ORSAY
N^o d'ordre: xxxx

UNIVERSITÉ DE PARIS-SUD

CENTRE D'ORSAY

THÈSE

présentée
pour obtenir

L'HABILITATION À DIRIGER DES RECHERCHES
DE L'UNIVERSITÉ PARIS-SUD

PAR

Sylvie PUTOT

—×—

SUJET:

Static Analysis of Numerical Programs and Systems

soutenue le 13 décembre 2012 devant la commission d'examen

Eric WALTER	Président
Jean-Claude BAJARD	Rapporteurs
Nicolas HALBWACHS	
Helmut SEIDL	
Jean GASSINO	Examinateurs
Michel RUEHER	
Jean SOUYRIS	
Eric GOUBAULT	Invités
Claude MARCHE	

Remerciements

Aux membres de mon jury, qui m'ont fait l'honneur de bien vouloir examiner mon travail. En premier lieu aux rapporteurs, pour l'attention bienveillante portée à mon mémoire. A Claude Marché, pour son soutien constant et actif dans la préparation de cette habilitation. A Eric Goubault, chef du laboratoire MEASI, qui a guidé mes premiers pas sur le sujet, et toujours encouragée depuis. Comment en quelques mots évoquer plus de 10 ans à bénéficier de son énergie et de sa curiosité scientifique? Plus généralement à mes collègues du CEA et de Digiteo; ce travail leur doit tout. Tout spécialement Karim Tekkal et Franck Védrine, grâce à qui l'aventure Fluctuat, avec ses hauts et ses bas, est restée un plaisir. A Khalil Ghorbal, mon premier étudiant en thèse, forcément unique.

A tous, merci!

Contents

Chapter 1

Introduction

Software-driven systems are pervasive in today's world, and grow more complex every day. While their - frequent - dysfunctions are mainly a hassle in most applications, the safety-critical systems can not allow bugs, their correct behavior has to be proved thoroughly. In that respect, the norms of the safety-critical industries, such as DO-178B [142] and its recent update DO-178C [143] for the aeronautic industry, are evolving towards replacing part of the tests by methods providing guarantees for all possible executions, such as static analysis.

Meanwhile, the complexity of computation in these safety-critical systems is quickly increasing. For instance, Airbus has been using fly-by-wire systems, that is flight control computers, to replace manual flight control, for all its commercial airplanes since the A320 program, officially launched in 1984. Of course, if correct, these computer-driven systems increase the safety and performance of the flight control. But proving their correction, whatever the conditions driven by the airplane's environment, is a challenge at the state of the art of validation. These systems now involve millions of lines of programs, and, this will be the point of this document, quite intricate numerical computations.

Indeed, the control and monitoring of physical systems needs computations in real numbers. Of course, these can only be approximated on a computer: most applications now use floating-point approximations, a flexible and efficient representation, well specified by the IEEE-754 standard [1]. By specifying in particular the size of the mantissa and the exponent of floating-point numbers, this norm greatly improved the portability of programs: we know how the rounding error committed at each arithmetic operation is bounded. But, even though these approximations are well specified at the arithmetic operations level, their propagation through a chain of computation is, as we will see, difficult to predict. They can lead to subtle problems - with potentially catastrophic consequences - all the more so because some of the good algebraic properties of real arithmetic, such as, for instance, associativity, are no longer true in floating-point arithmetic. The consequences can be of two natures. First, a result computed in finite precision can be only a rough approximation of the real result. But also, the behavior of the program using finite precision can be unexpected: for instance, an algorithm that was proved to converge very well in real numbers, can no longer converge in finite precision, especially if some stopping criterion is not well suited to the precision.

The impact of rounding errors on numerical computations, and particularly in algebraic processes, has been receiving attention since the very beginning of computers and numerical analysis, in a famous 1947 article by Von Neumann and Goldstine [156]. The interest has not dropped since, let us for instance mention two classic surveys, by Wilkinson [157] and Higham [99]. Also, trying to build automatic methods to

estimate the rounding errors was already of interest as early as 1958 [54], and has remained so ever since; let us highlight the work of Miller [127] which in 1975 aimed at searching numerical instabilities in a spirit close to our present work.

Moreover, this problem of the propagation of rounding errors is only part of the larger problem of validating numerical software. A difficulty, and not the least one actually when considering this larger problem of validating numerically a possibly large program, is to decide and specify which properties have to be proved, to increase the level of assurance that the software meets its goal. The absence of run-time errors, with a semantics handling floating-point computations, addressed for instance by the Polyspace[1] or Astree [42] analyzers, is the first property that must be ensured in this context. But then, how can we know for sure that the program computes what it is supposed to, and not just something that does not produce an execution error? Of course, in general, the question is really hard to answer. But we have information at our disposal, as the safety-critical software obey very strict rules of development, involving a lot of documentation and requirement specifications. There can then be different and complementary answers, that correspond to different levels of view on the software. Locally, we may have specifications of functional behavior on elementary symbols or portions of software. For instance, that such function, for a given range of inputs, should compute approximately some mathematical function, with not more than a given approximation error. Then, globally on a large part of the software, we may for instance want to prove that the values of some variables representing physical quantities remain within their expected ranges. Of course, this supposes to be able to model the environment precisely enough. And finally, locally or globally, we can prove that the use of finite precision in the computation does not significantly alter the result from the one that would be computed in real numbers. So that the end-user can decide whether his software meets the accuracy requirements, including modeling and computation errors.

My work takes inspiration from two worlds that take into account the use of finite precision numbers: the first is reliable computing, in which set-based methods are used to build computations that are guaranteed by construction to contain the real result; the second is static analysis by abstract interpretation, which aims at proving properties on existing programs.

Reliable computing The first tool for guaranteed computations, introduced by Moore, is interval arithmetic [133], which he used to mechanize error analysis in computations as soon as 1959 [134]. Directly rewriting the methods of numerical analysis with intervals gives very poor results, as dependencies between numerical values are not taken into account. This motivated some methods that became classic, such as the interval Newton method and methods to solve interval linear systems, for which Hansen was pioneer [92, 96, 93, 95]. Many extensions of interval arithmetic have been developed since then, in order to overcome the dependency problem: let us mention Hansen's generalized interval arithmetic [94] and affine arithmetic [33, 45], on which most of the work presented here is based. More accurate models include for instance Taylor model methods, introduced by Berz and Makino [12, 120], and used in particular for validated integration of ordinary differential equations. Ellipsoid arithmetic can also be an answer to the dependency problem, and could be especially interesting in the fixpoint computations induced by static analysis as it is well suited to determine stability regions of non-linear discrete dynamical systems, as noted for instance by Neumaier [136]. There is no simple way to compute with ellipsoids, which makes them scarcely used for verified computation although they received some attention quite early; but approximated computations with ellipsoids have been used successfully [137, 108], and we hope to use them in the future, in conjunction

[1] http://www.mathworks.com/products/polyspace/

with affine forms. These set-based methods can be supplemented for bounded program verification by constraint solving on real or floating-point numbers [126, 32]. Let us finally refer to Rump's survey article [144] for a large overview on the field of reliable computing. The main direction of my work is to try to build abstractions for static analysis on some of these methods.

Static analysis by abstract interpretation In contrast to dynamic analysis, which tests the program around possible executions, static analysis aims at proving properties on programs without execution, for potentially infinite sets of behaviors. But designing an analysis that can answer exactly and automatically, on all programs, whether a non-trivial property holds, is not possible. To make the problem tractable, one can make different choices. A first choice is to lose automation: for instance, interactive theorem provers can be used to assist the development of formal proofs: we will come back later to their use in the context of floating-point computation analysis. A second choice is to lose soundness and possibly miss some errors. In this case, tools are rather used as bug-finders, or to give some level of confidence on the correctness of programs. This is for instance the choice made by Grammatech[2] for its CodeSonar analysis tool. Finally, and this is the choice we made for the work presented here, we can lose completeness by *abstracting* (over or under approximating) the properties of interest. The analyzer may not be able to conclude about the satisfaction of a property and produce false alarms, that is indicate possible erroneous behavior, that cannot occur in reality. For a given property of interest, the art of abstraction is to find the optimal trade-off between expressiveness and cost of the analysis. When my work started, this problem of false alarms indeed sullied the reputation of the first commercial sound static analyzer for run-time errors, Polyspace verifier[3], which handles floating-point semantics. Demonstrating that a static analyzer could prove the absence of run-time errors for large scale aeronautic programs was actually the starting point of Cousot, Cousot, Feret, Mauborgne, Miné, Monniaux and Rival, in their design of Astrée [42].

Our work also makes the choice of soundness by abstract interpretation, but aims at verifying refined numerical properties, for safety-critical systems of a possibly large size. We introduced a new class of zonotopic abstract domains well suited to the analysis of numerical properties, and built a static analyzer named Fluctuat which is not specialized to programs from the aeronautic industry - the price to pay being less scalability for the analyses on these particular programs. Still, Fluctuat is able to analyze several dozens of thousands of lines of typical control command software in a few hours (of course, this is only indicative as the analysis time highly depends on the software analyzed).

Most of the work I present here is motivated by the practical verification needs of actors of the safety-critical industry, such as Airbus and IRSN (French Expert in nuclear and radiological risk). Both have been supporting our work since its starting point in 2000-2001 and ever since, Airbus first by its participation to the European project DAEDALUS[4], then through direct projects. However, in the long term, we can hope that these advances, developed in the context of safety-critical systems, will benefit to the numerically very complex programs from numerical simulation.

[2] http://en.wikipedia.org/wiki/GrammaTech

[3] http://www.mathworks.com/products/polyspace/

[4] Daedalus European IST project (2000-2002), http://www.di.ens.fr/ cousot/projects/DAEDALUS/

1.1 Analyzing the impact of rounding errors

Approximations and uncertainties lie everywhere in the numerical software that aim at modeling real phenomena. An additional source of approximation is due to the representation of real numbers on the computer, most of the time by floating-point numbers. The error due to the use of finite precision in computations is often overlooked for at least two reasons. First, considered locally on a particular operation, the rounding error as specified by the IEEE-754 norm [1], can seem insignificant compared to the other sources of uncertainties. Second, considered globally on a program, its accurate estimation is difficult. And most of the time its impact is indeed negligible.

Still, experience tells us that an issue directly due to rounding error, with potentially catastrophic consequences, can occur even on numerically not very complex programs. Take for instance the example of the Patriot anti-missile during the Gulf war, that widely missed its target because of an inaccurately computed time counter, that caused a major localisation problem [58]. The accident actually occurred with an implementation in 24 bits fixed point arithmetic, but a similar problem - with different amplitude though - would have occurred with floating-point numbers. Indeed, this time counter was regularly incremented by the constant 0.1, representing a time elapse. But constant 0.1, as harmless as it may seem to human eyes, is not represented exactly in a binary finite format, whatever the precision. This causes a new rounding error at each time increment, errors that accumulate. The computation of the time counter is thus affected, at each time increment, by the rounding of the result of the addition of the floating-point representation of 0.1 to the time counter, result which in general is also not representable and must be rounded. This makes the error actually not so obvious to predict manually, even for such a simple program[5]. And what makes things worse is that the problematic parts are most of the time buried among thousands of harmless lines of program, which calls for automatic tools able to extract and analyze the dubious parts.

Rounding error analysis As already mentioned, interest for the impact of the use of finite precision on the behavior of programs started as early as 1947. The next step, that is designing methods to automatically estimate this impact, soon followed, in 1958. Still, we cannot say the problem has been fully solved since, let us quickly discuss contemporary approaches for rounding error analysis.

A well-established dynamic method for the analysis of numerical programs is the CESTAC (Control and Stochastic Estimation of Rounding Errors) method [155], implemented in the CADNA[6] tool. In a few words, this method can be described as follows: the computation for each input is run several times, with random rounding: the deviation between these results gives the number of significant digits of the result, with some probability. But this result is valid only under the hypothesis that the rounding errors are independent centered random variables, hypothesis that is often wrong, as advocated with examples by Kahan [105]. Also, this method gives no insurance that a very high error can not occur, which may have catastrophic consequence even if this is with very small probability.

Another class of approaches to the estimation of the propagation of rounding errors relies on automatic differentiation, that computes values of function derivatives: a first order estimation of the sensitivity of the output of a computation to rounding error, or more generally to uncertainties, is given by derivatives or first order Taylor developments. This was used to estimate the propagation of rounding error [9, 118, 127, 115]. But these estimations of the errors can also be used to try to correct the results of finite precision

[5] http://proval.lri.fr/gallery/clockdrift.en.html
[6] http://www-pequan.lip6.fr/cadna/

computation. This is the aim of the CENA (French acronym for correcting numerical rounding errors) method [113, 114, 112], which relies on automatic differentiation to compute and compensate the effects of the first-order rounding errors.

Finally, formal proof using interactive theorem provers is also applied to the verification of floating-point algorithms. Harrison has been working on floating-point verification using HOL since 1995 [97]. Formalizations of floating-point arithmetics have been designed for several proof assistants [25, 18]. While we are mainly concerned with software verification in our work, hardware verification is more widely spread in formal proof, as hardware is most of the time designed in a more modular way, and such proofs as IEEE-754 compliance are both crucial and quite easily automated [145]. Indeed, the error in the floating-point division in an Intel Pentium processor in 1994 [139] certainly had a non-negligible impact for Intel. More recently, Boldo, Filliatre, Melquiond, Ayad and Marché have started developing proofs for floating-point arithmetic in Coq [15], with efforts towards automation [16, 7]. Pen and paper analyses of rounding errors constitute starting points for accurate computer-aided analyses of numerical algorithms, such as in Boldo's work [14]. But, for the time being, these analyses are not generic, they have to be conducted again for each new problem. And the construction of the proof may require a lot of work and expertise by the user, and prevent the analysis of very large programs.

The birth of Fluctuat Eric Goubault conceived, mid 2000, the project to write a static analyzer of finite precision computations. An idea was, so that the analysis would be the more automatic possible, that the specification would be implicit: the static analyzer should be able not only to handle floating-point computation, but also to compute its distance to the reference, the computation in a real number semantics.

He set in 2001 [73] the first ideas of an abstract domain that computes the floating-point value of a variable, by interval arithmetic, and its distance to the real value, decomposed along the program control points, by a sum of interval contributions. When I began in Eric Goubault's team at CEA (now the MEASI laboratory, standing for Modelling and Analysis of Interacting Systems) in January 2001, with a background more turned towards scientific computing than theoretical computer science, I was first in charge, by way of formation, to implement these first ideas. This was done in an emerging static analyzer for which a first structure had been designed [75], and which soon became FLUCTUAT [77]. Meanwhile Matthieu Martel, who had joined the team shortly before I did, formalized this same abstract domain [121].

It soon became clear that the analysis we had designed was under-valuing the numerical interest of affine arithmetic [33, 45], from which it had drawn its inspiration. Indeed, from an arithmetic which had been proposed in order to use affine relations between variables to improve interval arithmetic, we had reached an abstract domain that was purely non relational, based on intervals, even though it was adding a very valuable information: the provenance of rounding and approximation errors. Following that assessment, it took us several years, starting from 2002 and still on-going work, to design efficient abstract domains that also made use of the affine relations between variables. Coming back to the abstraction of real values of variables of programs, we proposed affine arithmetic based abstract domains, with zonotopic concretizations, which we called affine sets [81, 83, 84, 64, 65].

We then formalized the extension of these abstract domains to handle finite precision computations and rounding errors along with their provenance [85] (they had been already implemented in Fluctuat for some time [46]). Indeed, the relations that are true for real values are no longer true for floating-point values: but instead of using interval linearisation as Mine did [130], we developed an abstract domain relying on affine sets for both values and errors.

Let us first quickly introduce the notion of abstract domain, which is central to static analysis by abstract interpretation.

1.2 From set-based methods to static analysis

Abstract interpretation [38, 40] is a theory of approximation of program semantics. An analysis first relies on a concrete semantics, that is a description of the property of interest and the way it propagates through program instructions. In our case, the properties of interest will be the values of variables of numerical programs at each point of the program, considered both in real and floating-point numbers, and a decomposition of the discrepancy between these values.

