, then doing piecewise addition for each segment of the resulting JS.

$\mathcal{F}_{\text{UPP-PA}}$ is also stable by minimum under mild conditions.

Lemma 6. (`F_UPP_PA_min`) *Given two functions $f, f' \in \mathcal{F}_{\text{UPP-PA}}$, if the hypotheses of* (5.5) *their minimum* $\min(f, f')$ *is also UPP-PA.*

The computation proceed similarly to the one for addition with the following caveat. Whereas the jump sequence of a sum is the union of the jump sequences, the minimum can introduce new points as shown in Figure 5.6. The following definition gives such a jump sequence.

We are now interested in the convolution of two UPP-PA functions. Like in [BT08], we rely on the property that: $\forall f, g, h \in \mathcal{F}, \min(f, g) * h = \min(f * h, g * h)$. Then, any UPP-PA function can be decomposed as the minimum of elementary affine functions whose convolution is easy to compute. The convolution of two such functions can be computed by case disjunction in the same way as in Figure 5.7. Although handling possible discontinuities on the border of the segments lead to more than two sub-cases in practice, those can be handled similarly.

5.3.8 Finite Equality and Inequality Criteria on UPP-PA

To implement an equality test between f and f', one can use `JS_merge` to obtain a JS that suits both f and f', then perform an equality test for each segment of the resulting JS, c.f., `F_UPP_PA_eqb`.

Inequality tests can be obtained by combining the above minimum and equality test, indeed $f \le f'$ is equivalent to $\min(f, f') = f$.

5.3.9 Implementation

The implementation consists of 6.8k lines of Coq code. It offers an automatic tactic `nccoq` that performs reflexive proofs for goals of the following form

$$\begin{array}{lcl} \mathit{goal} & ::= & \mathit{expr} = \mathit{expr} \ \mid\ \mathit{exp} \le \mathit{expr} \ \mid\ \mathit{expr} \oslash \mathit{expr} \le \mathit{expr} \\ & & \mid\ \mathit{hDev}(\mathit{expr}, \mathit{expr}) \le \mathit{cst} \ \mid\ \mathit{vDev}(\mathit{expr}, \mathit{expr}) \le \mathit{cst} \\ \mathit{expr} & ::= & \mathit{fupp} \ \mid\ \mathit{expr} + \mathit{expr} \ \mid\ \min(\mathit{expr}, \mathit{expr}) \ \mid\ \mathit{expr} * \mathit{expr} \\ & & \mid\ -\mathit{expr} \ \mid\ \mathit{expr} - \mathit{expr} \ \mid\ \mathit{non\_decr\_closure}\ \mathit{expr} \end{array}$$

where $\mathit{cst} \in \mathbb{Q}$ and fupp are constant functions in $\mathcal{F}_{\text{UPP-PA}}$ given by their parameters and list of affine segments.

The implementation relies on the rational numbers defined in the MathComp library [MT18] and the real numbers from MathComp Analysis [ACK$^+$20]. This enables, among other things, the use of the big operators from MathComp [BGOBP08] as well as a nice formalization of algebra, that we extended with semi-rings and complete dioids. Semi-rings are now part of MathComp proper since its version 2 released in May 2023, see Section 6.1.2 for more details.

To obtain executable Coq programs, some adjustments were required, such as making the ρ and σ of Definition 19 explicit in the jump sequences. The final executable version uses the refinement of MathComp's rational numbers by the one in the bignums library [GT06] provided by the CoqEAL library [CDM13] and developed as part of Section 3.4.2.

Here is an example proof on the sum of the two functions f and g from Figure 5.4. We first declare f and g: (`mk_sequpp` is a mapping function)

```
Let f := F_of_sequpp (mk_sequpp 4 (* T *) 4 (* d *) 3 (* c *) [:: (0, (0, (2, 0)));
  (1, (2, (0, 2))); (2, ( 2,  (0, 3)));   (4, ( 3,  (0, 5))); (6, ( 5,  (0, 6)))]).
Let g := F_of_sequpp (mk_sequpp 4 4 (4/11) [:: (0, (0, (1/3, 0)));
                                             (3, (1, (1/11, 1)))]).
```

Then a function h that we want to prove equal to $f + g$ (this function could be obtained from an external oracle):

```
Let h := F_of_sequpp (mk_sequpp 4 4 (37/11) [:: (0, ( 0, (7/3, 0)));
  (1, (7/3, (1/3, 7/3))); (2, (8/3, (1/3, 11/3))); (3, (4, (1/11, 4)));
  (4, (45/11, (1/11, 67/11))); (6, (69/11, (1/11, 80/11)))]).
```

We can then use our new tactic `nccoq` to automatically prove the equality:

```
Goal f + g = h. Proof. nccoq. Qed.
```

This tactic performs a proof by reflection: it reduces the goal-to-prove down to a computation, which is then performed by Coq and whose success concludes the proof. The reduction is done with the help of the machinery provided by the CoqEAL library [CDM13].

5.3.10 Conclusion

Confidence in latency bounds computed by network calculus tools [BMF11, SZ06] relies, among other parts, on the correctness of the evaluation of algebraic expressions on *(min, plus)* operators [MPC, BCG$^+$09]. Instead of developing another toolbox, we developed, formalized and proved equality criteria that can be checked in finite time for each algebraic operation involved in actual computation of network calculus bounds.

The expected usage of this library is to delegate the evaluation of arbitrary algebraic expressions to an external tool [MPC], that can do efficient computations, before checking the final result with our Coq contribution. This external tool would then act as an untrusted oracle. We successfully experimented this approach on case studies representative of actual aircraft embedded networks with the PEGASE tool [BMF11], from the RTaW company, as untrusted oracle [BRD22]. Our Coq tactic confirmed the correctness of all results provided by PEGASE.

Chapter 6

Technical Activities

Part of my work is more technical than scientific and doesn't lead to any publication. This includes contributions to the development and maintenance of the software tools used in my research activities, as well as some internal contracts at ONERA. This chapter briefly gives some idea of this work with a short sum-up of my contributions to the Coq ecosystem in Section 6.1 and an example of contract in Section 6.2.

6.1 Contributing to the Coq and MathComp Ecosystems

During my PhD and the following years, I mostly developed research prototypes alone: a prototype static analyzer as described in Chapter 2 or the OCaml OSDP library (c.f., Section 3.3.4). I was almost the only one to run those codes, let alone read / modify them. This changed when I started collaborating to Coq and it proves that having others review and criticize my code was a wonderful way to learn a lot and improve it, for the benefit of all developers and users it is now shared with. This kind of contributions is usually not a matter for publications but I still consider that maintaining and improving our research tools is a fundamental contribution to the future research that will be performed with them. Thus, I dedicate a few pages below to describe my main contributions to the Coq and MathComp ecosystems in the last few years.

6.1.1 Contributing to Coq

I started contributing to Coq in 2019 with primitive floats, as seen in Chapter 4. I then became a regular maintainer in 2021 and was invited to join the core team in 2023. I served as release manager of Coq 8.20 during summer 2024 (with the kind help of Guillaume Melquiond). It may be worth noting that I arrived soon after Théo Zimmermann moved the development to Github.[1] This may have facilitated my contributions, whereas I'm not part of one of the historic research teams that made Coq up to that time.

My first contribution, primitive floats[2] actually led me to contribute parsing and printing of decimal values[3] like 3.14 (only integers were parsable before that and it was common to write things like 314/100). This ended up requiring more work than I initially anticipated but thanks to the kind support of many Coq developers, this ended up in a good enough shape to be merged. Still, this required a few more refinements[4]. I also later added hexadecimal support.`https://github.com/coq/coq/pull/11948`

Similarly, primitive floats led me to contribute to the virtual machine (behind the `vm_compute` tactic). For instance[5] to retrieve more recent (well 16 years) changes from the OCaml VM or some fixes of potential inconsistencies[6]. This is also how I got my first proof of `False`[7].

Another contribution triggered by primitive floats was to ease the use of `native_compute`. Indeed, to perform native computation, the tactic requires all dependencies of the computation to be compiled by the OCaml compiler. This can be done on the fly by the tactic but the overhead rapidly becomes unbearable. Another solution was to precompile every dependency with the OCaml compiler, at the same time their Coq code is compiled. At that time, this was done by default for Coq standard library but there was no documented solution to perform this for other libraries and the details varied from build system to build system (`coq_makefile`, dune, handmade makefiles,...). This made the use of `native_compute` utterly impractical for any development with a few dependencies. Thus, in collaboration with Érik `Martin-Dorel`,

[1]`https://github.com/coq/coq`
[2]`https://github.com/coq/coq/pull/9867`
[3]`https://github.com/coq/coq/pull/8764`
[4]for instance `https://github.com/coq/coq/pull/11848`
[5]`https://github.com/coq/coq/pull/9905`
[6]`https://github.com/coq/coq/pull/9925`
[7]`https://github.com/coq/coq/issues/10031`

we opened a Coq Enhancement Proposal[8] (CEP) and after some discussions to converge on a satisfying solution, we implemented and documented it.[9]. As a result, `native_compute` is easier to use since Coq 8.13, as it is now enough to install the new `coq-native` OPAM package to get all required precompilations done. The dune build system took a few more years to properly adapt but things are now properly working, even with it.

More generally, the experience of primitive floats came handy to help develop / review / merge further contributions from other developers such as signed primitive integers[10] or primitive strings[11].

Since the decimal numbers contribution, I also got some general interest in the extensible parsing and notation mechanisms of Coq. This materialized by contributions to extend the notation mechanism for numbers to noninductive types[12] but it all started with some OCaml GADT fiddling in the parse code[13]. Other examples of contributions to the notation mechanism come with enabling multiple scopes in `Arguments` commands[14] or later work to somewhat "improve" notation modularity.[15]

After contributing to MathComp, I also got interested in the coercion mechanism. I removed the uniform inheritance condition[16] (that was an old restriction enabling to instantiate coercions purely syntactically whereas now the pretyping mechanism can do it based on typing) following advice from Enrico TASSI. I then took part in finalising the implementation of reverse coercions[17] and added a coercion hook that can be used from the elpi metalanguage[18] to implement elaborate coercions that are not possible as basic Coq coercions.