The following step is to abstract this concrete semantics, that is to define a sound approximation of this concrete semantics, on which the property of interest (here over-approximations of values of variables and rounding errors) can be automatically inferred on programs. The abstract domain defines this approximation of discrete values. Abstract operators, that allow to compute sound abstractions of transfer functions must be defined. Finally, a partial order is needed, allowing to compare, join and intersect abstract elements: we will compute safety properties of programs, that is invariants, obtained by fixpoint computations in this partially ordered structures; a key point of abstract interpretation is that fixpoints of the concrete semantics can be soundly approximated in the abstract world, thus allowing the design of fully automatic abstract interpreters. Finally, abstract operators, that allow to compute sound abstractions of transfer functions, must be defined.

When using set-based methods, such as affine arithmetic, the abstract arithmetic operators will naturally follow from the guaranteed computation that was defined for use in numerical analysis. The difficulty is to define a partially ordered structure, on which abstract elements can be compared, and join and meet operations defined.

The abstract interpretation framework Classic abstract interpretation relies on Galois connections between partially ordered sets: a Galois connection between two partially ordered sets $(\mathcal{D}^\flat, \subseteq^\flat)$ and $(\mathcal{D}^\sharp, \subseteq^\sharp)$ is a function pair (α, γ) such that

1. $\alpha : \mathcal{D}^\flat \to \mathcal{D}^\sharp$ is monotonic,
2. $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$ is monotonic,
3. $\forall (x^\flat, x^\sharp) \in \mathcal{D}^\flat \times \mathcal{D}^\sharp,\ \alpha(x^\flat) \subseteq^\sharp x^\sharp \Leftrightarrow x^\flat \subseteq^\flat \gamma(x^\sharp)$

In this definition, $\mathcal{D}^\flat$ is the concrete domain, $\mathcal{D}^\sharp$ the abstract domain, α the abstraction function, and γ the concretization function. Here, $\alpha(x^\flat)$ is the best abstraction for $x^\flat$, and $\alpha(x^\flat) \subseteq^\sharp x^\sharp$ formalizes the fact that $x^\sharp$ is a sound abstraction for $x^\flat$.

Let then $f^\flat$ be an operator on the concrete domain. The best abstraction of $f^\flat$ is $\alpha \circ f^\flat \circ \gamma$, and a sound abstraction for $f^\flat$ is any operator $f^\sharp$ such that for all $x^\sharp \in \mathcal{D}^\sharp,\ (\alpha \circ f^\flat \circ \gamma)(x^\sharp) \subseteq^\sharp f^\sharp(x^\sharp)$.

This framework is well suited to some abstractions, such as intervals. However, it is sometimes too restrictive, as, for instance, there is sometimes no best abstraction of a concrete element. This will be the case for the affine arithmetic based abstract domains, and we will use a relaxed framework [40], which allows to rely on a concretization function only. We will see that the concretization function by which our abstraction by affine sets is defined, is the zonotope defining the inputs and variables of the program. And indeed, the union of two zonotopes is not a zonotope, and neither is the intersection of zonotopes.

Here, a concretization function from the abstract domain $(\mathcal{D}^\sharp, \subseteq^\sharp)$ to the concrete domain $(\mathcal{D}^\flat, \subseteq^\flat)$, these domains being posets, is simply a monotonic function $\gamma : \mathcal{D}^\sharp \to \mathcal{D}^\flat$. The abstract value $x^\sharp$ is a sound abstraction for $x^\flat$ if $x^\flat \subseteq^\flat \gamma(x^\sharp)$. The abstract operator $f^\sharp$ is a sound abstraction for the concrete operator $f^\flat$ if for all $x^\sharp \in \mathcal{D}^\sharp,\ (f^\flat \circ \gamma)(x^\sharp) \subseteq^\flat (\gamma \circ f^\sharp)(x^\sharp)$. There are no best values and operators abstraction. While in the Galois connection framework, if the best operator abstractions are used, the accuracy of an abstract domain is directly related to the accuracy of its concretization function, this is no longer the case here.

Numerical abstract domains The more basic abstract domains are non relational: they abstract independently the value of each variable. Intervals are a typical example of a non relational abstract domains. Relational abstract domains additionally abstract some relations between variables. A number of relational abstract domains have been defined, relying on concretization functions, that is ranges for values of tuples of variables, giving particular polyhedra. The first of these domains is the general convex polyhedra ($\sum_k a_k x_k \leq c$) abstract domain [44], discovering invariant linear inequalities among numerical variables of a program. In order to find trade-offs between cost and accuracy, were introduced particular polyhedra for which efficient abstract operators could be designed, among which zones ($x - y \leq c$) [128], octagons ($\pm x \pm y \leq c$) [129], polyhedra with two variables per equality ($ax + by \leq c$) [152], octahedra ($\pm x_j \pm \ldots \pm x_k \geq c$) [31], pentagons ($a \leq x \leq b \wedge x \leq y$) [119], or sub-polyhedra ($a \leq x \leq b \wedge \sum_k a_k x_k = c$) [116]. Another way to yield a scalable analysis is to fix the coefficients a_k in $\sum_k a_k x_k \leq c$: this is the starting idea of the template polyhedra [147]. The implementation of polyhedral abstract domains in finite precision may need some precaution [28]: part of the interest of interval polyhedra ($\sum_k [a_k, b_k] x_k \leq c$) [29] is to address this issue. The concrete semantics of a program often involves non-convex behaviors, in particular in the case of conditional branch statements. Designing possibly non-convex abstractions is thus of interest. Interval polyhedra have non convex concretization, and so have min and max invariants on max-plus polyhedra ($\max(\lambda_0, x_1 + \lambda_1, \ldots, x_n + \lambda_n) \leq \max(\mu_0, x_1 + \mu_1, \ldots, x_n + \mu_n)$ and $\min(\lambda'_0, x_1 + \lambda'_1, \ldots, x_n + \lambda'_n) \leq \min(\mu'_0, x_1 + \mu'_1, \ldots, x_n + \mu'_n)$) [4], that are well suited for handling disjunctions as in algorithms on strings or arrays. Some of these abstract domains aim at capturing specific patterns or behaviors, such as for instance the relation to a loop counter [50], the output of order 2 linear filters [49], or linear absolute value relations ($\sum_k a_k x_k + \sum_k b_k |x_k| \leq c$) [30]. Designing a static analyzer relying on such abstract domains thus requires to combine these abstract domains [43], in order to be able to handle sufficiently general programs.

We proposed a new class of abstract domains, all connected to affine forms, that are sufficiently non-specialized to handle a large class of programs, and which accuracy can be parameterized.

Zonotopic abstract domains The concretization function of affine sets - our affine arithmetic based abstract domains - is the zonotope including the inputs and variables of the program: the parameterization of zonotopes by affine forms defines a functional abstraction, providing linearized input-output relations. Although yielding sub-polyhedric relations only, the comparison of its accuracy to the accuracy of polyhedra is not straightforward, as the non linear operations can be very conveniently abstracted by first-order Taylor expansions. Indeed, the accuracy of abstract domains cannot be compared solely based on their geometric shapes, as would be tempting to do, but depends on the accuracy of operator abstractions.

The main difficulty, as we will see, was to propose efficient set-theoretic operations. This required several steps: a first ordered structure was proposed in 2006 [81], improved in preprints [83, 84], then enhanced with a meet operation [65]. This meet operation was handled by a particular logical product [91]

between the affine sets and an abstract domain on the noise symbols that play a role in the affine sets: this was the subject of the PhD thesis of Khalil Ghorbal that I co-supervised. In this case, the concretization of an abstract element is no longer a zonotope, but a more general polyhedron. Affine sets can then be extended quite easily to handle floating-point computations. We will also demonstrate how these domains can be used also for the analysis of sensitivity to input uncertainties, or the proof of functional properties of numerical programs.

Zonotopes are also successfully used in reachability analysis of hybrid systems [66]. Of course, the same problem of computing unions of zonotopes, and intersections of zonotopes with guards arises [67, 5].

Refining the results An issue of abstract interpretation is the possibility of so-called false alarms: when the analysis is not able to conclude about some property, or gives possibly larger than expected values or rounding errors for a program variable, how does the user know if the program can indeed lead to some erroneous behavior, or if the accuracy of the analysis is at stake? Building abstract domains that are always more accurate can only be a partial answer. In that purpose, we recently started, in the frame of the ANR project CPP[7] (Confidence, Proof, Probability), some work allowing to combine some deterministic and probabilistic information [20, 21], when for instance inputs or parameters of a program are only imperfectly known, but with more information than just a range: we combine this information, using a combination of probability boxes [53] and more precisely Dempster-Shafer structures [151] with affine sets. In the future, this can be used to compute sound abstractions of some statistical characteristics of uncertainties of the outputs of some algorithm: for instance, in many cases, on top of giving a guaranteed range, we will be able to say that the extreme values will be reached with extremely low probability. We hope to draw some inspiration from the work on stochastic analysis of uncertainties in numerical programs, for instance as treated in the MASCOT-NUM[8] (Stochastic Analysis Methods for Numerical Programs and Treatments) research group.

A further solution to complement the information given by over-approximations, is to compute under-approximations, that is sets of values which we know are reached for some sets of inputs within the specification of the inputs. Kaucher arithmetic [106] was designed in order to compute such under-approximations, but its scope of application is limited. Building on affine sets and recent work revisiting modal intervals and Kaucher arithmetic [69], we proposed in 2007 [82] an abstract domain to compute under-approximation of real values of variables. The distance between under and over approximations allows us to qualify the quality of the analysis. Moreover, as we will see, this under-approximation gives a way to define test cases that aim at reaching values close to the maximum values of given variables. I co-supervise a PhD student funded by a Digiteo DIM LSC[9] project, SANSCRIT, on these under-approximations, their extension for finite precision computations, and their application to robust control. This cross-fertilization between guaranteed computation and static analysis for numerical programs can hopefully benefit to each community. This is also among the aims of other projects in which I took part, such as the ANR project ASOPT[10] (Static analysis and Optimisation) and the Digiteo DIM LSC project PASO (Proof, Static Analysis and Optimisation).

[7] CPP (Confidence Proof Probability), ANR 2009-2012, http://www.lix.polytechnique.fr/ bouissou/cpp/

[8] GdR MASCOT NUM (Methods of stochastic analysis of codes and numerical treatments), http://www.gdr-mascotnum.fr/

[9] Digiteo research program: Software and Complex Systems, http://www.digiteo.fr/Digiteo_dim_lsc/

[10] ASOPT (Analyse Statique et OPTimisation), ANR Arpege 2008, http://asopt.inrialpes.fr/index.php/

Fixpoint computation As already stated, invariants of the program are computed as fixpoints of the functional describing the behavior of the program. More specifically, we are interested in the least fixpoint greater than the initial state.

A fixpoint of an operator f is an element x such that $f(x) = x$. Let $\mathrm{lfp}_x f$ denote the least fixpoint, if it exists, of f larger than x. Let $\mathcal{D}^\sharp$ a complete lattice and $f^\sharp$ a sound abstraction of the concrete function. The least fixpoint over the concrete domain can be soundly abstracted in the abstract domain, and Kleene's theorem gives a constructive way to compute these abstract least fixpoint: if $f^\sharp$ is monotonic, then the Kleene iteration $x^\sharp_{i+1} = f^\sharp(x^\sharp_i)$ converges towards $\mathrm{lfp}_{x^\sharp_0} f^\sharp$. However, if $\mathcal{D}^\sharp$ has infinite increasing chains, this iteration can take infinite time. A way of getting an over-approximation of the fixpoint in finite time in the general case is to use widening operators [38, 40], that is operators $\nabla^\sharp$ that compute an upper bound of two elements, such that the increasing chain $y^\sharp_0 = x^\sharp_0,\ y^\sharp_{i+1} = y_i \nabla^\sharp f^\sharp(y^\sharp_i)$ reaches a post fixpoint $y^\sharp_n$ (such that $f^\sharp(y^\sharp_n) \subseteq^\sharp y^\sharp_n$) after a finite time.

In our case, where we are not in complete lattices, we will however still use Kleene-like iterations with potentially a final widening to intervals, which will allow us to compute post fixpoints. One of the issues in static analysis is to be able to compute these fixpoints efficiently. A first question, which we partially answered, is on which classes of programs Kleene-like iterations with domains of affine sets give finite fixpoints. Some particular chaotic iterations can also improve these computations. However, Kleene iterations with numerical convergence accelerations and widening, can be quite slow. I participated to the introduction for static analysis of policy iteration methods, for the domain of intervals [37]. They were since extended to other abstract domains, and we are currently working on their extension for affine sets domains.

1.3 Analyzing real life programs

I implemented these abstract domains, worked on improvements of fixpoint computations, and experimented different ideas in the FLUCTUAT analyzer, starting from an emerging analyzer [75]. We carried out numerous case studies, on programs from the aeronautic and nuclear industries, for Airbus and IRSN, but also for Hispano-Suiza, Dassault aviation, EADS Astrium, Delphi, Continental, MBDA, Thales Avionics and Sagem. These case studies lead to joint papers [87, 46, 19], but also triggered new developments and ideas. They were conducted either through direct contracts or projects such as MODRIVAL[11] or EDONA[12] from the SYSTEMATIC pole, the ITI project of the European Space Agency SSVAI[13], the ITEA_2 European project ES_PASS, or the FP7 European project INTERESTED[14].

The design of a pre-industrial static analyzer such as Fluctuat, dedicated to complex properties, does not only require theoretical work and new abstract domains, but also a lot of everyday work to build a methodology that makes its use accessible to end-users. Interactions with these end-users lead us to gradually define a substantial assertion language, for instance allowing interactions with a continuous environment, but also for instance test case generation or proof of functional properties. The large quantity of information available for an analysis also required some thinking for the user interface. A glimpse of these aspects will be given in Chapter 5, but the reader can also refer to the Fluctuat user manual [88].

[11] MODRIVAL (2006-2008), Usine Logicielle, SYSTEMATIC pole, http://www.usine-logicielle.org/
[12] EDONA (2007-2009), SYSTEMATIC pole, http://www.edona.fr/
[13] SSVAI (Space Software Validation using Abstract Interpretation), ITI project from ESA (2007-2008)
[14] INTERESTED (2008-2010), FP7 European project, http://www.interested-ip.eu/

1.4 Organization of the document

This document synthesizes my work since I started at CEA-LIST, after my PhD, and mainly relies on selected parts of my publications during these years. Chapter 2 presents different numerical abstract domains relying on affine sets for semantics in real numbers [81, 64, 65, 83, 84]. Chapter 3 quickly describes two extensions of affine sets [82, 21]. Chapter 4 then discusses the crucial issue of fixpoint computations, and quickly describes a new method [37] based on policy iteration. Chapter 5 presents the Fluctuat static analyzer, with an emphasis on the extension of the affine sets abstract domains to handle finite precision and the case studies conducted with it [77, 80, 85, 79, 87, 46, 19, 22]. I conclude in Chapter 6 with some perspectives on the future of static analysis of numerical programs.

Chapter 2

Zonotopic Abstract Domains for Functional Abstraction

In this chapter, we describe the zonotopic abstract domains we developed between 2003 and today for the analysis of numerical properties of programs. These abstract domains are based on affine arithmetic [45], a more accurate version of interval arithmetic, that propagates affine correlations between variables. If computations rely on finite precision, these correlations do no longer hold exactly, but only approximately so. However, accurate abstractions are necessarily relational, that is have to capture relations between program variables. Even in order to abstract finite precision behavior, we thus have to start from a good abstraction of the corresponding idealized computations in real values. This is all the more the case if, like it is the case here, we want to compare the finite precision with the idealized computation. Chronologically, we first proposed to use affine arithmetic to improve the estimation of the propagation of rounding errors [80, 78], before deciding to concentrate on variables value [81]. This lead us to go in depth into the definition of a suitable ordered structure with efficient and accurate join and meet operators [83, 64, 84, 65], an aspect which is of primary importance for a use in static analysis.

We will first discuss in the present chapter these abstract domains for the analysis of programs operating on real-values variables, as we presented them in [83, 64, 84, 65]. The extension of these abstract domains to finite precision analysis will then be discussed in Chapter 5.

Classic relational abstract domains most often have abstract values which concretization are sub-polyhedra. The abstract domains we developed make no exception. We define abstract elements as tuples of affine forms, which we call affine sets; they describe particular polytopes, called zonotopes [160], which are bounded and center-symmetric. But these zonotopes are also explicitly parameterized, as affine combinations of symbolic variables, called noise symbols, that stand for uncertainties in inputs and parameters of the program, or approximations in the analysis. The affine sets then define a sound abstraction of relations that hold between the current values of the variables, for each control point, and the inputs of a program. This leads us to domains where abstract values are seen less as geometric envelopes than as enclosures of sets of functional behaviors in the sense of Jeannet et al. [102], providing linearized input-output relations, at low cost. The interests of such relations for static analysis are well-known [41], we mention but a few. We can design precise compositional inter-procedural abstractions, making our domain very scalable as we recently described [89], and sketched in Section 2.2.4. This information can also be used for proofs of complex invariants (involving relations between inputs and outputs), sensitivity analysis and test case generation, as

we will demonstrate in Section 3.1.2 and in Chapter 5 devoted to Fluctuat.

We will present in this chapter the abstract domain we built on this arithmetic and highlight the difficult points we are still working on, such as the join operator. Experimental results with its implementation in the Fluctuat analyzer will be mostly left for Chapter 5. More exploratory variations on this abstract domain will be discussed in Chapter 3.

Related work

Affine arithmetic and self validated methods Affine arithmetic, like interval arithmetic, has been introduced as a model for self-validated numerical analysis, also known as reliable or guaranteed computing. The aim in that context is to compute sets of values that are guaranteed to contain the real result, even when computing in finite precision, the reader can refer to Rump's recent survey paper [144].

The first important developments of interval arithmetic began in the 60's with Moore's work [134, 133]. The underlying idea is to compute the result of a function or a program no longer with numbers, but with intervals of numbers, often defined by their two bounds. Computations can thus be extended on intervals in such a way that an enclosure which is guaranteed to contain the exact result is obtained. For static analysis, where the aim is to prove some behavior of a program, one of the first numerical abstract domains [38] indeed relies on interval arithmetic. However, if applied by a direct extension of elementary operations to their interval counterpart, as is the natural way to apply it in static analysis for instance, the enclosure obtained is often too large. One of the main problem is that two correlated variables (for instance having same value in an interval) are treated as completely uncorrelated in interval arithmetic. Several improvements of interval arithmetic were proposed, among which affine arithmetic [33, 45], initially introduced for computer graphics applications. Among other such more accurate methods, we can also mention Hansen's generalized interval arithmetic [94], Taylor model methods [12, 120] (affine arithmetic is comparable, though better, to a first order Taylor method) or ellipsoidal arithmetic [137, 108, 136].

Zonotopes The concretization of an abstract element defined by affine forms is a zonotope. A large amount of work has been carried out using zonotopes, mostly in control theory [107, 34, 3, 35] and for reachability analysis in hybrid systems [66]. But in general, in hybrid systems analysis, no union operator is defined, whereas it is an essential feature of our work. Also, the methods used are purely geometrical: no information is kept concerning input/output relationships, e.g. as witnessed by the methods used for computing intersections [67]. Zonotopes have also been used in imaging, in collision detection for instance [90], where purely geometrical joins have been defined.

Contents

In Section 2.1, we quickly introduce the principles of affine arithmetic, and the link to zonotopes. We then describe in Section 2.2 the steps towards defining an abstract domain relying on this arithmetic: we will first, in Subsection 2.2.1, differentiate noise symbols that correspond to uncertainty on inputs, from noise symbols that are introduced to handle uncertainty in the analysis - this underlies a functional input-output abstraction - and define and characterize a functional order; then, in Subsection 2.2.2, we will shortly introduce constrained affine sets, that allow us to interpret tests; the join operator is then discussed, before hinting in Subsection 2.2.4 at the use of our abstract domain for modular analysis. We finally conclude with some short experiments.

2.1 Affine arithmetic and zonotopes

2.1.1 Affine arithmetic

Affine arithmetic is an extension of interval arithmetic introduced in 1993 in the context of computer graphics by Comba and Stolfi [33]. It operates over affine forms, that is affine combinations of some primitive variables called noise symbols, which stand for sources of uncertainty in the data or due to approximations made during the computation. It automatically derives first-order guaranteed approximations of arithmetic expressions, thus being especially well suited in the approximation of the enclosure of smooth functions.

An *affine form* is defined as a formal sum over a set of *noise symbols* ε_i

$$\hat{x} \stackrel{def}{=} \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i,$$

with $\alpha_i^x \in \mathbb{R}$ for all i. Each noise symbol ε_i stands for an independent component of the total uncertainty on the quantity $\hat{x}$, its value is unknown but bounded in [-1,1]; the corresponding coefficient α_i^x is a known real value, which gives the magnitude of that component. The same noise symbol can be shared by several forms, indicating correlations among them. These noise symbols can not only model uncertainty in data or parameters, but also uncertainty coming from computation. This last point is one more interest of affine arithmetic compared to interval arithmetic: it allows one to control the over-approximation due to the use of finite precision in the implementation of the analysis, while keeping a sound implementation.

In all the document, we will use the usual convention, that note affine forms by letters topped by a hat symbol. The set of values that a variable x defined by an affine form $\hat{x}$ can take, lies in the interval

$$[\alpha_0^x - \sum_{i=1}^{n} |\alpha_i^x|, \alpha_0^x + \sum_{i=1}^{n} |\alpha_i^x|].$$

The assignment of a variable x whose value is given in a range $[a, b]$, introduces a fresh noise symbol ε_i :

$$\hat{x} = \frac{(a+b)}{2} + \frac{(b-a)}{2}\,\varepsilon_i.$$

Affine forms are closed by affine operations: for two affine forms $\hat{x}$ and $\hat{y}$, and a real number r, we get

$$\hat{x} + r\hat{y} = (\alpha_0^x + r\alpha_0^y) + \sum_{i=1}^{n} (\alpha_i^x + r\alpha_i^y)\varepsilon_i$$

For non affine operations, we select an approximate linear resulting form. Bounds for the error committed using this approximate form are computed, and using these bounds a new noise term is added to the linear form. For instance, the non-affine part of the multiplication of $\hat{x}$ and $\hat{y}$, defined on the set of noise symbols $\varepsilon_1, \ldots, \varepsilon_n$, can be over-approximated by

$$\sum_{i=1}^{n} |\alpha_i^x \alpha_i^y| [0,1] + \sum_{1 \le i < j \le n} |\alpha_i^x \alpha_j^y + \alpha_j^x \alpha_i^y| [-1,1].$$

Introducing a new noise symbol ε_{n+1}, we then write

$$\hat{x}\times\hat{y} = \left(\alpha_0^x\alpha_0^y + \frac{1}{2}\sum_{i=1}^{n}|\alpha_i^x\alpha_i^y|\right) + \sum_{i=1}^{n}\left(\alpha_i^x\alpha_0^y + \alpha_i^y\alpha_0^x\right)\varepsilon_i + \left(\frac{1}{2}\sum_{i=1}^{n}|\alpha_i^x\alpha_i^y| + \sum_{1\le i<j\le n}|\alpha_i^x\alpha_j^y + \alpha_j^x\alpha_i^y|\right)\varepsilon_{n+1}.$$

Example 1 *Let us detail the interpretation of the program of example 1 with affine arithmetic.*

```
1  real a = [-2,0];
2  real b = [1,3];
3  real x = a + b;
4  real y = -a;
5  real z = x * y;
```

The assignments of a and b to a possible range of value create new noise symbols, say ε_1 and ε_2: $\hat{a} = -1+\varepsilon_1$ and $\hat{b} = 2+\varepsilon_2$. Affine expressions are handled exactly, that is we get $\hat{x} = 1+\varepsilon_1+\varepsilon_2$ and $\hat{y} = 1-\varepsilon_1$. The multiplication produces a new noise symbol, say ε_3, and we get $\hat{z} = 0.5+\varepsilon_2+1.5\varepsilon_3$. The range of z given by this affine form is $[-2,3]$ while the exact range is $[-2,2.25]$, and with interval arithmetic, z can only be proved to be in $[-2,6]$.

We could compute a more accurate approximate form for z, using semi-definite programming, but our experiments [64] showed that this improvement is in general not worth the additional computational price.

Different semantics have been given and studied for the non-affine arithmetic operations, but this is not central here. We will mostly focus on the use of affine arithmetic in static analysis, and more precisely the definition of an ordered structure.

2.1.2 Affine sets and zonotopes

In what follows, we introduce matrix notations to handle tuples of affine forms, which we call *affine sets*. We then characterize the geometric concretization of sets of values taken by these affine sets.

We note $\mathcal{M}(n,p)$ the space of matrices with n lines and p columns of real coefficients. An affine set expressing the set of values taken by p variables over n noise symbols $\varepsilon_i,\ 1\le i\le n$, can be represented by a matrix $A\in\mathcal{M}(n+1,p)$. When these matrices represent affine spaces such as is the case in this section, we will number their rows starting from 0, while their columns start from 1. For $A\in\mathcal{M}(n+1,p)$, we note $A_+\in\mathcal{M}(n,p)$ the matrix obtained by keeping all lines but the first one. Finally, for a vector $e\in\mathbb{R}^n$, we will need its l_∞ and l_1 norms, defined by

$$\|e\|_\infty = \max_{1\le i\le n}|e_i| \text{ and } \|e\|_1 = \sum_{i=1}^{n}|e_i|.$$

Example 2 *Consider the affine set*

$$\hat{x} = 20 - 4\varepsilon_1 + 2\varepsilon_3 + 3\varepsilon_4 \quad (2.1)$$

$$\hat{y} = 10 - 2\varepsilon_1 + \varepsilon_2 - \varepsilon_4, \quad (2.2)$$

we have $n=4$, $p=2$ and : ${}^tA = \begin{pmatrix} 20 & -4 & 0 & 2 & 3 \\ 10 & -2 & 1 & 0 & -1 \end{pmatrix}$.

We formally define the concretization of affine sets by :

Definition 1 (Geometric concretization) *Let an affine set with p variables over n noise symbols, defined by a matrix $A \in \mathcal{M}(n+1, p)$. Its geometric concretization is the zonotope*

$$\gamma(A) = \left\{ {}^tA \begin{pmatrix} 1 \\ e \end{pmatrix} \mid e \in \mathbb{R}^n, \|e\|_\infty \leq 1 \right\} \subseteq \mathbb{R}^p.$$

A zonotope [160] defining p variables over n noise symbols, is the affine image of a n-hypercube in a p-dimensional space: it is a center-symmetric bounded convex polyhedra, which faces are zonotopes of lower dimension.

Example 3 *For instance, Figure 2.1 represents the concretization of the affine set defined by (2.1) and (2.2). It is a zonotope, with center $(20, 10)$ given by the vector of constant coefficients of the affine forms. A zonotope in $\mathcal{M}(n+1, p)$ can also be seen as the Minkowski sum of n segments in $\mathbb{R}^p$: for instance here, $n = 4$ and $p = 2$, we represent in Figure 2.2 the steps of the construction of the zonotope of Figure 2.1, taking $n = 1$, $n = 2$, $n = 3$. Direction a_i is given by the $i+1$-th line of matrix A.*

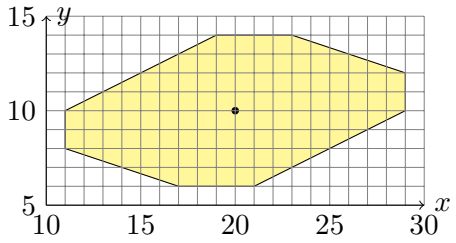


Figure 2.1: Geometric concretization $\gamma(A)$ of affine set {(2.1)-(2.2)}

Definition 2 (Linear concretization) *Let an affine set defined by matrix $A \in \mathcal{M}(n, p)$. We call its linear concretization the zonotope centered on 0*

$$\gamma_{lin}(A) = \left\{ {}^tAe \mid e \in \mathbb{R}^n, \|e\|_\infty \leq 1 \right\} \subseteq \mathbb{R}^p.$$

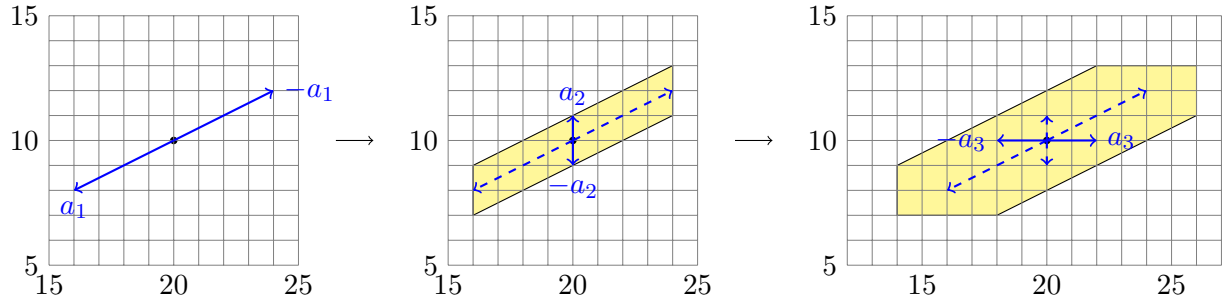


Figure 2.2: Construction of the geometric concretization of affine set {(2.1)-(2.2)}: from left to right, n from 1 to 3

The definition and study of these concretization and linear concretization will be both useful, as the concretization of an element X of our abstract domain will be the Minkowski sum of a central zonotope $\gamma(C^X)$ and a perturbation zonotope centered on 0, $\gamma_{lin}(P^X)$. There is an easy way to characterize the linear concretization $\gamma_{lin}(A)$:

Lemma 1 *For matrices $X \in \mathcal{M}(n_x, p)$ and $Y \in \mathcal{M}(n_y, p)$, we have $\gamma_{lin}(X) \subseteq \gamma_{lin}(Y)$ if and only if $\|Xu\|_1 \leq \|Yu\|_1$ for all $u \in \mathbb{R}^p$.*

The following lemma now characterizes the geometric ordering on zonotopes, i.e. extends the result of the previous lemma to zonotopes that are no longer centered on zero:

Lemma 2 *For matrices $X \in \mathcal{M}(n_x + 1, p)$ and $Y \in \mathcal{M}(n_y + 1, p)$, $\gamma(X) \subseteq \gamma(Y)$ if and only if, for all $u \in \mathbb{R}^p$,*

$$\left| \sum_{i=1}^{p} (y_{0,i} - x_{0,i}) u_i \right| \leq \|Y_+ u\|_1 - \|X_+ u\|_1$$

As for polyhedra, there is no pseudo-inverse α to this "concretization" operation, γ, i.e. there is no best over-approximation of a set of points in $\mathbb{R}^n$ by a zonotope. The abstract interpretation we can define with zonotopes is thus concretization-based, but not based on a Galois connection [40].

2.2 An ordered structure for functional abstraction

Static analysis by abstract interpretation requires the underlying abstract domain to be an ordered structure. For instance, intervals, where an interval $[a, b]$ with $a > b$ represents the empty interval, and with

$$\begin{aligned} [a,b] \subseteq [c,d] &\Leftrightarrow a \leq c \text{ and } d \leq b \\ [a,b] \cup [c,d] &= [\min(a,c), \max(b,d)] \\ [a,b] \cap [c,d] &= [\max(a,c), \min(b,d)] \end{aligned}$$

is a complete lattice. The situation with affine sets is more problematic, as the union of two zonotopes is in general not a zonotope, and neither is the intersection of two zonotopes.

A first idea to build an ordered structure on affine sets, is to transpose the order and join operations from intervals to the affine sets, by defining affine sets with interval coefficients, and defining the lattice operations component-wise on these coefficients. This would be sound and presents the interest of handling in a trivial way the implementation in finite precision of affine sets. However, we would lose much of the interest of affine forms, and again come across the problem of loss of correlation inherent to intervals.

Another natural idea is to define the order on affine sets by the inclusion order on the corresponding geometric concretization as zonotopes, that is $X \leq Y$ iff $\gamma(X) \subseteq \gamma(Y)$. We do not make this choice, as we want an abstraction that expresses input/output dependencies. We will thus define a stronger order on affine sets, which preserves this functional abstraction: it will express the inclusion order on the geometric concretization of the affine set augmented by the inputs of the program. For that, we will introduce perturbed affine sets, on which we define this (pre-)order relation and a join operator. The intersection of zonotopes, or of a zonotope and a hyper-plane cannot generally well represented as a zonotope, we will thus then introduce constrained affine sets, that allow to interpret tests quite naturally. Affine arithmetic can be

naturally extended to define abstract transformers for arithmetic operations for constrained affine sets, this is not detailed here. Finally, as a natural application, we demonstrate that this abstract domain is well suited to be used for an accurate modular analysis. For details and proofs on the results stated in this section, we refer the reader to [83, 84, 65, 89, 74].