I sometimes contribute to the reference manual although I don't have much time to devote to it.[19]

Compared to other proof assistants, Coq still has some very strong points, but its standard library is in a pretty poor state. In fact it has been in a kind of zombie state for probably more than a decade. It's somewhat alive as it is used by almost every Coq development around.[20] But at the same time it's essentially dead as contributions are not really encouraged, and when they happen nonetheless, we usually do a pretty poor job at incorporating them, if only by lack of interested reviewers. I personally feel that contributing to the standard library is currently complicated, even for Coq developers. For instance, it is widely known that the `Vector` dependent type of the standard library is a nasty trap. Nonetheless, a simple proposition to add a warning about this (relaying what everyone was repeating over and over on the Zulip forum for users[21]) took endless discussions and months to get in[22] (although I must admit it was an opportunity to ad a deprecation mechanism for entire files). Recognizing that developers of Coq itself and the standard library are mostly different teams, I opened a CEP[23] to discuss the possibility to give the standard library its own repository, considering it's likely a necessary step (although not sufficient) to its revival. After the discussion, there is an agreement on the poor situation of the library in terms of welcoming contributions, as well as the fact that some of its components are subpart / no longer state of the art (there is however no agreement on the extent of this latter issue, some of us think that it's pretty wide, starting with the definition of $\leq$ on natural numbers for instance). We thus agreed to give the library its own repository, distinct from the Coq repository that will only keep Coq itself, basic tools (like `coqwc coqchk`,...), the prelude (code loaded by default when starting Coq), basic tactics and plugins and small parts of the standard library (typically what's needed to specify primitive types and operators implemented in the kernel). I'm currently implementing this. This was a good opportunity to clarify the structure of the stdlib by identifying, documenting and checking (in CI)[24] clear subcomponents and their dependencies[25] (indeed, currently virtually every file in the library can depend on any other file, as long as no cyclic dependency appears, thus over time there remains only a rather loose link between the directory structure and the compilation dependencies, and accidentally creating impossible-to-untangle cyclic dependencies becomes a more and more serious threat). As no agreement was reached on the packaging, the `coq` OPAM metapackage will keep depending on the `coq-stdlib` package.[26]

[8]`https://github.com/coq/ceps/pull/48`

[9]`https://github.com/coq/coq/pull/13352` and `https://github.com/coq/coq/pull/13684`

[10]`https://github.com/coq/coq/pull/13559`

[11]`https://github.com/coq/coq/pull/18973`

[12]`https://github.com/coq/coq/pull/12218` and `https://github.com/coq/coq/pull/14525`

[13]`https://github.com/coq/coq/pull/9815`

[14]`https://github.com/coq/coq/pull/16472`

[15]`https://github.com/coq/coq/pull/19049` and `https://github.com/coq/coq/pull/19149`

[16]`https://github.com/coq/coq/pull/15789`

[17]`https://github.com/coq/coq/pull/15693`

[18]`https://github.com/coq/coq/pull/17794` and `https://github.com/LPCIC/coq-elpi/pull/484`

[19]For instance `https://github.com/coq/coq/pull/15836` or `https://github.com/coq/coq/pull/17532`

[20]I can only think of HoTT and Unimath that don't use it, and MathComp that almost doesn't use it.

[21]`https://coq.zulipchat.com`

[22]`https://github.com/coq/coq/pull/18032`

[23]`https://github.com/coq/ceps/pull/83`

[24]The combination of the Nix package manager and the Cachix cache is very convenient to check nontrivial dependencies without recompiling things over and over.

[25]I would personally have gone as far as publishing those subcomponentss as OPAM packages, enabling users to clarify which part of the – huge – stdlib, their development actually depends on. However, we didn't met an agreement on this, so the subcomponents will remain an internal thing.

[26]I personally think it should be the reverse, after all when you install `g++` for instance, you just want a C++ compiler, not

I contributed many other small pull requests to Coq, mostly small fixes and maintenance.[27] For instance, I launched a long term effort to clarify the meaning of `:>` that was initially used to declare coercion on record fields but was accidentally reused for type-classes in the records that were type-class declarations. That effort started in 2022[28] but to avoid breaking users code and enabling everyone enough time to adapt with nice deprecation messages, this will only be completed with Coq 8.21 in 2025. Only then will `:>` uncoditionally mean coercion and `::` type class subinstances. Finally, as a maintainer I merged close to 200 pull requests from other contributors.[29]

6.1.2 Contributing to MathComp

I was introduced to the MathComp library by Érik Martin-Dorel during a postdoc we spent in the same office were I was delighted to use its nice matrices and big Σ notations [Rou16]. Later, after both getting positions in the same city, we developed together the ValidSDP library (c.f., Section 3.4), again using MathComp. The library proved once more very useful during Lucien's thesis as explained in Section 5.3.9. More specifically, we used in it in our NC-Coq library, for which we had to develop new algebraic structures for complete dioids (semirings, dioids (idempotent semi rings), complete lattices and complete dioids). To do this, we first took inspiration from MathComp's `ssralg.v` file that was implementing all basic algebraic structures. This was effective but proved quite painful and error prone. At the same time, the Hierarchy-Builder (HB) tool [CST20] was under development to ease the implementation of hierarchies of algebraic structures, and enable seamless modifications of existing hierarchies. We thus used it to reimplement our structures, which proved much easier. I then took an important part in the port of MathComp to HB,[30] starting in spring 2021 with the porting sprint [AAB$^+$21] and leading to the release of MathComp 2.0 in May 2023, as a co-release-manager with Reynald Affeldt. The port of MathComp to HB was an opportunity to add a structure of semi-rings, as well as semigroups (monoids not requiring a neutral element, useful for the bigop mechanism with operators such as min or max).[31]

Besides the HB port, my first contribution to mathcomp consisted in backporting various lemmas about list ordering from our NC-Coq library[32]. The lemmas statements and proofs did evolve quite a bit between my initial proposal and the version that eventually got merged. I indeed learned a lot from reviews from MathComp experts. More generally, I'd encourage anyone developing any serious Coq code to take time to identify parts in the code that could be of more general interest and contribute them to the library our code depends on. This may sounds like just an altruistic move and a loss of time, but in the longer term, it can save quite a lot of effort in future developments and maintenance. Indeed, this is often a great opportunity to benefit reviews from more expert users that not only improves our initial lemmas and proofs but can teach us a lot, thus leading to a win-win situation: the upstream library got enriched with useful lemmas and we reduce the technical debt of the initial library. A good practice is then to maintain in each development a `upstream_extra.v` file for each upstream library, containing lemma candidates for move to the `upstream` library, and regularly backport those lemmas, and eventually cleanup the file once we can require the release of `upstream` offering the new lemmas.

I'm now a regular maintainer of the MathComp library with almost 150 pull requests at the time of writing this.[33] Among other contributions, one can find lemmas about (monovariate) degree-two polynomials[34] of nicer notations for numeric constants for arbitrary rings[35] (using the notation mechanism discussed in above Section 6.1.1) but the core of those pull requests are maintenance stuff and cleanups, such as for instance, cleanup of phantom types after reverse coercions were introduced in Coq[36]. Finally, as a maintainer I reviewed more than 100 pull requests from other contributors.[37] One of the striking qualities of the MathComp library is its ease of maintenance, while offering a large set of results.

The contribution to MathComp and its port to HB led me to contribute a bit to HB itself, for instance by adding the `HB.howto` command to help users discover ways to instantiate a given structure on a given type[38] or through performance improvements.[39]

the compiler and the entire `boost` library.

[27]At time of writing, 150 PRs on `https://github.com/coq/coq/pulls?q=is%3Apr+author%3Aproux01`

[28]`https://github.com/coq/coq/pull/16230`

[29]`https://github.com/coq/coq/pulls?q=assignee%3Aproux01`

[30]`https://github.com/math-comp/math-comp/pull/733`

[31]`https://github.com/math-comp/math-comp/pull/910`

[32]`https://github.com/math-comp/math-comp/pull/738`

[33]`https://github.com/math-comp/math-comp/pulls?q=is%3Apr+author%3Aproux01`

[34]`https://github.com/math-comp/math-comp/pull/1002`

[35]`https://github.com/math-comp/math-comp/pull/841`

[36]`https://github.com/math-comp/math-comp/pull/1046`

[37]`https://github.com/math-comp/math-comp/pulls?q=reviewed-by%3Aproux01`

[38]`https://github.com/math-comp/hierarchy-builder/pull/305`

[39]For instance `https://github.com/math-comp/hierarchy-builder/pull/380` and `https://github.com/math-comp/hierarchy-builder/pull/391`

6.1.3 Contributing to MathComp Analysis

Developing NC-Coq (c.f., Section 5) required a formalization of extended reals $\overline{\mathbb{R}}$. At that time, MathComp Analysis was in an initial development stage and I was advised by one of its authors to not try to really use it. Thus, we settled to the Coquelicot library [BLM15] which was already pretty stable. In Coquelicot, the addition on $\overline{\mathbb{R}}$ was defined with $+\infty + -\infty = 0$. This offers symmetry around 0, hence some nice properties for the opposite $-$. However, this comes with a major drawback: this addition is not even associative, removing all hopes to do even the smallest bit of algebra on $\overline{\mathbb{R}}$. Coquelicot also came with a coercion from $\overline{\mathbb{R}}$ to $\mathbb{R}$ (sending infinities to 0), that was obviously not injective. It can be convenient but I strongly disliked this since it made reasonable-looking statements appear wrong once the elaborator silently introduced the coercion to make it type-check (this was even worse in the hands of newcomers to Coq like Lucien at the beginning of his PhD). Discussions with one of the library authors did not make me feel like they would welcome modifications on those points (to be fair, it was before the use of github/gitlab became widespread, greatly facilitating discussions around this kind of modifications) and this was a strong incentive to switch to MathComp Analysis when it became a strong contender a few years later.