2.2.1 Perturbed affine sets and functional order

As stated in Section 2.1, noise symbols in affine arithmetic model uncertainty both on the inputs and on the analysis: in order to define a functional abstraction, we will for now on distinguish these two kinds of noise symbols:

- the noise symbols that correspond to uncertain inputs will be called *central* noise symbols and noted ε_i. They model uncertainty in data or parameters, and are used to produce input-output relations.
- the noise symbols handling the uncertainty due to the analysis will be called *perturbation* noise symbols and noted η_j. We will have more flexibility on these symbols as they do not have any meaning in terms of input-output relations.

Example 4 *For instance, the multiplication in example 1 yields a symbol η_1 instead of a ε_3: we have $z = 0.5 + \varepsilon_2 + 1.5\eta_1$, where $0.5 + \varepsilon_2$ indicates the linearized correlation to the input, while $1.5\eta_1$ indicates the possible perturbation from this linearized form.*

Definition 3 (Perturbed affine sets) *We define a perturbed affine set X, expressing p variables over n central noise symbols and m perturbation noise symbols by the pair of matrices $(C^X, P^X) \in \mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$. The k^{th} variable of X is described by*

$$\hat{x}_k = c^X_{0k} + \sum_{i=1}^{n} c^X_{ik}\varepsilon_i + \sum_{j=1}^{m} p^X_{jk}\eta_j.$$

In what follows, we will always handle perturbed affine sets, but, for more simplicity, simply call them affine sets.

The geometric concretization of perturbed affine sets X is thus defined as the Minkowski sum [11] of a *central* zonotope $\gamma(C^X)$ and of a *perturbation* zonotope centered on 0, $\gamma_{lin}(P^X)$.

An affine set $X = (C^X, P^X) \in \mathcal{M}(n+1, p) \times \mathcal{M}(m+1, p)$ over n input noise symbols ε_i, abstracts a set of functions of the form $x : \mathbb{R}^n \to \mathbb{R}^p$, with a relational function-abstraction in the sense of Jeannet et al. [102]. We write $x_1, \ldots, x_p$ its p components. X gives an over-approximation of the set of values that $(\varepsilon_1, \ldots, \varepsilon_n, x_1, \ldots, x_p)$ can take conjointly. In terms of sets of functions from $\mathbb{R}^n$ to $\mathbb{R}^p$, its concretization is

$$\gamma_f(X) = \left\{ f : \mathbb{R}^n \to \mathbb{R}^p \mid \forall \epsilon \in [-1,1]^n, \exists \eta \in [-1,1]^m, f(\varepsilon) = {}^tC^X \begin{pmatrix} 1 \\ \varepsilon \end{pmatrix} + {}^tP^X \eta \right\}.$$

The functional order is given naturally, as for any concretization-based abstract interpretation [40], by $X \leq_f Y$ iff $\gamma_f(X) \subseteq \gamma_f(Y)$, which is equivalent to $\gamma(\tilde{X}) \subseteq \gamma(\tilde{Y})$, where the augmented zonotope $\gamma(\tilde{X})$ is defined

by

$$\tilde{X} = \begin{pmatrix} \begin{pmatrix} {}^t 0_n \\ I_{n\times n} \end{pmatrix} & C^X \\ 0_{m\times n} & P^X \end{pmatrix} \tag{2.3}$$

with $I_{n\times n}$ the identity matrix of $\mathcal{M}(n,n)$, 0_n the vector with $n+1$ components equal to 0, and $0_{m\times n}$ the null matrix of $\mathcal{M}(m,n)$. $\gamma(\tilde{X})$ is the zonotope augmented by the input noise symbols.

Now, this order on the augmented zonotopes can be directly reformulated in terms of the current parameterization (C^X, P^X) of abstract values for variables $x_1, \ldots, x_p$, as formalized in Definition 4 and Lemma 3:

Definition 4 (Order relation) *Let two affine sets over p variables, n input noise symbols and m perturbation noise symbols, $X = (C^X, P^X)$ and $Y = (C^Y, P^Y)$ in $\mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$. We say that $X \leq Y$ iff*

$$\sup_{u\in\mathbb{R}^p} \left(\|(C^Y - C^X)u\|_1 + \|P^X u\|_1 - \|P^Y u\|_1 \right) \leq 0$$

Lemma 3 *The binary relation $\leq$ of Definition 4 is a pre-order, and we have*

- *$X \leq Y$ is equivalent to $X \leq_f Y$,*
- *$X \leq Y$ always implies $\gamma(X) \subseteq \gamma(Y)$.*

The equivalence relation $X \sim Y$ generated by this pre-order can be characterized by $C^X = C^Y$ and $\gamma_{lin}(P^X) = \gamma_{lin}(P^Y)$.

We still denote $\leq$ / $\sim$ by $\leq$ in the rest of the text.

Example 5 *Consider the two affine sets: X defined by $x_1 = 1 + \varepsilon_1$ and $x_2 = 1 + \eta_1$, and Y defined by $y_1 = 1 + \eta_2$ and $y_2 = 1 + \eta_1$. We have*

$$C^X = \begin{pmatrix} 1 & 1 \\ 1 & 0 \end{pmatrix}, P^X = \begin{pmatrix} 0 & 1 \\ 0 & 0 \end{pmatrix}, C^Y = \begin{pmatrix} 1 & 1 \\ 0 & 0 \end{pmatrix}, P^Y = \begin{pmatrix} 0 & 1 \\ 1 & 0 \end{pmatrix}.$$

Their concretizations are the same: $\gamma(X) = \gamma(Y) = [0,3] \times [0,3]$, and $X \leq Y$, but $Y \not\leq X$. Indeed, they do not express the same input-output relations: in X, the first variable has direct dependency to the input represented by ε_1, while there is no known dependency in Y, or between Y and the input. Otherwise stated, the zonotope representing $(\varepsilon_1, x_1, x_2)$ is included in the zonotope for $(\varepsilon_1, y_1, y_2)$, but the contrary does not hold: (ε_1, x_1) is a line while (ε_1, y_1) is a box. This can also be seen using Definition 4: for all $u = {}^t(u_1, u_2)$,

$$\left\|(C^X - C^Y)u\right\|_1 + \left\|P^Y u\right\|_1 - \left\|P^X u\right\|_1 = \left\| \begin{matrix} 0 \\ u_1 \end{matrix} \right\|_1 + \left\| \begin{matrix} u_2 \\ u_1 \end{matrix} \right\|_1 - \left\| \begin{matrix} 0 \\ u_2 \end{matrix} \right\|_1 = 2 \mid u_1 \mid .$$

So $\|(C^X - C^Y)u\|_1 + \|P^Y u\|_1 - \|P^X u\|_1 > 0$ for all $u = {}^t(u_1, u_2)$ such that $u_1 \neq 0$.
But we have $X \leq Y$: indeed, $\|(C^X - C^Y)u\|_1 + \|P^X u\|_1 - \|P^Y u\|_1 = 0$, for all u.

Note that this order is computable:

Lemma 4 *Let two affine sets $X = (C^X, P^X)$ and $Y = (C^Y, P^Y)$ in $\mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$. The test $X \leq Y$ is decidable, with a complexity bounded by a polynomial in p and an exponential in $n + m$.*

Hopefully, there is no need to use this general decision procedure in practice for static analysis. In some cases, the order on affine sets can be efficiently computed using the order independently on each affine form:

Lemma 5 *Let $(t_k)_{k=1,\ldots,p}$ an orthonormal (in the sense of the standard scalar product in $\mathbb{R}^p$) basis of $\mathbb{R}^p$, and $X = (C^X, P^X)$ and $Y = (C^Y, P^Y)$ two affine sets, such that P^X and P^Y are Minkowski sums of segments $\lambda_k^X t_k$ and $\lambda_k^Y t_k$ respectively, where λ_k^X and λ_k^Y are any real numbers. Then $X \leq Y$ if and only if for all $k = 1, \ldots, p$,*

$$\|(C^Y - C^X)t_k\|_1 \leq \|P^Y t_k\|_1 - \|P^X t_k\|_1$$

A particular case is given by $t_1 = (1, 0, \ldots, 0)$, ..., $t_k = (0, \ldots, 1, \ldots, 0)$, ..., $t_p = (0, \ldots, 0, 1)$, the canonical basis of $\mathbb{R}^p$: $X \leq Y$ is equivalent to component-wise inclusion, that is $\hat{x}_k \leq \hat{y}_k$ for all $k = 1, \ldots, p$, which can be decided in $O((m + n)p)$.

In the general case, component-wise inclusion can still be used as a necessary condition for inclusion. So that the full inclusion will be computed only if this weaker condition is satisfied. Moreover, we will define a join operator that introduces a box perturbation, so that we will often be able to use this more efficient criterion.

Example 6 *Consider, for all noise symbols in $[-1, 1]$, affine sets X and Y defined by $x_1 = 1 + \varepsilon_1,\ x_2 = 1 + \varepsilon_2$, and $y_1 = 1 + \eta_1,\ y_2 = 1 + \eta_1$. We have $x_1 \leq y_1$ and $x_2 \leq y_2$. However we do not have $X \leq Y$. But Z, defined by $z_1 = 1 + \eta_2$ and $z_2 = 1 + \eta_3$, with a box perturbation, is an upper bound of X and Y.*

2.2.2 Constrained affine sets: handling zonotopes intersection

There is no natural intersection on zonotopes. We proposed in 2010 [65] in the context of K.Ghorbal's PhD thesis, an approach to handle these intersections, introducing an abstract domain on the noise symbols to take into account the constraints induced by tests. In order to give a short intuition on that approach, let us consider the following program:

```
1  x = [0,1];     // (1) x = 0.5 + 0.5 eps1
2  y = 2*x;       // (2) y = 1 + eps1 in [0,2]
3  if (y >= 1)       (3)
4     z = y -x;      (4)
```

Interpreting this example simply using a reduced product of affine sets with boxes, we have at control point (3) $y = 1 + \varepsilon_1 \cap [1, 2]$ and $x = 0.5 + 0.5\varepsilon_1$. We deduce at control point (4) $z = 0.5 + 0.5\varepsilon_1 \cap ([1, 2] - [0, 1]) \in [0, 1]$. Indeed, the constraint on y does not influence the value of x.

Now, instead of considering the noise symbols to be always in $[-1, 1]$, we interpret the constraints introduced by the tests on the noise symbols in an abstract domain, for instance intervals. At point (3) we now have $y = 1 + \varepsilon_1$ and $x = 0.5 + 0.5\varepsilon_1$, with $\varepsilon_1 \in [0, 1]$, at point (4) $z = 0.5 + 0.5\varepsilon_1$ with $\varepsilon_1 \in [0, 1]$, and thus $z \in [0.5, 1]$.

This can be formalized by introducing the logical product of the domain $\mathcal{A}_1$ of perturbed affine sets, with a lattice $(\mathcal{A}_2, \leq_2, \cup_2, \cap_2)$ that abstracts the values of the noise symbols ε_i and η_j. Formally, supposing that we have n noise symbols ε_i and m noise symbols η_j, we are given a concretization function $\gamma_2 : \mathcal{A}_2 \to \mathcal{P}(\{1\} \times \mathbb{R}^n \times \mathbb{R}^m)$.

Definition 5 (Constrained affine sets) *A constrained affine set U is a pair $U = (X, \Phi^X)$ where $X = (C^X, P^X)$ is an affine set, and Φ^X is an element of $\mathcal{A}_2$. Equivalently, we write $U = (C^X, P^X, \Phi^X)$.*

Abstractions of constraints on the noise symbols can be performed with any abstract domain that will not make our abstraction too costly, for instance intervals, zones, octagons. Let us now define an order relation on the product domain $\mathcal{A}_1 \times \mathcal{A}_2$. First, $X = (C^X, P^X, \Phi^X) \leq Y = (C^Y, P^Y, \Phi^Y)$ should imply that $\Phi^X \leq_2 \Phi^Y$, i.e. the range of values that noise symbols can take in form X is smaller than for Y. Then, we have to adapt Definition 4 for noise symbols no longer defined in $[-1, 1]$, but in the range of values Φ^X common to X and Y. Noting that:

$$\|C^X u\|_1 = \sup_{\varepsilon_i \in [-1,1]} |\langle \varepsilon, C^X u \rangle|,$$

where $\langle ., . \rangle$ is the standard scalar product of vectors in $\mathbb{R}^{n+1}$, we define:

Definition 6 (Pre-order on constrained affine sets) *Let X and Y be two constrained affine sets expressing p variables. We say that $X \leq Y$ iff $\Phi^X \leq_2 \Phi^Y$ and, for all $t \in \mathbb{R}^p$,*

$$\sup_{(\varepsilon,-)\in\gamma_2(\Phi^X)} |\langle (C^Y - C^X)t, \varepsilon \rangle| \leq \sup_{(-,\eta)\in\gamma_2(\Phi^Y)} |\langle P^Y t, \eta \rangle| - \sup_{(-,\eta)\in\gamma_2(\Phi^X)} |\langle P^X t, \eta \rangle| \ .$$

The binary relation defined in Definition 6 is a pre-order on constrained affine sets which coincides with Definition 4 in the "unconstrained" case, that is when $\Phi^X = \Phi^Y = [-1, 1]^{n+m}$.

Definition 7 *Let X be a constrained affine set, expressing p variables. Its concretization in $\mathcal{P}(\mathbb{R}^p)$ is*

$$\gamma(X) = \left\{ {}^t C^X \varepsilon + {}^t P^X \eta \mid \epsilon, \eta \in \gamma_2(\Phi^X) \right\} \ .$$

As for affine sets, the order relation of Definition 6 is stronger than the geometric order: if $X \leq Y$ then $\gamma(X) \subseteq \gamma(Y)$.

In the case of an equality test, we can go further than on inequality tests, and also modify the affine forms: one of the noise symbols involved in the test can be substituted in the affine sets using the relation induced by the test. The test will thus be exactly satisfied. Of course, we can also keep these constraints on noise symbols, in order to use a constraint solver when needed.

Example 7 *Consider, with an interval domain for the noise symbols, $Z = [\![x_1 == x_2]\!]X$ where*

$$\begin{cases} \Phi^X = 1 \times [-1, 1] \times [-1, 1] \times [-1, 1] \\ \hat{x}_1 = 4 + \varepsilon_1 + \varepsilon_2 + \eta_1, \quad \gamma(\hat{x}_1) = [1, 7] \\ \hat{x}_2 = -\varepsilon_1 + 3\varepsilon_2, \quad \gamma(\hat{x}_2) = [-4, 4] \end{cases}$$

We look for $\hat{z} = \hat{x}_1 = \hat{x}_2$, with $\hat{z} = z_0 + z_1\varepsilon_1 + z_2\varepsilon_2 + z_3\eta_1$. Using $\hat{x}_1 - \hat{x}_2 = 0$, i.e.

$$4 + 2\varepsilon_1 - 2\varepsilon_2 + \eta_1 = 0, \tag{2.4}$$

and substituting η_1 in $\hat{z} - \hat{x}_1 = 0$, we deduce $z_0 = 4z_3$, $z_1 = 2z_3 - 1$, $z_2 = -2z_3 + 3$. The abstraction in intervals of constraint (2.4) yields tighter bounds on the noise symbols: $\Phi^Z = 1 \times [-1, -0.5] \times [0.5, 1] \times$

$[-1, 0]$. *We now look for* z_3 *that minimizes the width of the concretization of* $\hat{z}$*, that is* $0.5|2z_3 - 1| + 0.5|3 - 2z_3| + |z_3|$*. A straightforward* $O((m + n)^2)$ *method to solve the problem evaluates this expression for* z_3 *successively equal to* 0*,* 0.5 *and* 1.5*: the minimum is reached for* $z_3 = 0.5$*. We then have*

$$\begin{cases} \Phi^Z = 1 \times [-1, -0.5] \times [0.5, 1] \times [-1, 0] \\ \hat{z}_1 = \hat{z}_2 = 2 + 2\varepsilon_2 + 0.5\eta_1, \quad \gamma(\hat{z}_1) = \gamma(\hat{z}_2) = [2.5, 4] \end{cases}$$

Note that the concretization $\gamma(\hat{z}_1) = \gamma(\hat{z}_2)$ *is not only better than the intersection of the concretizations* $\gamma(\hat{x}_1)$ *and* $\gamma(\hat{x}_2)$ *which is* $[1, 4]$*, but also better than the intersection of the concretization of affine forms* $(\hat{x}_1)$ *and* $(\hat{x}_2)$ *for noise symbols in* Φ^Z*. Note also that there is not always a unique solution minimizing the width of the concretization.*

Among the interests of this approach of constrained affine sets, we can state two points. It allows to define abstract values that are no longer necessarily zonotopes. It is also computationally very efficient. For instance, when expressing a constraint on a possibly complicated expression, we do not have to inverse all operations in order to propagate the constraints on all variables involved in the expression. This is naturally done by sharing noise symbols between variables.

More details can be found in our 2010 paper [65] and in K. Ghorbal's PhD thesis [63].

2.2.3 A join operator

In general, there exists no least upper bound operator on affine sets. We define here a join operator for affine sets which, in some cases, gives a minimal upper bound, and in any case gives an efficient upper bound operator. For that, we proceed in two phases:

- We first study the one-dimensional case, that is when at most one component is different in the two values joined. We define in this case a join operator which gives a minimal upper bound in most cases, and an upper bound in other cases.
- We then derive a full p-dimensional join operator on affine sets, using either component-wise the one-dimensional join, or affine relations between program variables common to the abstract values joined.

We restrict ourselves here to partial results, mostly in the unconstrained case. More details can be found in preprints [83, 84], and for the constrained case to our 2010 paper [65] and to Khalil Ghorbal's PhD thesis [63], which studies this problem in detail.

Note that we could also have defined upper bounds by using geometric join operations on the augmented zonotopes. But these are costly operations in general, the approach we propose is both efficient and convenient to use in order to propagate input-output relations.

Let us first recall the definition of a minimal upper bound (mub):

Definition 8 (Minimal upper bound) *Let* $\sqsubseteq$ *be a partial order on a set* X*. We say that* z *is a* mub *of two elements* x, y *of* X *if and only if*

- z *is an upper bound of* x *and* y*, i.e.* $x \sqsubseteq z$ *and* $y \sqsubseteq z$*,*
- *for all* z' *upper bound of* x *and* y*,* $z' \sqsubseteq z$ *implies* $z = z'$*.*

The one-dimensional case

We will use the following definitions to (partially) characterize minimal upper bounds in the one-dimensional case:

Definition 9 (Generic positions) *Let $\boldsymbol{x}$ and $\boldsymbol{y}$ be two intervals. We say that $\boldsymbol{x}$ and $\boldsymbol{y}$ are in generic positions if, whenever $\boldsymbol{x} \subseteq \boldsymbol{y}$, $\inf \boldsymbol{x} = \inf \boldsymbol{y}$ or $\sup \boldsymbol{x} = \sup \boldsymbol{y}$. By extension, we say that two affine forms $\hat{x}$ and $\hat{y}$ are in generic position when $\gamma(\hat{x})$ and $\gamma(\hat{y})$ are intervals in generic positions.*

For two real numbers α and β, let $\alpha \wedge \beta$ denote their minimum and $\alpha \vee \beta$ their maximum. We define

$$\text{argmin}_{|.|}(\alpha, \beta) = \{\gamma \in [\alpha \wedge \beta, \alpha \vee \beta], |\gamma| \text{ minimal}\}$$

The one-dimensional join computes in $O(n + m)$ time, a mub in some cases, or a tight upper bound in all cases:

Definition 10 (One-dimensional join) *Let two affine sets X and Y in $\mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$ such that at least $p - 1$ components (or variable values) are equal: for all $l \in [1, p]$, $l \neq k$, $\hat{x}_l = \hat{y}_l$. We define $Z = X \sqcup_1 Y$ by:*
- if $X \subseteq Y$ then $Z = Y$ else if $Y \subseteq X$ then $Z = X$
- else for all $l \in [1, p]$, $l \neq k$, $\hat{z}_l = \hat{x}_l = \hat{y}_l$ and $\hat{z}_k = \hat{x}_k \sqcup_1 \hat{y}_k$ is defined by:

- $c^Z_{0,k} = mid\,(\gamma(\hat{x}_k) \cup \gamma(\hat{y}_k))$
- $c^Z_{i,k} = \textit{argmin}_{|.|}(c^X_{i,k}, c^Y_{i,k}),\ \forall i = 1 \in [1, n]$
- $p^Z_{j,k} = \textit{argmin}_{|.|}(p^X_{j,k}, p^Y_{j,k}),\ \forall j \in [1, m]$
- $p^Z_{m+k,k} = \sup \gamma(\hat{x}_k) \cup \gamma(\hat{y}_k) - c^Z_{0,k} - \sum_{i=1}^{n} |c^Z_{i,k}| - \sum_{j=1}^{m} |p^Z_{j,k}|$
- $p^Z_{m+k,l} = 0,\ \forall l \in [1, p],\ l \neq k$

Lemma 6 *Under the hypotheses of Definition 10, $Z = X \sqcup_1 Y$ is an upper bound of X and Y such that, considering the interval concretizations, $\gamma(\hat{z}_k) = \gamma(\hat{x}_k) \cup \gamma(\hat{y}_k)$. And it is a minimal upper bound whenever $\gamma(\hat{x}_k)$ and $\gamma(\hat{y}_k)$ are in generic positions.*

In words, the center of the affine form $\hat{z}_k$ is taken as the center of the concretizations of the two affine forms $\hat{x}_k$ and $\hat{y}_k$. Then the coefficients of $\hat{z}_k$ over the noise symbols present in $\hat{x}_k$ and $\hat{y}_k$, are chosen to express the smallest dependency common to $\hat{x}_k$ and $\hat{y}_k$. Finally we add a new noise term $p^Z_{m+k,k}\eta_{m+k}$ to account for the uncertainty due to the join operation.

This operator has the advantages of presenting a simple and explicit formulation, a stable concretization with respect to the operands, and of being associative. And indeed, the solution with minimal interval concretization is particularly interesting for its stability when computing fixpoints in loops, as we will see in Theorem 20.

Example 8 *Take $X : (\hat{x}_1 = 1 + \varepsilon_1, \hat{x}_2 = \varepsilon_1)$ and $Y : (\hat{y}_1 = 2\varepsilon_1, \hat{y}_2 = \varepsilon_1)$. We have $\gamma(\hat{x}_1) = [0, 2]$ and $\gamma(\hat{y}_1) = [-2, 2]$, so that $\hat{x}_1$ and $\hat{y}_1$ are in generic positions. Then $Z = X \sqcup_1 Y : (\hat{z}_1 = \varepsilon_1 + \eta_1, \hat{z}_2 = \varepsilon_1)$ is a minimal upper bound of X and Y.*

Example 9 *Now take $X : (\hat{x}_1 = 1 + \varepsilon_1)$ and $Y : (\hat{y} = 4\varepsilon_1)$, this time $\gamma(\hat{x})$ and $\gamma(\hat{y})$ are not in generic positions. The join of Definition 10 still gives an upper bound: $Z : (\hat{z} = \varepsilon_1 + 3\eta_1)$, but it is not a minimal upper bound. For instance $W : (\hat{w} = 2\varepsilon_1 + 2\eta_1)$ is also an upper bound, and we have $W \subseteq Z$.*

Indeed, we can also characterize minimal upper bounds in the non generic case, but for simplicity's sake we omit them here. Moreover, these minimal upper bounds may be less interesting in many applications than the upper bound defined here, because they keep more noise symbols in the affine sets, so that the computational cost will be higher.

This one-dimensional join operator generalizes to constrained affine sets where noise symbols are abstracted using intervals, except that some attention must be paid to the relative positions of the ranges of noise symbols. Let us consider two constrained affine sets (X, Φ_X) and (Y, Φ_Y) where at least $p - 1$ components of X and Y are equal. A minimal upper bound (Z, Φ_Z) of (X, Φ_X) and (Y, Φ_Y) is obtained by joining the interval ranges of the noise symbols, $\Phi_Z = \Phi_X \sqcup \Phi_Y$, and using the property that a sufficient condition for (Z, Φ_Z) to be a minimal upper bound is to enforce a minimal concretization for the component k on which the join is performed, that is, on interval concretizations, $\gamma(\hat{z}_k) = \gamma(\hat{x}_k) \cup \gamma(\hat{y}_k)$, and then minimize the perturbation term among upper bounds with this concretization. This underlies the algorithm that is described in our 2010 paper [65] and K. Ghorbal's PhD thesis, and which computes a minimal upper bound in some cases, and else an upper bound with minimal concretization.

Component-wise join operator for affine sets

We can extend this one-dimensional operator in order to obtain a p-dimensional join operator, when all dimensions of the affine sets are joined.

Definition 11 (Component-wise join) *Let two affine sets X and Y in $\mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$. We define $Z = X \sqcup_C Y$ by: for all $k \in [1, p]$, $\hat{z}_k = \hat{x}_k \sqcup_1 \hat{y}_k$.*

Lemma 7 *Then $Z = X \sqcup_C Y$ is an upper bound of X and Y such that, for all $k \in [1, p]$, if $\hat{x}_k$ and $\hat{y}_k$ are in generic positions, then $\gamma(\hat{z}_k) = \gamma(\hat{x}_k) \sqcup \gamma(\hat{y}_k)$.*

Intuitively, we independently compute an upper bound on each variable x_k of the environment, using the one-dimensional join. The perturbation added to take into account the uncertainty due to the join is a diagonal block $p \times p$: a new noise symbol η_{m+k} is added for each variable x_k, and these new noise symbols are not shared. This join operator thus loses some relation that may exist between variables. However, the concretizations on each axis (i.e. the immediate concretization of all program variables) are optimal. and its cost of computation is $O((n + m)p)$ only.

Example 10 *Consider*

$$X = \begin{pmatrix} 1 + \varepsilon_1 + \varepsilon_3 \\ 1 + 2\varepsilon_2 + \varepsilon_3 \end{pmatrix} \qquad Y = \begin{pmatrix} -2 + \varepsilon_1 \\ -2 + 2\varepsilon_2 \end{pmatrix} \qquad Z = \begin{pmatrix} \varepsilon_1 + 2\eta_1 \\ 2\varepsilon_2 + 2\eta_2 \end{pmatrix}$$

$Z = X \sqcup_C Y$, given by Definition 11, is an upper bound for X and Y. But it is not a minimal upper bound:

$$W = \begin{pmatrix} \varepsilon_1 + 2\eta_1 \\ 2\varepsilon_2 + 2\eta_1 \end{pmatrix}$$

is also an upper bound, and it is such that $W \subseteq Z$.

We see that a better upper bound is obtained by using the fact that there is some dependency on the terms (η) expressing the uncertainty of control flow added on each variable. So that applying the join operator component-wise on each variable naturally loses some precision. This observation is at the origin of the improved (global) join operator of the next paragraph.

A join operation preserving affine input/output relations

We recently proposed [74] a new join operation, which, instead of computing the join independently on each component, computes and preserves the affine relations between variables and input noise symbols, that are common to the two affine sets joined. This new join operation uses the component-wise join introduced here on part of the variables, and then deduces a global upper bound using the relations. Let us consider a simple example:

Example 11 *Let us consider the following program:*

```
1  real x1 := [1,3];
2  real x2 := [1,3];
3  real x3;
4  if (random()) {
5     x1 = x1 + 2;
6     x2 = x2 + 2; }
7  x3 = x2 - x1;
```

Joining the two branches supposes to join the following two affine sets X and Y:

$$\begin{array}{rcl} \hat{x}_1 &=& 2+\varepsilon_1 \\ \hat{x}_2 &=& 2+\varepsilon_2 \end{array} \quad \textit{and} \quad \begin{array}{rcl} \hat{y}_1 &=& 4+\varepsilon_1 \\ \hat{y}_2 &=& 4+\varepsilon_2 \end{array}$$

If we apply the component-wise join, we obtain Z such that $\hat{z}_1 = 3+\varepsilon_1+\eta_1$ and $\hat{z}_2 = 3+\varepsilon_2+\eta_2$, where η_1 and η_2 are independent new noise symbols. And we do not capture the somehow disjunctive information, that either 2 or 4 is added to both x_1 and x_2, but that it can not be that 2 is added to x_1 and 4 is added to x_2. And $\hat{z}_3 = \hat{z}_2 - \hat{z}_1 = \varepsilon_2+\eta_2-\varepsilon_1-\eta_1 \in [-4,4]$

Now, it can be observed that there is a relation between the variables and noise symbols which is true for both branches joined: $\hat{x}_2 - \hat{x}_1 = \varepsilon_2 - \varepsilon_1$ in both branches. In order to get a global join operation on X and Y, we can thus use the one-dimensional join operator one one variable, say x_1, and naturally deduce the affine form for the second variable x_2 by this relation: this gives W such that $\hat{w}_1 = 3+\varepsilon_1+\eta_1$ as previously, and we deduce $\hat{w}_2 = 3+\varepsilon_2+\eta_1$. We thus obtain here a minimal upper bound of X and Y, and obtain the expected result $\hat{w}_3 = \hat{w}_2 - \hat{w}_1 = \varepsilon_2 - \varepsilon_1 \in [-2,2]$.

The projection on (x_1, x_2) of the concretization of X, Y, Z and W are represented Figure 2.3. The component-wise join gives the box $\gamma(Z)$ in which no relation are preserved between z_1 and z_2. Whereas the global join gives the zonotope $\gamma(W)$, which is here a minimal upper bound of $\gamma(X)$ and $\gamma(Y)$.

We now quickly formalize this join operation.

Definition 12 (Affine relations over affine sets) *Let an affine set X in $\mathcal{M}(n+1,p) \times \mathcal{M}(m,p)$. We call* affine relation *in X, an affine equation over the p program variables and the n noise symbols, that holds for*

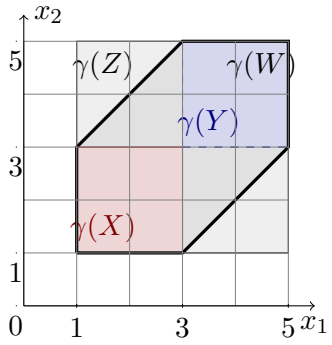


Figure 2.3: X, Y and results Z and W of the component-wise and global join operators

any values of the noise symbols, i.e. is given by $\alpha_1, \ldots, \alpha_p, \beta_0, \ldots, \beta_n \in \mathbb{R}$ *such that:*

$$\sum_{r=1}^{p} \alpha_r x_r = \beta_0 + \sum_{i=1}^{n} \beta_i \varepsilon_i \tag{2.5}$$

holds for any $\varepsilon = {}^t(\varepsilon_1, \ldots, \varepsilon_n) \in [-1,1]^n$ *and* $\eta = {}^t(\eta_1, \ldots, \eta_m) \in [-1,1]^m$ *and* $(x_1, \ldots, x_p) = {}^tC^X\varepsilon + {}^tP^X\eta$.

The set of affine relations defined in Equation 2.5, identified with vector $(\alpha_1, \ldots, \alpha_p, \beta_0, \ldots, \beta_n)$, is a sub-vector space of $\mathbb{R}^{p+n+1}$, hence generated by a finite number l of independent affine relations. This l is necessarily smaller or equal to p, since the n noise symbols are linearly independent, and there is a constant term.

Computing common affine relations An affine relation common to X and Y is an affine relation in X which is also an affine relation in Y. Consider we have k such affine relations, k being necessarily less or equal to p as previously stated:

$$\sum_{r=1}^{p} \alpha_{l,r} x_r = \beta_{l,0} + \sum_{i=1}^{n} \beta_{l,i} \varepsilon_i, \ \forall l \in \{1, \ldots, k\}. \tag{2.6}$$

When the $(x_r)_{1 \leq r \leq p}$ are defined by an affine set $X = (C^X, P^X)$, we can rewrite Equation 2.6:

$$\textstyle\sum_{r=1}^{p} \alpha_{l,r} x_r \ = \ \sum_{i=0}^{n} \left(\sum_{r=1}^{p} \alpha_{l,r} c_{i,r}\right) \varepsilon_i \ + \ \sum_{j=0}^{m} \left(\sum_{r=1}^{p} \alpha_{l,r} p_{j,r}\right) \eta_j, \ \forall l \in \{1, \ldots, k\}$$

These relations being true for every value of the noise symbols ε_i and η_j implies that for all $1 \leq l \leq k$, $0 \leq i \leq n$ and $1 \leq j \leq m$, we have:

$$\sum_{r=1}^{p} \alpha_{l,r} c_{i,r} = \beta_{l,i} \text{ and } \sum_{r=1}^{p} \alpha_{l,r} p_{j,r} = 0$$

Example 12 *Consider again the two affine sets joined in Example 11:*

$$\begin{array}{rcl}\hat{x}_1 &=& 2+\varepsilon_1\\ \hat{x}_2 &=& 2+\varepsilon_2\end{array} \quad \textit{and} \quad \begin{array}{rcl}\hat{y}_1 &=& 4+\varepsilon_1\\ \hat{y}_2 &=& 4+\varepsilon_2\end{array}$$

We want the affine relations common to X and Y, of the form: $\alpha_1 x_1 + \alpha_2 x_2 = \beta_0 + \beta_1\varepsilon_1 + \beta_2\varepsilon_2$ that hold for all $(\varepsilon_1, \varepsilon_2) \in [-1, 1]$. Substituting X and Y in this relation yields

$$\begin{array}{rcl}\beta_0 &=& 2\alpha_1 + 2\alpha_2 = 4\alpha_1 + 4\alpha_2\\ \beta_1 &=& \alpha_1\\ \beta_2 &=& \alpha_2\end{array}$$

The solutions can be parameterized by a $\lambda \in \mathbb{R}$, they are of the form $\beta_0 = 0$, $\alpha_1 = \beta_1 = \lambda$, $\alpha_2 = \beta_2 = -\lambda$. For instance we can choose $x_2 - x_1 = \varepsilon_2 - \varepsilon_1$.