Contrary to Coquelicot, Analysis defines $+\infty + -\infty$ as $-\infty$, making the addition on $\overline{\mathbb{R}}$ associative. In fact, this makes $(\overline{\mathbb{R}}, \max, +)$ a semi-ring. However, as seen in Section 5.3, we rather needed the dual $(\overline{\mathbb{R}}, \min, +)$ semi-ring where $+\infty + -\infty = +\infty$. Thus, adding this dual addition was my first contribution to MathComp Analysis.[40] Later, as a relatively heavy user of extended reals, I contributed multiple other improvements to their theory.[41]

Analysis is all about classical analysis, which requires some tools to work with classical logic (whereas MathComp proper is made on top of Coq constructive logic without adding any axiom). Recognizing that those tools could be of wider interest than "just" analysis, I split the `mathcomp-analysis` package into two: `mathcomp-classical` and `mathcomp-analysis`, enabling use of the classical logic features without requiring the whole analysis library.[42]

Another tool we came to appreciate a lot when working on UPP functions (c.f., Section 5.3) was the automatic proofs of positivity / nonnegativity provided by Analysis. I contributed a number of improvements / consolidation of this tool[43] (file `signed.v` of Analysis[44]). Later I similarly added a tool to infer interval bounds rather than just sign.[45] The two abstractions should be merged, but I haven't found time to do it yet.[46]

A more recent part of my contributions was related to probability, adding a notion of covariance[47] and a proof of the Cantelli inequality.[48] This was part of a work by Filip Markovic [MRB+23] I collaborated to during a five months visit to the team of Björn Brandenburg at MPI-SWS in Kaiserslautern in 2023.

As for MathComp, many of my more than 80 pull requests[49] are various small improvements or maintenance tasks. Finally, I reviewed more than 90 pull requests from other contributors.[50]

6.1.4 Contributing to MathComp Algebra Tactics

When it comes to formalize mathematical results in the Coq proof assistant, I found MathComp and the ssreflect tactic language vastly superior to the legacy Coq tactics and standard library, both in terms of efficiency and ease to develop new proofs, robustness of those proofs and ease of maintenance of the existing proofs.[51] However, there was a point where, until a few years ago, MathComp was still lagging behind: large scale reflection tactics such as `ring`, `field`, `lia` or `lra` using decision procedures to solve goals by associativity and commutativity or linear algebra respectively (`lia` stands for Linear Integer Arithmetic and `lra` for Linear Real Arithmetic).

Kazuhiko Sakaguchi developed `mczify` to provide access to the `lia` tactic within MathComp developments. He also later developed MathComp Algebra Tactics with a reimplementation of the `ring` and `field` tactics for MathComp algebraic structures [Sak22]. This even added a new feature with support of

[40]`https://github.com/math-comp/analysis/pull/374`

[41]For instance `https://github.com/math-comp/analysis/pull/466`, `https://github.com/math-comp/analysis/pull/535`, `https://github.com/math-comp/analysis/pull/546` or `https://github.com/math-comp/analysis/pull/887`

[42]`https://github.com/math-comp/analysis/pull/600`

[43]For instance `https://github.com/math-comp/analysis/pull/511` and `https://github.com/math-comp/analysis/pull/601`

[44]`https://github.com/math-comp/analysis/blob/master/theories/signed.v`

[45]`https://github.com/math-comp/analysis/pull/869`

[46]Edit: at the time of writing in summer 2024, now done in 2025.

[47]`https://github.com/math-comp/analysis/pull/918` and `https://github.com/math-comp/analysis/pull/919`

[48]`https://github.com/math-comp/analysis/pull/920`

[49]`https://github.com/math-comp/analysis/pulls?page=3&q=is%3Apr+author%3Aproux01`

[50]`https://github.com/math-comp/analysis/issues?q=reviewed-by%3Aproux01`

[51]Note that there is a very good rationale to this: MathComp was developed at least a decade later and could benefit all the lessons learned the hard way in the standard library and tactics.

morphisms, enabling for instance the `ring` tactic to prove goals like $f(x + 2y) = 2f(y) + f(x)$ when f is an additive morphism.

I contributed[52] an interface to the micromega tactic of Coq [Bes06], giving access not only to the `lra` tactic for MathComp algebraic structures, but also the `nra` and `psatz` tactics, able to handle some nonlinear goals. Similarly to the already existing `ring` and `field`, the new tactics handle morphisms.

6.1.5 Maintenance

Finally, I maintain a few other libraries, namely CoqEAL, paramcoq (a plugin used by CoqEAL) and bignums, so as to provide releases that keep compiling with recent versions of Coq. This is mostly a low effort work.

6.2 Example of Application: TTEthernet on Ariane 6 Launcher

At ONERA we perform various kind of studies for industrial actors or public agencies of the aerospace domain. The outcome of these studies can usually not be published (either to avoid betraying industrial secrets or simply because it is of no scientific interest). Here is an example of such a study, omitting most information that would not be already publicly available.

A few years ago, we were mandated to do a bibliographical study of the Time Triggered Ethernet (TTEthernet) technology used in the Ariane 6 launcher. Contrary to most of the work of Chapter 5, TTEthernet is a time triggered real-time network technology. As discussed in Section 5.1, an important part of these protocols is the clock synchronization protocol that is used to keep a bounded drift between clocks of each element in the network. A few papers [DEHS12, DS04, SDS08, SD10, SD11a, SD11b, SD13, SRSP04] analyze this synchronization protocol, particularly with model-checking techniques. We reproduced the model-checking analyses provided in [SD11a] for the SAL model checker [dMOS03].

The synchronization protocol has two main kind of actors: Synchronization Masters (SM) are clients of the synchronization algorithm, they provide their local clocks and periodically update them according to the instructions of the Compression Masters (CM) that gather all local clocks, select a kind of median value (compression function) and broadcast adjustments. CM are usually switches of the network while SM are usually end systems. It is assumed that each local clock can drift by at most max_drift during each period.

For the analyses, two fault modes are considered for the SM:

inconsistent omission : a SM in that fault mode can, at each time instant, send or not its clock to each CM (independently, i.e., the clock can be sent to only some of the CM);

byzantine : a SM in that fault mode can, at each time instant, send or not to each CM a clock that is correct or not.

For CM, a single fault mode is considered:

inconsistent omission : a CM in that fault mode can, at each instant, send or not the result of its compression function to each SM.

The results are shown in Table 6.1. It is worth noting that these bounds are obtained on a simplified model of the synchronization protocol, ignoring variable transmission delays or some mechanisms. Actual maximum reachable drifts are thus expected to be a bit larger.

The main result of the study was the discovery that the studied publications only dealt with SM-SM clock drifts, whereas Table 6.1 shows that drifts with CM can be larger under faults.

[52] `https://github.com/math-comp/algebra-tactics/pull/54`

CM	SM io	SM by	SM-SM	SM-CM	CM-CM
0	0	0	2	2	2
0	1	0	2	5/2	3
0	0	1	2	5/2	3
0	2	0	2	3	4
0	1	1	2	3	4
0	3	0	2	3	4
1	0	0	2	2	2
1	1	0	8/3	10/3	10/3
1	0	1	8/3	10/3	10/3
1	2	0	4	6	6
1	1	1	4	6	6
1	3	0	4	6	6
0	0	2	2	∞	∞
1	0	2	∞	∞	∞
0	2	1	2	∞	∞
0	1	2	2	∞	∞
0	0	3	2	∞	∞
1	2	1	∞	∞	∞

Table 6.1: For each row, the three first columns indicate the considered fault configuration. The first column indicates the number of CM in “inconsistent omission” mode. The next two columns indicate the number of SM in “inconsistent omission” and “byzantine” mode respectively. For each configuration, the SM-SM column indicates a bound b such that for all pair of SM, it is proved that, at each time instant, their clock don’t differ by more than $b \times max\_drift$. The SM-CM column indicates such a bound for all pair (SM, CM) and the CM-CM column for all CM pair.

Chapter 7

Perspectives

Sections 2.7, 3.3.6, 3.4.4, 3.4.5, 4.6, 5.3.3 and 5.3.10, have already explained, for each part of the presented work, how it compares to related works, what are its shortcomings, and how it could be improved further. I will not rehash all these details here. Instead, this short chapter will give an overview of the directions in which my research work may progress in the next few years.

7.1 Static Analysis of Control-Command Software

In the introduction in Chapter 1 we motivated the work of Chapter 2 on control-command program verification by the example of digital flight-command software now ubiquitous on large commercial aircraft. However, while Chapter 2 presented some state of the art improvements, this still lefts us very far from being able to prove functional properties of actual flight-command software.

A few years ago, we had the opportunity, during a DGAC[1] funded project, to collaborate with a local aircraft manufacturer. They provided us with a Matlab Simulink model of the longitudinal control of an actual commercial aircraft. Here longitudinal control means only pitch, in a vertical plane like on Figure 1.1, page 2, without any roll nor yaw. They introduced bugs in the model, representative of the kind of bugs actually encountered while developing the plane.[2] The best results actually achieved during this project were obtained by a colleague. Rémi Delmas was able, by using Monte-Carlo simulations, to automatically find one of the bugs [DLBC19]. This even revealed bug instances leading to more critical situations than initially known. But when it comes to proving that no such bad scenario can happen,[3] neither the control theorists involved in the project, nor us, got anywhere close to a proof. Indeed, beyond the linear controller at their core, those are pretty complex software. There is in fact not a single linear controller but one for each flight-point (mostly speed and altitude, but also mass and mass distribution), with some interpolation between them. To make things even more complicated, they usually feature a number of nonlinear mechanisms such as saturations or antiwindups.[4] Finally, those are hybrid systems since the behavior of the plane is continuous (time) while the control law is implemented as a discrete (time) program. Interestingly, this is not the worst difficulty, since in practice the discrete program is nothing else than the discretization of the continuous law designed by control theorists.

Thus, the problem remains a challenging and interesting research topic. I'm now intimately convinced that proving actual functional properties on actual flight command software would require tight collaboration not only with control theorists but also with expert engineers developing such systems. My lab ONERA[5] is quite an ideal place to foster collaborations between control theorists and computer scientists. This indeed lead to some progress and mutual understanding. However the path to an actual effective collaboration probably remains quite long.

An interesting point, when comparing usual computer science programs with this kind of control-command programs, are the mechanisms used to master the complexity. In software engineering, it is a common good practice to split large programs into smaller units with clear and well defined interfaces. This way, each smaller unit can be developed and verified somewhat independently. The development of control-command programs seems much more monolithic. One of the rare modularity practices I'm aware of is to develop units

[1]Direction Générale de l'Aviation Civile, the French civil aviation administration.

[2]Typical programming bugs, like inversion of branches of if-then-else constructs, memory not correctly reset during some mode switching,...

[3]On the fixed code, of course.

[4]A kind of saturation that, when triggered, instead of keeping the saturated value, activates an alternative law aggressively trying to resolve the saturation.

[5]A kind of french equivalent of NASA.

working on separate frequency ranges, for instance a controller handling some high frequency phenomenon, coupled with another handling lower frequency behaviors.

7.2 Improving Proof Assistant Usability

When using proof-assistants to formalize already established mathematical results, a large part of the difficulty comes from the distance between the initial mathematical statements, as they are written on paper, and the fully formal statements, understandable by the tool, with the many details intuitively added by the reader.[6] For instance, in a statement like "for all $u \in \mathbb{R}^{\mathbb{N}}$ such that $u_{k+2} = u_k + u_{k+1}$ for all k,...", the reader intuitively understands that $k \in \mathbb{N}$ and that $k+1$ and $k+2$ are addition on $\mathbb{N}$ whereas $u_k + u_{k+1}$ is an addition on $\mathbb{R}$. In proof assistants, this kind of gaps are filled by a pre-processing phase called *elaboration*. The elaborator takes a term with holes, or even some typing errors, fills the holes and fixes the typing errors to come up with a fully detail term that should typecheck. This is done through various mechanisms like unification, implicit coercions, type-class inference or canonical-structure inference[7] in Coq.

To illustrate the various pre-processing steps performed by Coq on user input, to produce a fully formal term that can be typechecked by Coq's kernel, let's consider an example.[8] Considering the statement

$$\text{“if } f \text{ is linear then } f(2 \cdot x + y) = 2 \cdot f(x) + f(y)\text{”},$$

the reader understands that this assumes some ambient vector spaces over a field[9] and that the unbound x and y variables are in fact universally quantified. So the reader actually understands the statement as something like

$$\text{“if } R \text{ is a ring, } U \text{ and } V \text{ are modules over } R \text{ and } f : U \to V \text{ is linear then, for all } x,\ y, \text{ we have } f(2 \cdot x + y) = 2 \cdot f(x) + f(y)\text{”.}$$

Note that this is still not fully precise. We didn't specify in which set x and y live, so we should write $x \in U$ but actually U is an algebraic structure, not a set, so we should rather write $x \in |U|$ to denote that x is taken in the carrier set of U. Similarly, what does 2 denote? 2 is a notation for $1 + 1$ in ring R, so we could write 2_R. We should also write $\cdot_U$, $+_U$, $\cdot_V$ and $+_V$ to specify that $\cdot$ and $+$ denote the operators of U and V respectively. We finally need to precise the "order" of operators by adding parentheses. We end up with

$$\text{“if } R \text{ is a ring, } U \text{ and } V \text{ are modules over } R \text{ and } f : |U| \to |V| \text{ is linear then, for all } x, y \in |U|, \text{ we have } f((2 \cdot_R x) +_U y) = (2 \cdot_R f(x)) +_V f(y)\text{”.}$$

This is much more precise, but also starts to look pretty unreadable, with all the details cluttering the main message. Fortunately, using the Mathematical Components library [MT18], Coq accepts an input close to the intermediate version above:

```
From mathcomp Require Import all_ssreflect all_algebra.  (* loading MathComp *)

Local Open Scope ring_scope.

Example elaboration (R : ringType) (U V : lmodType R) (f : {linear U → V}) :
  ∀x y, f (2 *: x + y) = 2 *: f x + f y.
Proof. exact: GRing.linearP. Qed.
```

The elaborator of Coq will then fill all the details to feed the kernel with something close to the third version of the statement above.

More precisely, Coq treats each input statement via the following successive phases:

1. The *lexing* phase splits the input into a linear stream of words called token. For instance, we would here get `Example`, `"elaboration"`, `(`, `"R"`, `:`, `"ringType"`, `)`,...

2. The *parsing* phase takes the flat result of lexing and structures it as a tree, called Abstract Syntax Tree (AST). The fact that `*:` has higher priority than `+` is decided by this phase that produces the subtree

[6]Of course, this is assuming we start from a reasonable mathematical statement and not just a vague English sentence. Otherwise, a first pen-and-paper formalization phase might be needed and can also come with serious hardships.

[7]Canonical structures are an ancestor of type-classes in Coq, they tend to offer better performances at the price of much restricted capabilities.

[8]Borrowed from Andrej Bauer in his talk "The Dawn of Formalized Mathematics" available at `https://math.andrej.com/2021/06/24/the-dawn-of-formalized-mathematics/` and simply translated from Lan's Mathlib to Coq's MathComp.

[9]Actually, modules over a ring are enough.

```
          +
        /   \
      *:      y
    /   \
   2      x
```

rather than

```
      *:
    /   \
   2      +
        /   \
      x       y
```

for instance. The parser of Coq is mostly an LL1 parser, with the particularity that it can be extended, typically through the `Reserved Notation` command, or by plugins.

3. The *notation interpretation* phase gives a meaning to each syntax. Different meanings can be attached to a same syntax depending on the context, via a mechanism of *scopes*. For instance, by default `_ + _` denotes the addition `Nat.add` of natural numbers but thanks to the line `Local Open Scope ring_scope` it is now mapped[10] to the generic ring addition `GRing.add` of MathComp. Similarly, `2` gets interpreted as `GRing.natmul _ (GRing.one _) (S (S O))` where `S (S O)` is the natural number 2. So, after notation interpretation, the above terms become `GRing.add _ (GRing.scale _ _ (GRing.natmul _ (GRing.one _) (S (S O))) x) y` where the placeholders `_` are for the, still unknown, algebraic structures.

4. The *elaboration* phase[11] fills the holes (`_`). For instance, knowing that `x` is given as argument to `f`, unification infers that the type of `x` is `U`. Then, the holes in `GRing.scale _ _` are filled as `GRing.scale R U` in the left-hand side of the equality and `GRing.scale R V` in the right-hand side.

5. Finally, the *typechecking* phase, performed by the kernel, takes the elaborated term and check that it correctly types.

In recent versions of Coq, phases 1., 2. and 3. are grouped as a *syntax interpretation* phase and phases 4. and 5. as an *interpretation* phase. This way, syntax interpretation is all one needs to parse a Coq source file, whereas interpretation performs the actual checking work, which can be much more costly.

The elaborator of Coq is already a fantastic tool, but there is still room for progress. For instance, the implicit coercion mechanism is able to automatically insert coercions as casts from two given types. As an example, given an integer `p : int` and a natural number `n : nat`, the term `p + n` will be elaborated as `p + int_of_nat n`. However, coercions don't work for parameterized type, for instance they can't cast matrices of natural numbers to matrices of integers, the user will have to insert an explicit cast, cluttering its statement with uninteresting details.

Another example of coercions' shortcomings comes with the cast in MathComp from natural numbers to values in a ring, that is the `GRing.natmul _ (GRing.one _) n` seen above, denoted `n%:R`. With `x` in some ring, one would like to be able to write `n * x` rather than having to write `n%:R * x` as we currently do. Getting this right is a nontrivial task. Mathlib solves the problem by having heterogeneous operators, that is instead of having a single ring multiplication `R -> R -> R`, there will be a ring multiplication `R -> R -> R`, a multiplication between nat and ring `nat -> R -> R`, between ring and nat `R -> nat -> R` and so on. This does work but requires a high number of instances on the two operands, just for applying a coercion on a single operand.

I've started prototyping other ideas with the help of the Elpi [DGCT15] extension language. Elpi is a wonderful tool for fast prototyping this kind of thing, that would otherwise require tinkering with the low level OCaml code of Coq itself. For instance, here I just added a hooks in Coq[12] and was able to easily experiment by programming arbitrary coercions in the elpi language. Once a satisfactory solution is found, if performance is not good enough, it will still be time to reimplement it in OCaml inside Coq. A direct use of the hook enables to get things like `x * n` elaborated as `x * n%:R`. Unfortunately `n * x` doesn't work as the type of the multiplication is decided by the first argument: in the first case it is thus the ring of `x` and the natural number `n` can be casted to it, whereas in the second case the multiplication is of type `nat` and `x` cannot be cast to a `nat`. A simple idea is then to try both directions for inference: left-to-right as usual and if it fails, then try right-to-left. This works but the complexity can become exponential in the number

[10]Alternatively to the `Open Scope` command, scopes can be open locally for a given term with the `term%scope_key` syntax, for instance `(f (2 *: x + y) = 2 *: f x + f y)%R`.

[11]Called `pretyping` in the source code of Coq.

[12]See `https://github.com/coq/coq/pull/17794`

of nested operators in an expression. Our current idea with Cyril `Cohen` is to use its Trocq [CCM24] proof translation tool to directly elaborate the terms with the coercions.

7.3 Interval Arithmetic for Formal Proofs of Numerical Results

Proof assistants are good at manipulating arbitrary logic formulas. Among them, Coq also offers some rather unique computation capabilities. For instance, its `vm_compute` and `native_compute` evaluation mechanisms make it amenable for relatively efficient computations inside the tool. Coq is then an appropriate tool to perform proofs by reflection, that is proofs with an intensive computation part, as seen for instance in Section 3.4. Two other families of tools are available to users in need of symbolic or numerical computations, like applied mathematicians, computer scientist or control theorists:

- Computer algebra systems (CAS), like Maple or Mathematica perform symbolic manipulations, like proof assistants, but are specialized to algebraic expressions rather than arbitrary logical formulas. They implement complex algorithms, for instance quantifier elimination algorithms like the cylindrical algebraic decomposition (CAD), with many heuristics for efficiency.

- Numerical frameworks, such as Octave, Matlab or Scilab, compute pretty much everything with floating-point arithmetic and offer a convenient access to many numerical procedures.

In both cases, these tools are very good in what they do but:

- They are not designed to easily enable combination with other forms of reasoning. Consider for instance a mathematical proof that would imply both some combinatorics and algebraic reasoning or both some graph theory and numerical arguments.

- Soundness of their results relies on soundness of their implementations. Since the algorithms they implement are often far from trivial, it is relatively easy for small mistakes to sneak into them and remain undetected for some time. In the case of numerical frameworks, the situation is even more dire since they are unsound by their very nature, performing approximate computations with rounding errors.[13]

Proof-assistants may not replace CAS or numerical frameworks anytime soon, due to the performance overhead they introduce and, maybe more importantly, the – sometimes large – extra cost of developing verified implementations, but they can bring an answer to both points above.