The following lemma states that the affine input/output relations of Equation 2.6 are compatible with the functional order, which shows the necessity to preserve them:

Lemma 8 *Let $X = (C^X, P^X)$ and $Y = (C^Y, P^Y)$ be two affine sets such that*

1. *$X \leq Y$,*
2. *Y satisfies the k relations of Equation 2.6*

Then X satisfies these k relations.

Preserving these affine relations Up to a renumbering of the variables, we can always suppose the k independent common affine relations for X and Y are of the form:

$$x_i = R_i(x_{i+1}, \ldots, x_p, \varepsilon_1, \ldots, \varepsilon_n) \tag{2.7}$$

We thus choose among the p variables, $p - k$ variables on which we use the component-wise join. From there, using the k Equations 2.7, we can construct an upper bound for the p variables. We note $X_{>k}$ the projection of $X \in \mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$ on its last components $x_{k+1}, \ldots, x_p$, and now define the global join operator:

Definition 13 (Global join) *Let two affine sets X and Y in $\mathcal{M}(n+1, p) \times \mathcal{M}(m, p)$, which have in common k independent affine relations: for all $i \in [1, k]$, $x_i = R_i(x_{i+1}, \ldots, x_p, \varepsilon_1, \ldots, \varepsilon_n)$. We define $Z = X \sqcup_G Y$ by $Z_{>k} = X_{>k} \sqcup_C Y_{>k}$ and for all $i \in k, \ldots, 1$, $\hat{z}_i = R_i(z_{i+1}, \ldots, z_p, \varepsilon_1, \ldots, \varepsilon_n)$.*

Note that in Definition 13, the operations are ordered: first computation of $Z_{>k}$, then reconstruction of $\hat{z}_k$ to $\hat{z}_1$. This operation indeed defines an upper bound, which is a minimal upper bound in some cases:

Lemma 9 *$Z = X \sqcup_G Y$ is an upper bound of X and Y, and if $Z_{>k}$ is a minimal upper bound of $X_{>k}$ and $Y_{>k}$, then Z is a minimal upper bound of X and Y.*

2.2.4 Modular analysis

As already stated, affine sets provide linearized input-output relations, or functional abstraction, that are well suited for modular analyses. We recently proposed [89] a summary-based approach that is both natural and efficient, and still gives accurate results in realistic cases. The function summaries are pairs (I, O) of input and output zonotopes, parameterized by the same central noise symbols $\varepsilon_1, \ldots, \varepsilon_n$ representing the inputs of the program, so that both represent a function of these inputs. The pair also thus represents functions from $\gamma(I)$ to $\gamma(O)$. For a given input zonotope I, the output zonotope O is computed by applying the analysis with the zonotopic abstract domain, so that it is defined as a function of the input noise symbols $\varepsilon_1, \ldots, \varepsilon_n$, but also of the perturbation symbols I may contain. As I is usually obtained by joining several calling contexts, it contains some perturbation noise symbols corresponding to this join operation. The output O can then be instantiated to any particular calling context included in the input I of the summary, by substituting some of these perturbation noise symbols by their expression for the particular calling context.

Example 13 *Let us quickly sketch the summarizing process on a small example.*

```
1  real mult(real a, real b) {
2    return a*(b-2);
3  }
4
5  real compute(real x) {
6    real y1 = mult(x+1,x);
7    real y2 = mult(x,2*x);
8    return y2-y1;
9  }
10
11 compute([-1,1]);
```

Function `compute` *is called with* $\hat{x} = \varepsilon_1$ *(input in [-1,1]). We build a summary for function* `mult` *after its first call (*`y1 = mult(x+1, x);`*). Using the semantics on affine sets to abstract the body of the function, we get as summary the couple of affine sets* (I, O) *such that* $I = (\varepsilon_1 + 1, \varepsilon_1), O = (-1.5 - \varepsilon_1 + 0.5\eta_1)$*, where* I *abstracts the calling context and* O *the output.*

At the next call (`y1 = mult(x, 2*x);`*), we try to see if the previous summary can be used, that is if the calling context is contained in the input* I *of the summary: it is not the case as* $(\varepsilon_1, 2\varepsilon_1) \leq (\varepsilon_1 + 1, \varepsilon_1)$ *does not hold.*

We merge the two calling contexts with the join operator, and analyze again the body of the function: this gives a new (larger) summary for function `mult`*:* $I = (0.5 + \varepsilon_1 + 0.5\eta_2, 1.5\varepsilon_1 + 0.5\eta_3), O = (-\frac{1}{4} - \frac{5}{4}\varepsilon_1 - \eta_2 + \frac{1}{4}\eta_3 + \frac{9}{4}\eta_4)$.

Then, this new summary can be instantiated to the two calls (or any other call with calling context contained in affine set $I = (0.5 + \varepsilon_1 + 0.5\eta_2, 1.5\varepsilon_1 + 0.5\eta_3)$*). Without instantiation, the output value of the summary ranges in* $[-5, 4.5]$*. But the summary is a function defined over input* ε_1 *of the program, and over the symbols* η_2 *and* η_3 *that allow to express the inputs of function* `mult`*: we will thus get a tighter range for the output, as well as a function of input* ε_1*, by instantiating* η_2 *and* η_3 *and substituting them in the output of the summary. For instance, for the second call of function* `mult`*, with* $(\varepsilon_1, 2\varepsilon_1)$*, we identify* ε_1 *with* $0.5 + \varepsilon_1 + 0.5\eta_2$ *and* $2\varepsilon_1$ *with* $1.5\varepsilon_1 + 0.5\eta_3$*, and deduce* $\eta_2 = -1$ *and* $\eta_3 = \varepsilon_1$*, which yields for the output* $\frac{3}{4} - \varepsilon_1 + \frac{9}{4}\eta_4 \in [-\frac{5}{2}, 4]$*. Direct computation gives* $1 - 2\varepsilon_1 + \eta_4 \in [-2, 4]$*, which is just slightly tighter.*

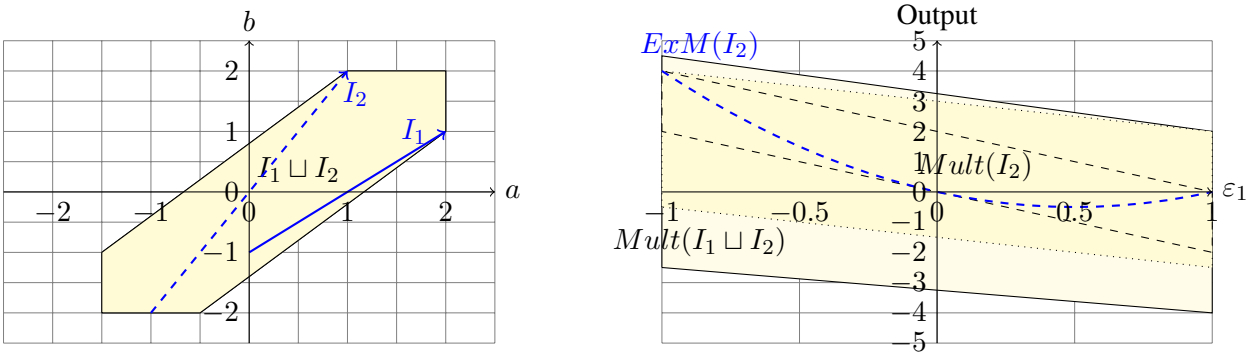


Figure 2.4: Summary and instantiation (left is input, right output)

We illustrate this in Figure 2.4: we represent on the left picture, the calling contexts (a, b) for the two calls (I_1 is first call, I_2 is second call), and the zonotopic concretization of the calling context after merge, $I_1 \sqcup I_2$. On the right part of the figure, the parabola is the exact results of the second call, $ExM(I_2)$. The dashed zonotope is the result of the abstraction with the semantics of affine sets of the second call, $Mult(I_2)$. The zonotope in plain lines is the output of the summary, $Mult(I_1 \sqcup I_2)$. The zonotope in dotted lines is the summary instantiated to the second call.

2.3 First experiments

The aim of this section is to study the behavior of the affine sets abstract domain, and compare it to other classic abstract domains. We restrict ourselves here to the analysis of programs in real number semantics, with small prototypes built for academic purposes. The extension to the study of finite precision computations, and the use of the static analyzer Fluctuat on real examples will be the subject of Chapter 5. Also, the study of fixpoint computations being left for Chapter 4, examples requiring fixpoint computations will be delayed until that chapter.

The presentation so far has assumed that the coefficients of affine sets are real numbers. In an implementation, however, floating-point coefficients are used. Using them without care would produce possibly unsound results: any rounding error would produce incorrect relationships between variables. A first idea to keep a sound over-approximation using finite precision numbers is to use affine sets with interval coefficients. The problem is that the width of these interval coefficients would grow with computations, we would suffer again from the defects of intervals. A better way to do this is to have floating-point coefficients. In order to keep a sound over-approximation, it is only needed to bound the error committed at each operation on these coefficients and account for this error by a new noise term. Practically, in order to reduce the number of noise symbols used, we group these errors with other approximation terms. But we could imagine to keep different kind of symbols, accounting respectively for the uncertainty on the outputs due to input uncertainty, abstract domain uncertainty, and uncertainty due to the finite precision implementation of the analysis.

K. Ghorbal implemented the (constrained) affine sets in the APRON library [103], as an abstract domain called Taylor1+ [64]. We now compare the performance of this domain, on unrolled linear and non linear it-

erative computations, with classic abstract domains such as boxes, octagons and polyhedra. No summarizing is being applied here. The upper bound operator implemented for these experiments is the component-wise one of Definition 11. More details can be found in [64, 65]. An implementation of the global join operator was also added to the Taylor1+ abstract domain, some results can be found in [74].

Linear schemes The first example we consider is the second order recursive filter:

$$x_i = 0.7e_i - 1.3e_{i-1} + 1.1e_{i-2} + 1.4x_{i-1} - 0.7x_{i-2},$$

where $x_0 = 0$, $x_1 = 0$, and the e_i are independent inputs between 0 and 1.

We here fully unroll the 2^{nd} order filter scheme to compute the abstract value at each iteration. Figure 2.5 compares accuracy and performance of Taylor1+ with three classic abstract domains provided in APRON: boxes, octagons and polyhedra (both PK [101] and PPL [140] implementations were tested). The polyhedra domain with exact arithmetic (using GMP) gives the exact bounds for the filter output. One can see that Taylor1+ wraps the exact range given by polyhedra (left figure) very closely, with great performance (right figure).

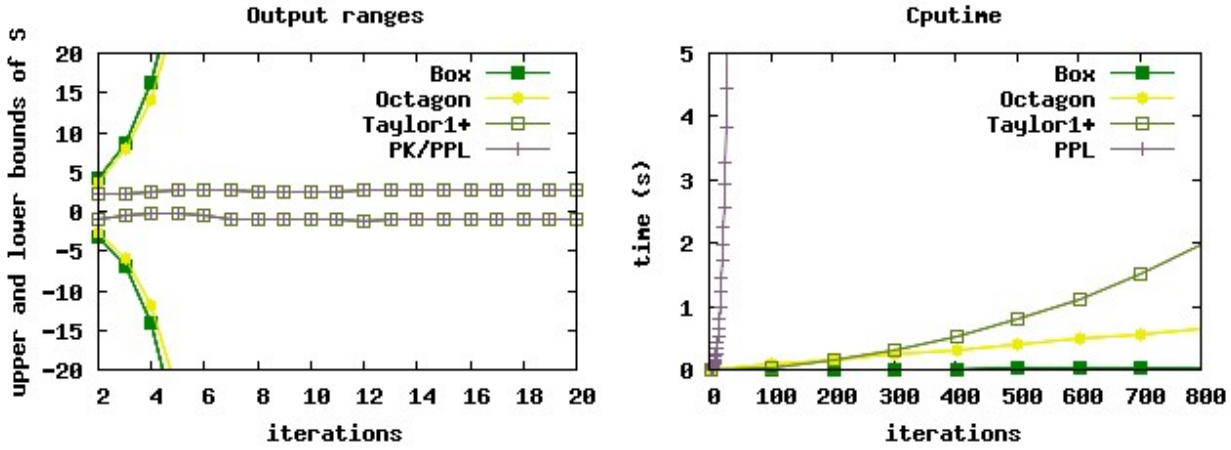


Figure 2.5: Unrolled scheme for the 2^{nd} order filter

Non-linear computations; test interpretation We now consider a toy example demonstrating the performance of the constrained version of the affine sets (constrained T1+ in its APRON implementation), where constraints on noise symbols are interpreted in intervals. We want to compute $g(g(x))$ on the range $x = [-2, 2]$, where

$$g(x) = \frac{\sqrt{x^2 - x + 0.5}}{\sqrt{x^2 + 0.5}}.$$

For that, we parameterize the program that computes $g(g(x))$ by a number of tests that subdivide the domain of the input variable (see Figure 2.6 left), in order to compare the cost and accuracy of the different domains when the size of the program grows. It can be noted (Figure 2.7 left) that the domain scales up well while giving here more accurate results (Figure 2.7 right) than the other domains. As a matter of fact, with

Figure 2.6: Implementation of g(g(x)) for x in [-2,2] (left) and plot of g(g(x)) (right)

```
1  g(x) = sqrt(x*x-x+0.5)/sqrt(x*x+0.5);
2  x = [-2,2];
3  /* for n subdivisions */
4  h = 4/n;
5  if (-x<=h-2)
6    y = g(x); z = g(y);
7  ...
8  else if (-x<=i*h-2) /* i in {2,...,n-1}*/
9    y = g(x); z = g(y);
10 ...
11 else
12   y = g(x); z = g(y);
```

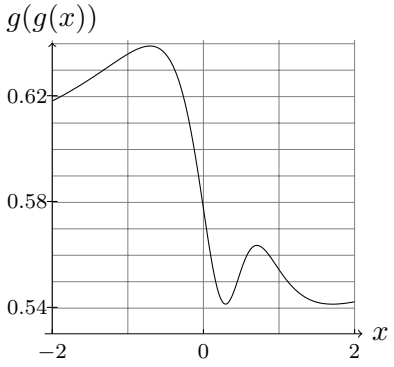
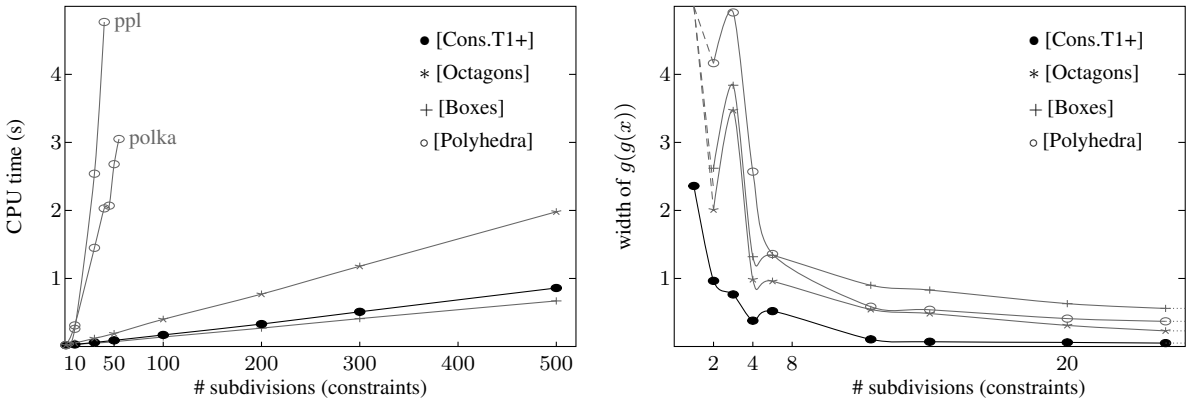


Figure 2.7: Comparing analysis time and results of the different APRON domains.



an interval domain for the noise symbols, all abstract transfer functions are linear or at worst quadratic in the number of noise symbols appearing in the affine forms. The fact that the results observed for 3 and 5 subdivisions (Figure 2.7 right) are less accurate respectively than those observed for 2 and 4 subdivisions, is related to the behavior of $g(g(x))$ on $[-2, 2]$ (see Figure 2.6 right): for example when a change of monotony appears near the center of a subdivision, the approximations will be less accurate than when it appears at the border.

Chapter 3

Extensions of the Zonotopic Approach

The zonotopic abstract domains of Chapter 2 lend themselves to several interesting extensions, still dealing with the abstraction of real-valued computations. These extensions no longer present zonotopic concretizations, but still use the ideas introduced in Chapter 2 in the zonotopic context. A first class of variations still relies on affine sets, but with coefficients that are no longer real numbers: we proposed in 2007 an under-approximating abstract domain based on affine sets with interval coefficients [82], which I will describe in Section 3.1. We are presently also considering extensions with zonotopic coefficients, that should define higher-order approximations. A second class of variations relies on a parametrization of variables values by affine sets with real coefficients, but the noise symbols no longer simply lie in intervals: we recently proposed [20, 21] to thus mix probabilistic and deterministic information by using p-boxes [52, 53] instead of interval noise symbols, this will be quickly described in Section 3.2. Finally, we are also considering to use vectors of noise symbols bounded in ℓ_2 norms for an ellipsoidal extension, or more generally in ℓ_p norm, as quickly discussed in the concluding chapter.

3.1 An under-approximating abstract domain

Most abstract interpretation numerical domains construct over-approximations of the range of program variables. Galois connections, which is the classic abstraction setting, naturally express over-approximations and we need dual Galois connections, or dual concretization-based frameworks, as developed by Schmidt [149, 148], to express under-approximations. But the main problem is that it is very difficult to compute good numerical under-approximations. Of course, testing or bounded model checking provide under-approximations, but what we want here is really some duals of classic over-approximating abstract domains.

We present here a variation of affine sets, using ideas from generalized interval arithmetic [70, 69], that allowed us to develop in 2007 [82] a first abstract interpretation domain for directly under-approximating the range of real values of program variables.

Such under-approximations, when combined with over-approximations, give an estimate of the quality of the result of a static analysis. But they can also be applied to statically find run-time errors that are bound to occur, from some given set of possible initial states: we will show how they can be used to generate test case scenarios that allow to minimize/maximize some variable.

$\boldsymbol{x} \times y$	$\boldsymbol{y} \in \mathcal{P}$	$\boldsymbol{y} \in \mathcal{Z}$	$\boldsymbol{y} \in -\mathcal{P}$	$y \in \text{dual}\mathcal{Z}$
$\boldsymbol{x} \in \mathcal{P}$	$[\underline{x}\underline{y}, \overline{x}\overline{y}]$	$[\overline{x}\underline{y}, \overline{x}\overline{y}]$	$[\overline{x}\underline{y}, \underline{x}\overline{y}]$	$[\underline{x}\underline{y}, \underline{x}\overline{y}]$
$\boldsymbol{x} \in \mathcal{Z}$	$[\underline{x}\overline{y}, \overline{x}\overline{y}]$	$[\min(\underline{x}\overline{y}, \overline{x}\underline{y}), \max(\underline{x}\underline{y}, \overline{x}\overline{y})]$	$[\overline{x}\underline{y}, \underline{x}\underline{y}]$	0
$\boldsymbol{x} \in -\mathcal{P}$	$[\underline{x}\overline{y}, \overline{x}\underline{y}]$	$[\underline{x}\overline{y}, \underline{x}\underline{y}]$	$[\overline{x}\overline{y}, \underline{x}\underline{y}]$	$[\overline{x}\overline{y}, \overline{x}\underline{y}]$
$\boldsymbol{x} \in \text{dual}\mathcal{Z}$	$[\underline{x}\underline{y}, \overline{x}\underline{y}]$	0	$[\overline{x}\overline{y}, \underline{x}\overline{y}]$	$[\max(\underline{x}\underline{y}, \overline{x}\overline{y}), \min(\underline{x}\overline{y}, \overline{x}\underline{y})]$

Table 3.1: Kaucher multiplication ([106])

3.1.1 Generalized affine sets for under-approximation

We first introduce the principles of generalized and Kaucher interval arithmetic [106] and their interpretation as modal intervals using quantifiers, and thus in some conditions as under-approximations [70, 69]. We then describe our contribution, which is to extend these ideas to define generalized affine forms that will be interpreted either as over or under-approximating forms for real values of variables.

Generalized intervals, modal intervals and Kaucher arithmetic Generalized intervals are intervals whose bounds are not ordered. The set of classic intervals is denoted by $\mathbb{IR} = \{[a, b],\ a \in \mathbb{R}, b \in \mathbb{R}, a \leq b\}$. The set of generalized intervals is denoted by $\mathbb{IK} = \{[a, b],\ a \in \mathbb{R}, b \in \mathbb{R}\}$. Intervals will be noted with bold letters.

For two real numbers a and b such that $a \leq b$, one can consider two generalized intervals, $[a, b]$, which is called proper, and $[b, a]$, which is called improper. For any a, b, we define the operations dual $[a, b] = [b, a]$ and pro $[a, b] = [\min(a, b), \max(a, b)]$.

The generalized intervals are partially ordered by inclusion which extends inclusion of classic intervals. Given two generalized intervals $\boldsymbol{x} = [\underline{x}, \overline{x}]$ and $\boldsymbol{y} = [\underline{y}, \overline{y}]$, the inclusion is defined by

$$\boldsymbol{x} \sqsubseteq \boldsymbol{y} \Leftrightarrow \underline{y} \leq \underline{x} \wedge \overline{x} \leq \overline{y}.$$

Kaucher addition extends addition on classic intervals :

$$\begin{aligned} \boldsymbol{x} + \boldsymbol{y} &= [\underline{x} + \underline{y}, \overline{x} + \overline{y}] \\ \boldsymbol{x} - \boldsymbol{y} &= [\underline{x} - \overline{y}, \overline{x} - \underline{y}] = \boldsymbol{x} + (-\boldsymbol{y}) \text{ where } -\boldsymbol{y} = [-\overline{y}, -\underline{y}]. \end{aligned}$$

We let $\mathcal{P} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \geq 0 \wedge \overline{x} \geq 0\}$, $-\mathcal{P} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \leq 0 \wedge \overline{x} \leq 0\}$, $\mathcal{Z} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \leq 0 \leq \overline{x}\}$, and dual $\mathcal{Z} = \{\boldsymbol{x} = [\underline{x}, \overline{x}],\ \underline{x} \geq 0 \geq \overline{x}\}$. Kaucher multiplication $\boldsymbol{x} \times \boldsymbol{y}$ is described in table 3.1. Kaucher division is defined for all y such that $0 \notin \text{pro}\, \boldsymbol{y}$ by $\boldsymbol{x}/\boldsymbol{y} = \boldsymbol{x} \times [1/\overline{y}, 1/\underline{y}]$. When restricted to proper intervals, these operations coincide with the classic interval operations.

Classic interval computations can be interpreted as quantified propositions. As an example, take f to be the function defined by $f(x) = x^2 - x$. Extended to interval arithmetic, its value on $x = [2, 3]$ is $f([2, 3]) = [2, 3]^2 - [2, 3] = [1, 7]$, which can be interpreted as the proposition

$$(\forall x \in [2, 3])\,(\exists z \in [1, 7])\,(f(x) = z).$$

Modal intervals extend classic intervals by coupling a quantifier to them. Extensions of modal intervals were recently proposed by Goldsztejn [70] in the framework of generalized intervals, and called AE extensions because universal quantifiers (All) always precede existential ones (Exist) in the interpretations. They give rise to a generalized interval arithmetic which coincides with Kaucher arithmetic. Let $f : \mathbb{R}^n \to \mathbb{R}$ a function in which each variable appears only once. Let $\boldsymbol{x} \in \mathbb{IK}^n$, which we can decompose in $\boldsymbol{x}_\mathcal{A} \in \mathbb{IR}^p$ and $\boldsymbol{x}_\mathcal{E} \in (\text{dual } \mathbb{IR})^q$ with $p+q=n$. We consider the problem of computing a quantifier Q_z and an interval $\boldsymbol{z} \in \mathbb{IK}$ such that

$$(\forall x_\mathcal{A} \in \boldsymbol{x}_\mathcal{A})\,(Q_z z \in \text{pro } \boldsymbol{z})\,(\exists x_\mathcal{E} \in \text{pro } \boldsymbol{x}_\mathcal{E})(f(x) = z). \tag{3.1}$$

In these expressions, if $\boldsymbol{z}$ is proper then $Q_z = \exists$, else $Q_z = \forall$. When all intervals are proper, we retrieve the interpretation of classic interval computation, which gives an over-approximation of the range of $f(x)$:

$$(\forall x \in \boldsymbol{x})\,(\exists z \in \boldsymbol{z})\,(f(x) = z).$$

And when all intervals are improper, we get an under-approximation :

$$(\forall z \in \text{pro } \boldsymbol{z})\,(\exists x \in \text{pro } \boldsymbol{x})\,(f(x) = z).$$

Affine forms for over and under-approximation We will now use these notions to compute over what we call generalized affine forms, where the $\boldsymbol{\alpha_i^x}$ coefficients are no longer real numbers but intervals :

$$\check{x} = \boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x}\varepsilon_1 + \ldots + \boldsymbol{\alpha_n^x}\varepsilon_n, \quad \text{with } \boldsymbol{\alpha_i^x} \in \mathbb{IR}. \tag{3.2}$$

The classic concretization $\gamma(\check{x})$ of $\check{x}$ is a proper interval obtained by the evaluation of expression (3.2) with proper intervals $\boldsymbol{\varepsilon_i} = [-1, 1]$ and classic interval arithmetic :

$$\gamma(\hat{x}) = \boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x}\boldsymbol{\varepsilon_1} + \ldots + \boldsymbol{\alpha_n^x}\boldsymbol{\varepsilon_n}.$$

We define the concretization $\delta(\check{x})$ of $\check{x}$ obtained by the evaluation of expression (3.2) with improper intervals $\boldsymbol{\varepsilon_i^*} = [1, -1]$ and Kaucher interval arithmetic :

$$\delta(\check{x}) = \boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_1^x}\boldsymbol{\varepsilon_1^*} + \ldots + \boldsymbol{\alpha_n^x}\boldsymbol{\varepsilon_n^*}.$$

Generalized affine forms are closed under affine operations. For affine forms $\check{x}$ and $\check{y}$, and real numbers λ_1 and λ_2, we have:

$$\lambda_1\check{x} + \check{y} + \lambda_2 = (\lambda_1\boldsymbol{\alpha_0^x} + \boldsymbol{\alpha_0^y} + \lambda_2) + (\lambda_1\boldsymbol{\alpha_1^x} + \boldsymbol{\alpha_1^y})\varepsilon_1 + \ldots + (\lambda_1\boldsymbol{\alpha_n^x} + \boldsymbol{\alpha_n^y})\varepsilon_n$$

We use for the under-approximation of the result of non affine arithmetic operations, an extension of the mean-value theorem to generalized intervals [69], which we extend to our generalized affine forms. We then derive an expression for the under-approximation of the multiplication. Note that we could also, in the same way, derive semantics for other arithmetic operations.

Mean-value theorem for generalized affine forms Suppose variables $x_1, \ldots, x_k$, are described as affine combinations of noise symbols $\varepsilon_1, \ldots, \varepsilon_n$. For a differentiable function $f : \mathbb{R}^k \to \mathbb{R}$, we write $f^\varepsilon : \mathbb{R}^n \to \mathbb{R}$ the function induced by f on ε_1 to ε_n. Suppose we have an over-approximation $\boldsymbol{\Delta_i}$ of the partial derivatives

$$\left\{ \frac{\partial f^\varepsilon}{\partial \varepsilon_i}(\varepsilon),\ \varepsilon \in [-1,1]^n \right\} \subseteq \boldsymbol{\Delta_i}. \tag{3.3}$$

Then

$$\tilde{f}^\varepsilon(\varepsilon_1, \ldots, \varepsilon_n) = f^\varepsilon(t_1, \ldots, t_n) + \sum_{i=1}^{n} \boldsymbol{\Delta_i}(\varepsilon_i - t_i), \tag{3.4}$$

where $(t_1, \ldots, t_n)$ is any point in $[-1,1]^n$, is interpretable in particular in the following sense :

- if $\tilde{f}^\varepsilon(\boldsymbol{\varepsilon_1^*}, \ldots, \boldsymbol{\varepsilon_n^*})$, computed with Kaucher arithmetic, is an improper interval, then pro $\tilde{f}^\varepsilon(\boldsymbol{\varepsilon_1^*}, \ldots, \boldsymbol{\varepsilon_n^*})$ is an under-approximation of $f^\varepsilon(\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_n})$.
- if $\tilde{f}^\varepsilon(\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_n})$ is a proper interval, then it is an over-approximation of $f^\varepsilon(\boldsymbol{\varepsilon_1}, \ldots, \boldsymbol{\varepsilon_n})$.

This theorem can be of course used when we take the ε_i in sub-ranges of $[-1,1]$, it will in fact be used in examples to improve the accuracy of the results.

We derive from this theorem a simple under-approximating model for the multiplication. We can easily prove by recurrence that, for all variables z whose real value is a linear function of noise symbols $\varepsilon_1, \ldots, \varepsilon_n$, the coefficient $\boldsymbol{\alpha_i^z}$ of the affine form obtained from our semantics is an over-approximation of $\frac{\partial z}{\partial \varepsilon_i}$. Then, for $f(x,y) = xy$, we can over-approximate in Equation (3.4)

$$\frac{\partial f^\varepsilon}{\partial \varepsilon_i}(x,y) = \frac{\partial x}{\partial \varepsilon_i} y + \frac{\partial y}{\partial \varepsilon_i} x$$

by

$$\boldsymbol{\Delta_i} = \boldsymbol{\alpha_i^x y} + \boldsymbol{\alpha_i^y x}, \tag{3.5}$$

for any over-approximation $\boldsymbol{x}$ and $\boldsymbol{y}$ of the values taken by x and y.

Example 14 *Let us consider $f(x) = x^2 - x$ when $x \in [2,3]$. The interval of values taken by $f(x)$ is $[2,6]$. Here, an under-approximation of $f(x)$ can not be computed directly by Kaucher arithmetic, since variable x does not appear only once in expression $f(x)$. An affine form for x is $x = 2.5 + 0.5\varepsilon_1$, and we deduce*

$$f^\varepsilon(\varepsilon_1) = (2.5 + 0.5\varepsilon_1)^2 - (2.5 + 0.5\varepsilon_1).$$

We bound the derivative by $\Delta_1 = 2 * 0.5 * (2.5 + 0.5\varepsilon_1) - 0.5 \subseteq [1.5, 2.5]$, and, using the mean-value theorem with $t_1 = 0$, we have

$$\tilde{f}^\varepsilon(\varepsilon_1) = 3.75 + [1.5, 2.5]\varepsilon_1.$$

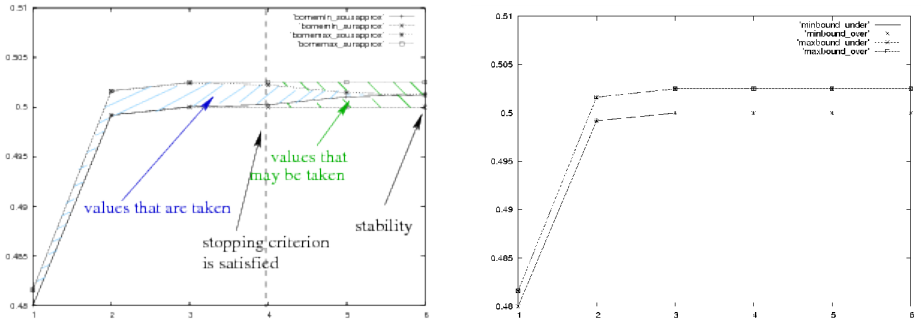
It can be interpreted as an under-approximation of the range of $f(x)$:

$$\check{\Gamma}(\tilde{f}^\varepsilon(\varepsilon_1)) = 3.75 + [1.5, 2.5][1, -1] = 3.75 + [1.5, -1.5] = [5.25, 4.25].$$

It can also be interpreted as an over-approximation :

$$\hat{\Gamma}(\tilde{f}^\varepsilon(\varepsilon_1)) = 3.75 + [1.5, 2.5][-1, 1] = 3.75 + [-2.5, 2.5] = [1.25, 6.25].$$

However, the range thus obtained for over-approximation is not as good as the one obtained by classic affine arithmetic, where $x^2 - x = [3.75, 4] + 2\varepsilon_1$, which gives the range $[1.75, 6]$.



no subdivision 32 subdivisions

Figure 3.1: Estimation of the maximum value of result x_i in the Newton algorithm.