Indeed, we have seen interest in implementing, and verifying, even complex algorithms such as CAD in Coq [Mah07] on one side and rigorous interval methods, such as in the CoqInterval library [MDM16] on the other. While CoqInterval is able to prove very nice results on univariate expressions, we got some results for multivariate polynomials with our ValidSDP library, as seen in Section 3.4. It would be interesting to see how this could be expanded to the state of the art in interval arithmetic, for instance for linear algebra [Rum10].

7.4 Formal Verification of Real-Time Network and Systems

In Section 5.2 we formally verified theorems of network calculus, later in Section 5.3 we developed an automatic tactic to produce proofs of correctness for the computations performed in network calculus, on the min-plus dioid of functions. However, these two works remain fairly independent and we didn't establish any link between them. In particular, we verified the main bricks involved in a network calculus analysis but didn't provide any mechanized guarantee on the result of an analysis.[14]

Ideally, we would implement in Coq:

- A language to formalize networks as an assembly of the, already formalized, network calculus servers.

- A semantics for the above language enabling to express delay and backlog bounds on the described networks, that is the kind of goals we want to ultimately prove. Again this can use the, already existing, definitions of network calculus. In particular, cumulative flows, defined in Section 5.2, would be the main object here.

[13]However, it must be noted that great care is often taken about the numerical stability of the implementations, so that the accuracy of the results if often very good, making the tools usable in practice.

[14]Other than manually on a given small case study. Although the manual proof process did confirm that we possess all involved bricks, it also proved absolutely not scalable.

- A language of network-calculus proofs, whose basic elements would be applications of the network calculus theorems already proved in Section 5.2. The data manipulated by this language would be network calculus arrival curves (the main abstraction manipulated by network-calculus analyses, to abstract the previous – concrete – cumulative flows, see Section 5.2 for more details) as well as delay and backlog bounds (horizontal and vertical deviations, simply represented by rationals in $\mathbb{Q}$).
- A small interpreter or checker for programs in the above language, that would use the code developed in Section 5.3 for verifying actual computations in the min-plus dioid of functions, which are the main operations performed during network-calculus analyses on the arrival curves.
- Finally, a proof of correctness of the above interpreter would, given a network and a network-calculus proof in the above languages, provide a formal proof about the network, under the assumption that the above checker succeeds on the provided network-calculus proof. This would enable proofs by reflection to automatically check results established by network-calculus analyses. This main correctness theorem should be relatively easy to prove, by reusing the already proved main network-calculus theorems.

The above process would automatically provide fully mechanized proofs of results on the network-calculus model of a network. However, during an actual analysis, this model is usually derived by some more or less heavy processing of some modeling of the network to be analyzed. Another direction of work would then be to formalize those higher level models and the transformations needed to come up with the network-calculus level model.

More generally, all this offers a nice framework to formalize and verify new analyses techniques of new network protocols or algorithms. Proofs in this field are not necessarily very hard but can easily get tricky, as seen in Section 5.2 and mechanized proofs are really valuable here to improve the confidence in the new (and existing) results and alleviate the reviewing work for new results. Finally, having a mechanized formalization makes it easier to study generalization of results, for instance by simply commenting out some hypotheses and observing the impact on the proofs compilation.

Bibliography

[AAB$^+$21] Reynald Affeldt, Xavier Allamigeon, Yves Bertot, Quentin Canu, Cyril Cohen, Pierre Roux, Kazuhiko Sakaguchi, Enrico Tassi, Laurent Théry, and Anton Trunov. Porting the mathematical components library to hierarchy builder. In *the COQ Workshop 2021*, 2021.

[Abr05] Jean-Raymond Abrial. *The B-book - assigning programs to meanings.* Cambridge University Press, 2005.

[ACK$^+$20] Reynald Affeldt, Cyril Cohen, Marie Kerjean, Assia Mahboubi, Damien Rouhling, and Kazuhiko Sakaguchi. Competing inheritance paths in dependent type theory: A case study in functional analysis. In Nicolas Peltier and Viorica Sofronie-Stokkermans, editors, *Automated Reasoning - 10th International Joint Conference, IJCAR 2020, Paris, France, July 1-4, 2020, Proceedings, Part II*, volume 12167 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2020.

[AFP09] Fernando Alegre, Éric Féron, and Santosh Pande. Using ellipsoidal domains to analyze control systems software. *CoRR*, abs/0909.1977, 2009.

[AGG10] Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *ESOP*, pages 23–42, 2010.

[AGM15] Assalé Adjé, Pierre-Loïc Garoche, and Victor Magron. Property-based Polynomial Invariant Generation Using Sums-of-Squares Optimization. In *SAS*, pages 235–251, 2015.

[AGST10] Michaël Armand, Benjamin Grégoire, Arnaud Spiwack, and Laurent Théry. Extending Coq with imperative features and its application to SAT verification. In Matt Kaufmann and Lawrence C. Paulson, editors, *Interactive Theorem Proving, First International Conference, ITP 2010, Edinburgh, UK, July 11-14, 2010. Proceedings*, volume 6172 of *Lecture Notes in Computer Science*, pages 83–98. Springer, 2010.

[ApS15] MOSEK ApS. *The MOSEK C optimizer API manual Version 7.1 (Rev. 40)*, 2015.

[BBLC18] Anne Bouillard, Marc Boyer, and Euriell Le Corronc. *Deterministic Network Calculus: From Theory to Practical Implementation.* Wiley, 10 2018.

[BBRS16] Sophie Bernard, Yves Bertot, Laurence Rideau, and Pierre-Yves Strub. Formal proofs of transcendence for e and pi as an application of multivariate and symmetric polynomials. In Jeremy Avigad and Adam Chlipala, editors, *Proceedings of the 5th ACM SIGPLAN Conference on Certified Programs and Proofs, Saint Petersburg, FL, USA, January 20-22, 2016*, pages 76–87. ACM, 2016.

[BCC$^+$12] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernelala, Assia Mahboubi, Alain Mebsout, and Guillaume Melquiond. A Simplex-Based Extension of Fourier-Motzkin for Solving Linear Integer Arithmetic. In *IJCAR*, 2012.

[BCC$^+$14] François Bobot, Sylvain Conchon, Evelyne Contejean, Mohamed Iguernlala, Stephane Lescuyer, and Alain Mebsout. *Alt-Ergo, version 0.99.1.* CNRS, Inria, Université Paris-Sud 11, and OCamlPro, December 2014. `http://alt-ergo.lri.fr/`.

[BCG$^+$09] Anne Bouillard, Bertrand Cottenceau, Bruno Gaujal, Laurent Hardouin, Sebastien Lagrange, Mehdi Lhommeau, and Eric Thierry. COINC library: a toolbox for the network calculus. In *Proceedings of the 4th international conference on performance evaluation methodologies and tools, ValueTools*, volume 9, 2009.

[BDG11] Mathieu Boespflug, Maxime Dénès, and Benjamin Grégoire. Full reduction at full throttle. In Jean-Pierre Jouannaud and Zhong Shao, editors, *Certified Programs and Proofs - First International Conference, CPP 2011, Kenting, Taiwan, December 7-9, 2011. Proceedings*, volume 7086 of *Lecture Notes in Computer Science*, pages 362–377. Springer, 2011.

[BDG$^+$13] Gilles Barthe, François Dupressoir, Benjamin Grégoire, César Kunz, Benedikt Schmidt, and Pierre-Yves Strub. Easycrypt: A tutorial. In *Foundations of Security Analysis and Design VII - FOSAD 2012/2013 Tutorial Lectures*, 2013.

[BEEFB94] Stephen Boyd, Laurent El Ghaoui, Éric Féron, and Venkataramanan Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of *SIAM*. SIAM, Philadelphia, PA, June 1994.

[Bes06] Frédéric Besson. Fast reflexive arithmetic tactics the linear case and beyond. In *TYPES*, 2006.

[BGOBP08] Yves Bertot, Georges Gonthier, Sidi Ould Biha, and Ioana Pasca. Canonical big operators. In Otmane Ait Mohamed, César Muñoz, and Sofiène Tahar, editors, *Theorem Proving in Higher Order Logics*, pages 86–101, Berlin, Heidelberg, 2008. Springer Berlin Heidelberg.

[BJLM13] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. A formally-verified C compiler supporting floating-point arithmetic. In Alberto Nannarelli, Peter-Michael Seidel, and Ping Tak Peter Tang, editors, *21st IEEE Symposium on Computer Arithmetic*, pages 107–115, Austin, TX, USA, June 2013.

[BJLM15] Sylvie Boldo, Jacques-Henri Jourdan, Xavier Leroy, and Guillaume Melquiond. Verified compilation of floating-point computations. *Journal of Automated Reasoning*, 54(2):135–163, 2015.

[BLM15] Sylvie Boldo, Catherine Lelay, and Guillaume Melquiond. Coquelicot: A user-friendly library of real analysis for coq. *Math. Comput. Sci.*, 9(1):41–62, 2015.

[BM06] Sylvie Boldo and Cesar Munoz. A high-level formalization of floating-point number in PVS. Technical Report 20070003560, NASA, National Institute of Aerospace, Hampton, VA, USA, 2006.

[BM11] Sylvie Boldo and Guillaume Melquiond. Flocq: A Unified Library for Proving Floating-point Algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, pages 243–252, Tübingen, Germany, July 2011.

[BMDR19] Guillaume Bertholon, Érik Martin-Dorel, and Pierre Roux. Primitive floats in Coq. In John Harrison, John O'Leary, and Andrew Tolmach, editors, *10th International Conference on Interactive Theorem Proving*, volume 141 of *Leibniz International Proceedings in Informatics*, pages 7:1–7:20, Portland, OR, USA, September 2019.

[BMF11] Marc Boyer, Jörn Migge, and Marc Fumey. PEGASE, A Robust and Efficient Tool for Worst Case Network Traversal Time. In *Proc. of the SAE 2011 AeroTech Congress & Exhibition*, Toulouse, France, 2011. SAE International.

[BNF12] Marc Boyer, Nicolas Navet, and Marc Fumey. Experimental assessment of timing verification techniques for AFDX. In *6th European Congress on Embedded Real Time Software and Systems*, Toulouse, France, February 2012.

[BNOT10] Marc Boyer, Nicolas Navet, Xavier Olive, and Eric Thierry. The PEGASE Project: Precise and Scalable Temporal Analysis for Aerospace Communication Systems with Network Calculus. In Tiziana Margaria and Bernhard Steffen, editors, *Leveraging Applications of Formal Methods, Verification, and Validation*, pages 122–136, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[Bor99] Brian Borchers. Csdp, a c library for semidefinite programming. *Optimization Methods and Software*, 11(1-4):613–623, 1999.

[BR16] Marc Boyer and Pierre Roux. Embedding network calculus and event stream theory in a common model. In *21st IEEE International Conference on Emerging Technologies and Factory Automation, ETFA 2016, Berlin, Germany, September 6-9, 2016*, pages 1–8. IEEE, 2016.

[BRD22] Marc Boyer, Pierre Roux, and Hugo Daigmorte. Checking validity of the min-plus operations involved in the analysis of a real-time embedded network. In *ERTS 2022- 11th European Congress Embedded Real Time System*, Toulouse, France, June 2022.

[BRDP21] Marc Boyer, Pierre Roux, Hugo Daigmorte, and David Puechmaille. A residual service curve of rate-latency server used by sporadic flows computable in quadratic time for network calculus. In Björn B. Brandenburg, editor, *33rd Euromicro Conference on Real-Time Systems, ECRTS 2021, July 5-9, 2021, Virtual Conference*, volume 196 of *LIPIcs*, pages 14:1–14:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2021.

[BS14] Steffen Bondorf and Jens B. Schmitt. The DiscoDNC v2 – A Comprehensive Tool for Deterministic Network Calculus. In *Proc. of the International Conference on Performance Evaluation Methodologies and Tools*, ValueTools '14, pages 44–49, December 2014.

[BSC12] Olivier Bouissou, Yassamine Seladji, and Alexandre Chapoutot. Acceleration of the abstract fixpoint computation in numerical program analysis. *J. Symb. Comput.*, 47(12):1479–1511, 2012.

[BT08] Anne Bouillard and Eric Thierry. An Algorithmic Toolbox for Network Calculus. *Discrete Event Dynamic Systems: Theory and Applications*, 18, 03 2008.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization.* Cambridge university press, 2004.

[BY08] Steven J. Benson and Yinyu Ye. Algorithm 875: DSDP5 - software for semidefinite programming. *ACM Trans. Math. Softw.*, 34(3), 2008.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CCI12] Sylvain Conchon, Évelyne Contejean, and Mohamed Iguernelala. Canonized rewriting and ground AC completion modulo Shostak theories : Design and implementation. *Logical Methods in Computer Science*, 2012. Selected Papers of TACAS.

[CCM24] Cyril Cohen, Enzo Crance, and Assia Mahboubi. Trocq: Proof transfer for free, with or without univalence. In Stephanie Weirich, editor, *Programming Languages and Systems - 33rd European Symposium on Programming, ESOP 2024, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2024, Luxembourg City, Luxembourg, April 6-11, 2024, Proceedings, Part I*, volume 14576 of *Lecture Notes in Computer Science*, pages 239–268. Springer, 2024.

[CDD$^+$13] Adrien Champion, Rémi Delmas, Michael Dierkes, Pierre-Loïc Garoche, Romain Jobredeaux, and Pierre Roux. Formal methods for the analysis of critical control systems models: Combining non-linear and linear analyses. In Charles Pecheur and Michael Dierkes, editors, *Formal Methods for Industrial Critical Systems - 18th International Workshop, FMICS 2013, Madrid, Spain, September 23-24, 2013. Proceedings*, volume 8187 of *Lecture Notes in Computer Science*, pages 1–16. Springer, 2013.

[CDGR11] Adrien Champion, Rémi Delmas, Pierre-Loïc Garoche, and Pierre Roux. Towards cooperation of formal methods for the analysis of critical control systems. *SAE International Journal of Aerospace*, 4(2):850–858, November 2011.

[CDM13] Cyril Cohen, Maxime Dénès, and Anders Mörtberg. Refinements for free! In Georges Gonthier and Michael Norrish, editors, *Certified Programs and Proofs*, volume 8307 of *LNCS*, pages 147–162. Springer, 2013.

[CGG$^+$05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, pages 462–475, 2005.

[CGK$^+$12] Sylvain Conchon, Amit Goel, Sava Krstic, Alain Mebsout, and Fatiha Zaïdi. Cubicle: A Parallel SMT-Based Model Checker for Parameterized Systems. In *CAV*, 2012.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[CKK^{+}12] Pascal Cuoq, Florent Kirchner, Nikolai Kosmatov, Virgile Prevosto, Julien Signoles, and Boris Yakobowski. Frama-C - a software analysis perspective. In *SEFM*, pages 233–247, 2012.

[Coq24] The Coq development team. *The Coq proof assistant reference manual*, 2024. Version 8.19.

[CSB16] Felipe Cerqueira, Felix Stutz, and Björn B Brandenburg. PROSA: A case for readable mechanized schedulability analysis. In *2016 28th Euromicro Conference on Real-Time Systems (ECRTS)*, pages 273–284. IEEE, 2016.

[CSC12] Arlen Cox, Sriram Sankaranarayanan, and Bor-Yuh Evan Chang. A Bit Too Precise? Bounded Verification of Quantized Digital Filters. In *TACAS*, 2012.

[CST20] Cyril Cohen, Kazuhiko Sakaguchi, and Enrico Tassi. Hierarchy builder: Algebraic hierarchies made easy in coq with elpi (system description). In Zena M. Ariola, editor, *5th International Conference on Formal Structures for Computation and Deduction, FSCD 2020, June 29-July 6, 2020, Paris, France (Virtual Conference)*, volume 167 of *LIPIcs*, pages 34:1–34:21. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2020.

[DEHS12] Bruno Dutertre, Arvind Easwaran, Brendan Hall, and Wilfried Steiner. Model-based analysis of timed-triggered ethernet. In *2012 IEEE/AIAA 31st Digital Avionics Systems Conference (DASC)*, pages 9D2–1–9D2–11, Oct 2012.

[Dén13] Maxime Dénès. Towards primitive data types for Coq 63-bits integers and persistent arrays. In *5th Coq Workshop*, Rennes, France, July 2013. `https://coq.inria.fr/files/coq5_submission_2.pdf`.

[DGCT15] Cvetan Dunchev, Ferruccio Guidi, Claudio Sacerdoti Coen, and Enrico Tassi. ELPI: fast, embeddable, λprolog interpreter. In Martin Davis, Ansgar Fehnker, Annabelle McIver, and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning - 20th International Conference, LPAR-20 2015, Suva, Fiji, November 24-28, 2015, Proceedings*, volume 9450 of *Lecture Notes in Computer Science*, pages 460–468. Springer, 2015.

[DGG^{+}18] Guillaume Davy, Christophe Garion, Pierre-Loïc Garoche, Pierre Roux, and Xavier Thirioux. Preserving functional correctness of cyber-physical system controllers: From model to code. In Hiren D. Patel, Tom J. Kazmierski, and Sebastian Steinhorst, editors, *2018 Forum on Specification & Design Languages, FDL 2018, Garching, Germany, September 10-12, 2018*, pages 5–16. IEEE, 2018.

[DLBC19] Rémi Delmas, Thomas Loquen, Josep Boada-Bauxell, and Mathieu Carton. An evaluation of monte-carlo tree search for property falsification on hybrid flight control laws. In Majid Zamani and Damien Zufferey, editors, *Numerical Software Verification - 12th International Workshop, NSV@CAV 2019, New York City, NY, USA, July 13-14, 2019, Proceedings*, volume 11652 of *Lecture Notes in Computer Science*, pages 45–59. Springer, 2019.

[dMB08] Leonardo Mendonça de Moura and Nikolaj Bjørner. Z3: an efficient SMT solver. In *TACAS*, 2008.

[dMOS03] Leonardo de Moura, Sam Owre, and N. Shankar. *The SAL Language Manual.* SRI International, 2003.

[dOM12] Diego Caminha B de Oliveira and David Monniaux. Experiments on the feasibility of using a floating-point simplex in an SMT solver. In *PAAR@ IJCAR*, 2012.

[DS04] Bruno Dutertre and Maria Sorea. Modeling and verification of a fault-tolerant real-time startup protocol using calendar automata. In Yassine Lakhnech and Sergio Yovine, editors, *Formal Techniques, Modelling and Analysis of Timed and Fault-Tolerant Systems, Joint International Conferences on Formal Modelling and Analysis of Timed Systems, FORMATS 2004 and Formal Techniques in Real-Time and Fault-Tolerant Systems, FTRTFT 2004, Grenoble, France, September 22-24, 2004, Proceedings*, volume 3253 of *Lecture Notes in Computer Science*, pages 199–214. Springer, 2004.

[Fer04] Jérôme Feret. Static analysis of digital filters. In *ESOP*, number 2986 in LNCS. Springer, 2004.

[Fer05] Jérôme Feret. Numerical abstract domains for digital filters. In *International workshop on Numerical and Symbolic Abstract Domains (NSAD)*, 2005.

[FG10] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for c programs with aspic and c2fsm. *Electr. Notes Theor. Comput. Sci.*, 267(2):3–13, 2010.

[FNOR08] Germain Faure, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. SAT modulo the theory of linear arithmetic: Exact, inexact and commercial solvers. In *SAT*, 2008.

[FP13] Jean-Christophe Filliâtre and Andrei Paskevich. Why3 - Where Programs Meet Provers. In *ESOP*, 2013.

[GAC12] Sicun Gao, Jeremy Avigad, and Edmund M. Clarke. δ-complete decision procedures for satisfiability over the reals. In *IJCAR*, 2012.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *CAV*, pages 627–633, 2009.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *ESOP*, pages 237–252, 2007.

[GKC13] Sicun Gao, Soonho Kong, and Edmund M. Clarke. dreal: An SMT solver for nonlinear theories over the reals. In *CADE*, 2013.

[GL02] Benjamin Grégoire and Xavier Leroy. A compiled implementation of strong reduction. In *7th ACM SIGPLAN International Conference on Functional programming*, pages 235–246, Pittsburgh, PA, USA, October 2002.

[GMT08] Georges Gonthier, Assia Mahboubi, and Enrico Tassi. A Small Scale Reflection Extension for the Coq system. Research Report RR-6455, INRIA, 2008.

[GP11] Éric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *VMCAI*, pages 232–247, 2011.

[GR06] Denis Gopan and Thomas W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.

[GR11] Pierre-Loïc Garoche and Pierre Roux. Dessine moi un domaine abstrait fini – une recette à base de Camlp4 et de solveurs SMT. In *22e journées francophones des langages applicatifs (JFLA'11)*, Studia Informatica Universalis. Hermann, 2011.

[GS07] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *ESOP*, pages 300–315, 2007.

[GS10] Thomas Martin Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *SAS*, pages 271–286, 2010.

[GSA$^+$12] Thomas Martin Gawlitza, Helmut Seidl, Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Abstract interpretation meets convex optimization. *J. Symb. Comput.*, 47(12):1416–1446, 2012.

[GT06] Benjamin Grégoire and Laurent Théry. A purely functional library for modular arithmetic and its application to certifying large prime numbers. In Ulrich Furbach and Natarajan Shankar, editors, *3rd International Joint Conference on Automated Reasoning*, volume 4130 of *Lecture Notes in Computer Science*, pages 423–437, Seattle, WA, USA, August 2006.

[Har99] John Harrison. A machine-checked theory of floating point arithmetic. In Yves Bertot, Gilles Dowek, André Hirschowitz, Christine Paulin-Mohring, and Laurent Théry, editors, *12th International Conference in Theorem Proving in Higher Order Logics*, volume 1690 of *Lecture Notes in Computer Science*, pages 113–130, Nice, France, 1999.

[Har07] John Harrison. Verifying nonlinear real formulas via sums of squares. In *TPHOLs*, 2007.

[HC08] Wassim M. Haddad and Vijay S. Chellaboina. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach.* Princeton University Press, 2008.

[HH12] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In *SAS*, pages 198–213, 2012.

[Hig96] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms.* Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[HJL] Viktor Härter, Christian Jansson, and Marko Lange. VSDP: verified semidefinite programming. `http://www.ti3.tuhh.de/jansson/vsdp/`.

[HMWC15] Duc Hoang, Yannick Moy, Angela Wallenburg, and Roderick Chapman. SPARK 2014 and gnatprove - A competition report from builders of an industrial-strength verifying compiler. *STTT*, 2015.

[HPR97] Nicolas Halbwachs, Yann-Erick Proy, and Patrick Roumanoff. Verification of real-time systems using linear relation analysis. *Formal Methods in System Design*, 11(2):157–185, 1997.

[IEE08] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. *IEEE Standard 754-2008*, 2008.

[JCK07] Christian Jansson, Denis Chaykin, and Christian Keil. Rigorous error bounds for the optimal value in semidefinite programming. *SIAM J. Numerical Analysis*, 46(1):180–200, 2007.

[JdM12] Dejan Jovanovic and Leonardo Mendonça de Moura. Solving non-linear arithmetic. In *IJCAR*, 2012.

[JPTY08] Bengt Jonsson, Simon Perathoner, Lothar Thiele, and Wang Yi. Cyclic dependencies in modular performance analysis. In Luca de Alfaro and Jens Palsberg, editors, *Proceedings of the 8th ACM & IEEE International conference on Embedded software, EMSOFT 2008, Atlanta, GA, USA, October 19-24, 2008*, pages 179–188. ACM, 2008.

[JR18] Claude-Pierre Jeannerod and Siegfried M. Rump. On relative errors of floating-point operations: Optimal bounds and applications. *Mathematics of Computations*, 87(310):803–819, 2018.

[KBT14] Tim King, Clark W. Barrett, and Cesare Tinelli. Leveraging linear and mixed integer programming for SMT. In *FMCAD*, 2014.

[KLYZ12] Erich Kaltofen, Bin Li, Zhengfeng Yang, and Lihong Zhi. Exact certification in global polynomial optimization via sums-of-squares of rational functions with rational coefficients. *J. Symb. Comput.*, 2012.

[Las01] Jean B. Lasserre. Global optimization with polynomials and the problem of moments. *SIAM Journal on Optimization*, 11(3):796–817, 2001.

[Las09] Jean-Bernard Lasserre. *Moments, Positive Polynomials, and Their Applications. 2009.* Imperial College Press, 2009.

[LBT01] Jean-Yves Le Boudec and Patrick Thiran. *Network Calculus: A Theory of Deterministic Queuing Systems for the Internet*, volume 2050 of *Lecture Notes in Computer Science*. Springer, 2001.

[LLL82] A. K. Lenstra, H. W. Lenstra, and L. Lovász. Factoring polynomials with rational coefficients. *Mathematische Annalen*, 261(4):515–534, 1982.

[Löf09] J. Löfberg. Pre- and post-processing sum-of-squares programs in practice. *IEEE Transactions on Automatic Control*, 54(5):1007–1011, 2009.

[Lya47] Aleksandr Mikhailovich Lyapunov. Problème général de la stabilité du mouvement. *Annals of Mathematics Studies*, 17, 1947.

[Mag14] Victor Magron. NLCertify: A Tool for Formal Nonlinear Optimization. In *Mathematical Software – ICMS 2014*, volume 8592 of *LNCS*, pages 315–320. Springer, 2014.

[MAGW15] Victor Magron, Xavier Allamigeon, Stéphane Gaubert, and Benjamin Werner. Formal proofs for nonlinear optimization. *Journal of Formalized Reasoning*, 2015.

[Mah07] Assia Mahboubi. Implementing the cylindrical algebraic decomposition within the coq system. *Math. Struct. Comput. Sci.*, 17(1):99–127, 2007.

[MBdD$^+$18] Jean-Michel Muller, Nicolas Brunie, Florent de Dinechin, Claude-Pierre Jeannerod, Mioara Joldes, Vincent Lefèvre, Guillaume Melquiond, Nathalie Revol, and Serge Torres. *Handbook of Floating-Point Arithmetic.* Birkhäuser, Basel, 2 edition, 2018.

[MBFM13] Etienne Mabille, Marc Boyer, Loïc Fejoz, and Stephan Merz. Towards certifying network calculus. In *Proc. of the 4th Conference on Interactive Theorem Proving (ITP 2013)*, Rennes, France, July 2013.

[MC11] David Monniaux and Pierre Corbineau. On the generation of positivstellensatz witnesses in degenerate cases. In *Interactive Theorem Proving - Second International Conference, ITP 2011, Berg en Dal, The Netherlands, August 22-25, 2011. Proceedings*, 2011.

[MDM16] Erik Martin-Dorel and Guillaume Melquiond. Proving tight bounds on univariate expressions with elementary functions in Coq. *Journal of Automated Reasoning*, 57(3):187–217, October 2016.

[Min95] Paul Miner. Defining the IEEE-854 floating-point standard in PVS. Technical Report 19950023402, NASA, Langley Research Center, Hampton, VA, USA, 1995.

[Min01] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[Min04] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004. `http://www.di.ens.fr/~mine/publi/article-mine-esop04.pdf`.

[MMR23] Érik Martin-Dorel, Guillaume Melquiond, and Pierre Roux. Enabling floating-point arithmetic in the coq proof assistant. *J. Autom. Reason.*, 67(4):33, 2023.

[MN13] César Muñoz and Anthony Narkawicz. Formalization of Bernstein polynomials and applications to global optimization. *J. Autom. Reasoning*, 51(2):151–196, 2013.

[Mon05] David Monniaux. Compositional analysis of floating-point linear numerical filters. In *CAV*, pages 199–212, 2005.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[Mon09] David Monniaux. On using floating-point computations to help an exact linear arithmetic decision procedure. In *CAV*, 2009.

[Moo63] Ramon E. Moore. *Interval Analysis.* Prentice-Hall, Englewood Cliffs, NJ, USA, 1963.

[MPC] RealTime-at-Work online Min-Plus interpreter for Network Calculus. `https://www.realtimeatwork.com/minplus-playground`. Accessed: 2020-11-18.

[MR17] Érik Martin-Dorel and Pierre Roux. A reflexive tactic for polynomial positivity using numerical solvers and floating-point computations. In Yves Bertot and Viktor Vafeiadis, editors, *Proceedings of the 6th ACM SIGPLAN Conference on Certified Programs and Proofs, CPP 2017, Paris, France, January 16-17, 2017*, pages 90–99. ACM, 2017.

[MRB$^+$23] Filip Markovic, Pierre Roux, Sergey Bozhko, Alessandro V. Papadopoulos, and Björn B. Brandenburg. CTA: A correlation-tolerant analysis of the deadline-failure probability of dependent tasks. In *IEEE Real-Time Systems Symposium, RTSS 2023, Taipei, Taiwan, December 5-8, 2023*, pages 317–330. IEEE, 2023.

[MT18] Assia Mahboubi and Enrico Tassi. *Mathematical Components.* online, 2018.

[NM13] Anthony Narkawicz and César Muñoz. A formally verified generic branching algorithm for global optimization. In *International Conference on Verified Software: Theories, Tools, Experiments*, volume 8164 of *LNCS*, pages 326–343, 2013.

[NPSS10] Pierluigi Nuzzo, Alberto Puggelli, Sanjit A. Seshia, and Alberto L. Sangiovanni-Vincentelli. Calcs: SMT solving for non-linear convex constraints. In *FMCAD*, 2010.

[Par03] Pablo A. Parrilo. Semidefinite programming relaxations for semialgebraic problems. *Math. Program.*, 96(2):293–320, 2003.

[PGH$^+$21] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux. Verifying the mathematical library of an UAV autopilot with frama-c. In Alberto Lluch-Lafuente and Anastasia Mavridou, editors, *Formal Methods for Industrial Critical Systems - 26th International Conference, FMICS 2021, Paris, France, August 24-26, 2021, Proceedings*, volume 12863 of *Lecture Notes in Computer Science*, pages 167–173. Springer, 2021.

[PGH$^+$23] Baptiste Pollien, Christophe Garion, Gautier Hattenberger, Pierre Roux, and Xavier Thirioux. A verified UAV flight plan generator. In *11th IEEE/ACM International Conference on Formal Methods in Software Engineering, FormaliSE 2023, Melbourne, Australia, May 14-15, 2023*, pages 130–140. IEEE, 2023.

[PQR09] André Platzer, Jan-David Quesel, and Philipp Rümmer. Real world verification. In *CADE*, 2009.

[RBR19] Lucien Rakotomalala, Marc Boyer, and Pierre Roux. Formal Verification of Real-time Networks. In *JRWRTC 2019, Junior Workshop RTNS 2019*, Toulouse, France, November 2019.

[RDG10] Pierre Roux, Remi Delmas, and Pierre-Loïc Garoche. SMT-AI: an abstract interpreter as oracle for k-induction. In David Delmas and Xavier Rival, editors, *Proceedings of the Tools for Automatic Program AnalysiS, TAPAS@SAS 2010, Perpignan, France, September 17, 2010*, volume 267 of *Electronic Notes in Theoretical Computer Science*, pages 55–68. Elsevier, 2010.

[RFM05] Mardavij Roozbehani, Éric Féron, and Alexandre Megretski. Modeling, optimization and computation for software verification. In *HSCC*, pages 606–622, 2005.

[RG13a] Pierre Roux and Pierre-Loïc Garoche. A polynomial template abstract domain based on bernstein polynomials. In *Sixth International Workshop on Numerical Software Verification (NSV'13)*, 2013.

[RG13b] Pierre Roux and Pierre-Loïc Garoche. Integrating policy iterations in abstract interpreters. In *ATVA*, 2013.

[RG14] Pierre Roux and Pierre-Loïc Garoche. Computing quadratic invariants with min- and max-policy iterations: A practical comparison. In Cliff B. Jones, Pekka Pihlajasaari, and Jun Sun, editors, *FM 2014: Formal Methods - 19th International Symposium, Singapore, May 12-16, 2014. Proceedings*, volume 8442 of *Lecture Notes in Computer Science*, pages 563–578. Springer, 2014.

[RG15] Pierre Roux and Pierre-Loïc Garoche. Practical policy iterations - A practical use of policy iterations for static analysis: the quadratic case. *Formal Methods Syst. Des.*, 46(2):163–196, 2015.

[RIC18] Pierre Roux, Mohamed Iguernlala, and Sylvain Conchon. A non-linear arithmetic procedure for control-command software verification. In Dirk Beyer and Marieke Huisman, editors, *Tools and Algorithms for the Construction and Analysis of Systems - 24th International Conference, TACAS 2018, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2018, Thessaloniki, Greece, April 14-20, 2018, Proceedings, Part II*, volume 10806 of *Lecture Notes in Computer Science*, pages 132–151. Springer, 2018.

[RJG15] Pierre Roux, Romain Jobredeaux, and Pierre-Loïc Garoche. Closed loop analysis of control command software. In Antoine Girard and Sriram Sankaranarayanan, editors, *Proceedings of the 18th International Conference on Hybrid Systems: Computation and Control, HSCC'15, Seattle, WA, USA, April 14-16, 2015*, pages 108–117. ACM, 2015.

[RJGF12] Pierre Roux, Romain Jobredeaux, Pierre-Loïc Garoche, and Éric Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *HSCC*, pages 105–114, 2012.

[Rou13] Pierre Roux. *Static Analysis of Control Command Systems: Synthetizing Non Linear Invariants.* PhD thesis, Institut Supérieur de l'Aéronautique et de l'Espace, 2013.

[Rou14] Pierre Roux. Innocuous double rounding of basic arithmetic operations. *J. Formaliz. Reason.*, 7(1):131–142, 2014.

[Rou16] Pierre Roux. Formal proofs of rounding error bounds - with application to an automatic positive definiteness check. *J. Autom. Reason.*, 57(2):135–156, 2016.

[RQB22] Pierre Roux, Sophie Quinton, and Marc Boyer. A formal link between response time analysis and network calculus. In Martina Maggio, editor, *34th Euromicro Conference on Real-Time Systems, ECRTS 2022, July 5-8, 2022, Modena, Italy*, volume 231 of *LIPIcs*, pages 5:1–5:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[RRB21] Lucien Rakotomalala, Pierre Roux, and Marc Boyer. Verifying min-plus computations with coq. In Aaron Dutle, Mariano M. Moscato, Laura Titolo, César A. Muñoz, and Ivan Perez, editors, *NASA Formal Methods - 13th International Symposium, NFM 2021, Virtual Event, May 24-28, 2021, Proceedings*, volume 12673 of *Lecture Notes in Computer Science*, pages 287–303. Springer, 2021.

[RS10] Pierre Roux and Radu Siminiceanu. Model checking with edge-valued decision diagrams. In César A. Muñoz, editor, *Second NASA Formal Methods Symposium - NFM 2010, Washington D.C., USA, April 13-15, 2010. Proceedings*, volume NASA/CP-2010-216215 of *NASA Conference Proceedings*, pages 222–226, 2010.

[Rum06] Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46:433–452, 2006.

[Rum10] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, May 2010.

[RVS16] Pierre Roux, Yuen-Lam Voronin, and Sriram Sankaranarayanan. Validating numerical semidefinite programming solvers for polynomial invariants. In Xavier Rival, editor, *Static Analysis - 23rd International Symposium, SAS 2016, Edinburgh, UK, September 8-10, 2016, Proceedings*, volume 9837 of *LNCS*, pages 424–446. Springer, 2016.

[RVS18] Pierre Roux, Yuen-Lam Voronin, and Sriram Sankaranarayanan. Validating numerical semidefinite programming solvers for polynomial invariants. *Formal Methods Syst. Des.*, 53(2):286–312, 2018.

[Sak22] Kazuhiko Sakaguchi. Reflexive tactics for algebra, revisited. In June Andronick and Leonardo de Moura, editors, *13th International Conference on Interactive Theorem Proving, ITP 2022, August 7-10, 2022, Haifa, Israel*, volume 237 of *LIPIcs*, pages 29:1–29:22. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[SB13] Yassamine Seladji and Olivier Bouissou. Numerical abstract domain using support functions. In *NFM*, 2013.

[SCP99] Hanrijanto Sariowan, Rene L. Cruz, and George C. Polyzos. SCED: A generalized scheduling policy for guaranteeing quality-of-service. *IEEE/ACM transactions on networking*, 7(5):669–684, October 1999.

[SD10] Wilfried Steiner and Bruno Dutertre. Smt-based formal verification of a *TTEthernet* synchronization function. In Stefan Kowalewski and Marco Roveri, editors, *Formal Methods for Industrial Critical Systems - 15th International Workshop, FMICS 2010, Antwerp, Belgium, September 20-21, 2010. Proceedings*, volume 6371 of *Lecture Notes in Computer Science*, pages 148–163. Springer, 2010.

[SD11a] Wilfried Steiner and Bruno Dutertre. Automated formal verification of the *TTEthernet* synchronization quality. In Mihaela Gheorghiu Bobaru, Klaus Havelund, Gerard J. Holzmann, and Rajeev Joshi, editors, *NASA Formal Methods - Third International Symposium, NFM 2011, Pasadena, CA, USA, April 18-20, 2011. Proceedings*, volume 6617 of *Lecture Notes in Computer Science*, pages 375–390. Springer, 2011.

[SD11b] Wilfried Steiner and Bruno Dutertre. Layered diagnosis and clock-rate correction for the ttethernet clock synchronization protocol. In Leon Alkalai, Timothy Tsai, and Tomohiro Yoneda, editors, *17th IEEE Pacific Rim International Symposium on Dependable Computing, PRDC 2011, Pasadena, CA, USA, December 12-14, 2011*, pages 244–253. IEEE Computer Society, 2011.

[SD13] Wilfried Steiner and Bruno Dutertre. The TTEthernet synchronisation protocols and their formal verification. *IJCCBS*, 4(3):280–300, 2013.

[SDS08] Maria Sorea, Bruno Dutertre, and Wilfried Steiner. Modeling and verification of time-triggered communication protocols. In *11th IEEE International Symposium on Object-Oriented Real-Time Distributed Computing (ISORC 2008), 5-7 May 2008, Orlando, Florida, USA*, pages 422–428. IEEE Computer Society, 2008.

[SH13] Alexey Solovyev and Thomas C. Hales. Formal verification of nonlinear inequalities with Taylor interval approximations. In *NASA Formal Methods*, volume 7871 of *LNCS*, pages 383–397, 2013.

[SJ11] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *SAS*, pages 233–248, 2011.

[SNS$^+$17] Yasser Shoukry, Pierluigi Nuzzo, Alberto L. Sangiovanni-Vincentelli, Sanjit A. Seshia, George J. Pappas, and Paulo Tabuada. SMC: satisfiability modulo convex optimization. In *HSCC*, 2017.

[SP] Stefan H. Schmieta and Gabor Pataki. Reporting solution quality for the DIMACS library of mixed semidefinite-quadratic-linear programs. `http://dimacs.rutgers.edu/Challenges/Seventh/Instances/error_report.html`. [Online; accessed March 23, 2016].

[SRSP04] Wilfried Steiner, John M. Rushby, Maria Sorea, and Holger Pfeifer. Model checking a fault-tolerant startup algorithm: From design exploration to exhaustive fault simulation. In *2004 International Conference on Dependable Systems and Networks (DSN 2004), 28 June - 1 July 2004, Florence, Italy, Proceedings*, pages 189–198. IEEE Computer Society, 2004.

[SZ06] Jens Schmitt and Frank Zdarsky. The DISCO network calculator: a toolbox for worst case analysis., 01 2006.

[TTT03] Reha H Tütüncü, Kim C Toh, and Michael J Todd. Solving semidefinite-quadratic-linear programs using SDPT3. *Mathematical programming*, 2003.

[Tuc02] Warwick Tucker. A rigorous ODE solver and Smale's 14th problem. *Foundations of Computational Mathematics*, 2:53–117, 2002.

[VB96] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[Wan06] Ernesto Wandeler. *Modular performance analysis and interface based design for embedded real time systems.* PhD thesis, ETH Zurich, 2006.

[WGR$^+$16] Timothy E. Wang, Pierre-Loïc Garoche, Pierre Roux, Romain Jobredeaux, and Eric Feron. Formal analysis of robustness at model and code level. In Alessandro Abate and Georgios Fainekos, editors, *Proceedings of the 19th International Conference on Hybrid Systems: Computation and Control, HSCC 2016, Vienna, Austria, April 12-14, 2016*, pages 125–134. ACM, 2016.

[WT06] Ernesto Wandeler and Lothar Thiele. Real-Time Calculus (RTC) Toolbox, 2006.

[YFN$^+$10] Makoto Yamashita, Katsuki Fujisawa, Kazuhide Nakata, Maho Nakata, Mituhiro Fukuda, Kazuhiro Kobayashi, and Kazushige Goto. A high-performance software package for semidefinite programs: Sdpa 7. Technical Report B-460, Tokyo Institute of Technology, Tokyo, 2010.