Note that the generalized affine forms form a relational abstraction, as relations between variables are used to define the transfer functions. Still, we only characterized here under-approximations for the range of each variable taken independently, but not jointly. The join concretization is in particular no longer zonotopic, and characterizing it is one of our future objectives.

Order-theoretic considerations

We define the order on generalized affine sets by the following order applied component-wise on each component: $\check{x} \sqsubseteq \check{y}$ if and only if $\forall i \geq 0,\ \boldsymbol{\alpha}_{\boldsymbol{i}}^{\boldsymbol{x}} \sqsubseteq \boldsymbol{\alpha}_{\boldsymbol{i}}^{\boldsymbol{y}}$. If $\check{x} \sqsubseteq \check{y}$, then we have on the concretizations, $\delta(\check{x}) \sqsubseteq \delta(\check{y})$. In the case $\delta(\check{x})$ and $\delta(\check{y})$ are improper intervals, this is equivalent to pro $\delta(\check{y}) \sqsubseteq$ pro $\delta(\check{x})$. Inclusion $\check{x} \sqsubseteq \check{y}$ thus expresses that $\check{x}$ is a better under-approximation that $\check{y}$, as it concretizes to an interval whose proper range is larger than the one of $\check{y}$. Also, $\check{x}$ is a better over-approximation than $\check{y}$, as it concretizes to an interval whose proper range is included in the one of $\check{y}$: $\gamma(\check{x}) \sqsubseteq \gamma(\check{y})$.

This means in particular that we can get safe over and under approximations when using finite precision, by outward rounding on the $\boldsymbol{\alpha}_{\boldsymbol{i}}^{\boldsymbol{x}}$ coefficients.

3.1.2 Applications and experiments

We consider, for some given A, the (non-linear) iteration of the Newton algorithm $x_{i+1} = 2x_i - Ax_i^2$. If we take x_0 not too far away from the inverse of A, this iteration converges to the inverse of A. As an example, we take $A \in [1.99, 2.00]$, i.e. $\check{A} = 1.995 + .005\epsilon_1$, $x_0 = .4$ and ask to iterate this scheme until $|x_{i+1} - x_i| < 5 \times 10^{-6}$. We get the concretizations of lower and upper forms shown in the left part of figure 3.1. After 6 iterations, we get stable concretizations, for both lower and over-approximations of x_6 :

$$[0.5012531328, 0.5012531328] \sqsubseteq x_6 \sqsubseteq [0.499996841, 0.502512574],$$

$$[0, 0] \sqsubseteq |x_6 - x_5| \sqsubseteq [-3.17241289 \times 10^{-6}, 3.17241289 \times 10^{-6}].$$

And for the stopping criterion $|x_{i+1} - x_i| < 5 \times 10^{-6}$, using simultaneously the over- and under-approximation, we obtain in addition that the Newton algorithm terminates after exactly 4 iterations, which

allows us to refine the invariant:

$$[0.5002532337, 0.5022530319] \sqsubseteq x_4 \sqsubseteq [0.499996841, 0.502512574].$$

This is a general fact: the combination of under and over approximations gives in general a very powerful method to determine the invariants of a program.

We can also refine the results using subdivisions of the input A, that give way to as many independent computations relying on different affine forms for A. For instance, when subdividing in 32 sub-intervals, we get after 6 iterations a relative difference between the bounds given by the over- and under-approximations, of less than 9×10^{-6}; the results obtained are shown in the right part of figure 3.1.

Filters, perturbations and worst case scenario

In the sequel, in order not to get into complicated details, we suppose that we have a real arithmetic at hand (or arbitrary precision arithmetic). We consider again the order 2 recursive filter considered in Chapter 2:

$$S_i = 0.7E_i - 1.3E_{i-1} + 1.1E_{i-2} + 1.4S_{i-1} - 0.7S_{i-2},$$

where E_i are independent inputs between 0 and 1, so that $\check{E}_i = \hat{E}_i = \frac{1}{2} + \frac{1}{2}\epsilon_{i+1}$, and $S_0 = S_1 = 0$. We first consider the output S_i of this filter for a fixed number of unfoldings, e.g. $i = 99$. Using the over-approximating semantics of affine sets, we get

$$\hat{S}_{99} = \check{S}_{99} = 0.83 + 7.8 \times 10^{-9}\varepsilon_1 - 2.1 \times 10^{-8}\varepsilon_2 - 1.6 \times 10^{-8}\varepsilon_3 \ldots - 0.16\varepsilon_{99} + 0.35\varepsilon_{100},$$

whose concretization gives an exact enclosure of S_{99} :

$$[-1.0907188500, 2.7573854753] \sqsubseteq S_{99} \sqsubseteq [-1.0907188500, 2.7573854753].$$

Also, the affine form gives the sequence of inputs E_i that maximizes S_{99} : $E_i = 1$ if the corresponding coefficient multiplying ε_{i+1} is positive, $E_i = -1$ otherwise. Note that the exact enclosure actually converges to $S_\infty = [-1.09071884989..., 2.75738551656...]$, and therefore the signal (sequence of inputs of size 99) leading to the maximal value of S_{99} is a very good estimate of the signal leading to the maximal value of S_i, for any $i \geq 99$. This can be used to find bad-case scenarios of a program, and generalizes to linear recursive filters of any order.

We now perturb this linear scheme by adding a non-linear term $0.005E_iE_{i-1}$:

$$S_i = 0.7E_i - 1.3E_{i-1} + 1.1E_{i-2} + 1.4S_{i-1} - 0.7S_{i-2} + 0.005E_iE_{i-1}.$$

Again, we consider a fixed unfolding $i = 99$. Using over-approximation, we get

$$\begin{aligned}\hat{S}_{99} \quad = \quad & 0.837 + 7.81 \times 10^{-9}\varepsilon_1 - 2.09 \times 10^{-8}\varepsilon_2 \ldots - 0.157\varepsilon_{99} + 0.351\varepsilon_{100} \\ & +1.77 \times 10^{-11}\eta_1 - 2.66 \times 10^{-11}\eta_2 + \ldots + 0.00175\eta_{97} + 0.00125\eta_{98},\end{aligned}$$

in which terms from η_1 to η_{98} account for the over-approximation of non-linear computations. A sequence of inputs leading to a bad-case scenario is thus not given directly by the sign of the $\varepsilon_1, \ldots, \varepsilon_{100}$ as in the linear case: one can get a plausible worst-case scenario by choosing the $E_0, \ldots, E_{99}$ that maximize the

sub affine form containing only these ϵ_k, but one has no assurance that this might be even close to the real supremum of S_{99}. But the under-approximating form allows us to choose at least part of the inputs. Using the under-approximating semantics, we get :

$$\begin{aligned}\check{S}_{99} &= 0.837 + 7.81 \times 10^{-9}\varepsilon_1 - 2.09 \times 10^{-8}\varepsilon_2 + \ldots + [-0.0577, 0.0635]\varepsilon_{93} \\ &\quad +[0.0705, 0.138]\varepsilon_{94} + [0.185, 0.223]\varepsilon_{95} + [0.25, 0.271]\varepsilon_{96} + [0.222, 0.234]\varepsilon_{97} \\ &\quad +[0.081, 0.0876]\varepsilon_{98} + [-0.158, -0.155]\varepsilon_{99} + [0.35, 0.352]\varepsilon_{100}.\end{aligned}$$

This gives the following estimates for the real enclosure of S_{99}:

$$[-0.47655194955570, 2.1515519079] \sqsubseteq S_{99} \sqsubseteq [-1.10177396494, 2.77677392330].$$

As the interval coefficient $\boldsymbol{\alpha_k}$ is an over-approximation of the partial derivative of S_{99} with respect to ε_k, we know how to choose input E_{k-1} (E_{k-1} corresponds to ε_k) to maximize or minimize the output. We thus know that $E_{93} = 1$ $E_{94} = 1$, $E_{95} = 1$, $E_{96} = 1$, $E_{97} = 1$, $E_{98} = 0$ and $E_{99} = 1$ is the best depth 7 choice of inputs that will maximize S_{99}. For the other inputs, we use as a heuristic the $\varepsilon_0, \ldots, \varepsilon_{92}$ that maximize the over-approximating term, here this gives $S_{99} = 2.766383$. We thus found a scenario which is very close to the worst-case bound 2.77677392330. A one hour simulation on a 2GHz PC for 10^9 random sequences of 100 entries gives as estimate of the supremum 2.211418, far from our estimate in both time and precision.

3.2 Mixing P-boxes and Affine Arithmetic

In program analysis, a difficulty is to properly handle the inputs of the program. They are often not exactly known, but should rather be considered as being in a set of possible values. However, it is often the case that we have more information on the inputs than the mere fact that they belong to some set, such as a probability distribution that states their likelihood to be at a specific point within the set.

P-boxes [52, 53] and Dempster-Shafer structures [151] are widely used to propagate both probabilistic and non-deterministic information. However, arithmetic rules on these structures are classically defined only between structures representing either independent random variables, or variables with unknown dependency. This makes the usual p-box arithmetic not well adapted when many numerical computations occur between quantities (values of the program variables for instance) that cannot be considered independent, but whose dependency cannot be determined easily. The computations suffer from the drawbacks of classic interval arithmetic.

We thus proposed [20, 21] a new arithmetic that combines affine arithmetic to propagate the affine relations between random variables and p-box arithmetic to over-approximate the probability distribution resulting from a given operation. This allows us to both increase the precision of the result and decrease the computation time compared to standard p-box arithmetic.

3.2.1 Basics: Dempster-Shafer structures and p-boxes

An interval based Dempster-Shafer structure [151] (we will call it DS in short) is a finite set of closed intervals (named focal elements) associated with a probability. DS structures thus represent real variables whose value can be obtained by first probabilistically picking up an interval, and then non-deterministically picking up a value within this interval. In this article, we write a DS structure d as

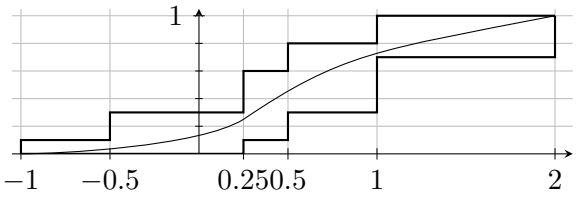


Figure 3.2: A p-box (thick lines) and a CDF (thin line) included in this p-box.

$d = \left\{\langle \boldsymbol{x_1}, w_1\rangle, \langle \boldsymbol{x_2}, w_2\rangle, \ldots, \langle \boldsymbol{x_n}, w_n\rangle\right\}$, where $\boldsymbol{x_i} \in \mathbb{IR}$ is a closed non-empty interval and $w_i \in]0,1]$ is the associated probability, with $\sum_{k=1}^{n} w_k = 1$.

A set of cumulative distribution functions can be represented by two non-decreasing functions $\underline{P}$ and $\overline{P}$ such that $\underline{P}$ is left-continuous, $\overline{P}$ is right-continuous and $\forall x,\ \underline{P}(x) \leq \overline{P}(x)$. Such a pair $\left[\underline{P}, \overline{P}\right]$ is called a p-box [52] and encloses all CDF P that satisfy $\forall x,\ \underline{P}(x) \leq P(x) \leq \overline{P}(x)$. We will only consider here *discrete* p-boxes, where $\underline{P}$ and $\overline{P}$ are step-functions. We refer the reader to Ferson et al. [52] for the correspondence between discrete p-boxes and DS structures; we note ζ the function that converts a DS to a p-box, and δ its inverse, that converts a discrete p-box to a DS.

Example 15 *Let* $d_1 = \left\{\langle[-1, 0.25], 0.1\rangle; \langle[-0.5, 0.5], 0.2\rangle; \langle[0.25, 1], 0.3\rangle; \langle[0.5, 1], 0.1\rangle; \langle[0.5, 2], 0.1\rangle; \langle[1, 2], 0.2\rangle\right\}$. *Then* $\left[\underline{P_2}, \overline{P_2}\right] = \zeta(d_1)$ *is as plotted in Figure 3.2.*

Many arithmetics were proposed for p-boxes and DS structures [10, 51, 56, 158], most of them are equivalent in term of precision [141]. Here, we briefly explain the method of Berleant et al. [10] which we used. Let two random variables X and Y represented by DS structures $d_X = \{\langle \boldsymbol{x_i}, w_i\rangle,\ i \in [1,n]\}$ and $d_Y = \{\left\langle \boldsymbol{y_j}, w'_j\right\rangle,\ j \in [1,m]\}$, and Z be the random variable such that $Z = X + Y$. We want to compute a DS that contains Z (the algorithms for other arithmetic operations are similar). We distinguish the case when X and Y are independent (in which case we denote the operation $\oplus$) and when they are not (in which case we denote the operation $+$).

Independent variables. If X and Y are independent random variables, then the DS for $Z = X \oplus Y$ is $d_Z = \{\langle \boldsymbol{z_{i,j}}, r_{i,j}\rangle,\ i \in [1,n],\ j \in [1,m]\}$ such that:

$$\forall i \in [1,n],\ j \in [1,m],\ \boldsymbol{z_{i,j}} = \boldsymbol{x_i} + \boldsymbol{y_j} \text{ and } r_{i,j} = w_i \times w'_j\ . \tag{3.6}$$

The number of focal elements within the DS structures clearly grows exponentially with the number of such operations. In order to keep the computation tractable, we bound the number of focal elements: of course, this introduces some over-approximation.

Variables with unknown dependency. If the random variables X and Y are not independent, the lower bound $\underline{F_Z}$ of the p-box $\left[\underline{F_Z}, \overline{F_Z}\right]$ for Z is obtained by solving the linear programming problem defined by Equation (3.7).

$$\begin{array}{rcll} \underline{F_Z}(z) & = & \textbf{maximize} & \sum_{\underline{x_i}+\underline{y_j} \leq z} r_{i,j} \\ & & \textbf{such that} & \forall i \in [1,n],\ \sum_{j=1}^{m} r_{i,j} = w_i \\ & & & \forall j \in [1,m],\ \sum_{i=1}^{n} r_{i,j} = w'_j \end{array} \tag{3.7}$$

The formula for $\overline{F_Z}$ is similar. We then define $d_Z = \delta(\left[\underline{F_Z}, \overline{F_Z}\right])$, where δ is the conversion function from p-boxes to DS.

3.2.2 Affine arithmetic with probabilistic noise symbols

We consider a set of uncertain quantities that will be propagated through a program. We will now express the linearized (deterministic) dependencies between these quantities using affine arithmetic, while we express the uncertainty on each variable, by associating a DS to each noise symbol of the affine forms. In this way, we add structure to DS arithmetic. As in classic affine sets, we will define two kinds of noise symbols: the ε_i that are considered independent, and the η_k that cannot be considered as independent from other noise symbols.

Definition 14 (Probabilistic affine form.) *We define a probabilistic affine form for variable x, on n independent noise symbols $(\varepsilon_1, \ldots, \varepsilon_n)$ and m noise symbols $(\eta_1, \ldots, \eta_m)$ with unknown dependency to the others, by a form*

$$\hat{x} = \alpha_0^x + \sum_{i=1}^{n} \alpha_i^x \varepsilon_i + \sum_{j=1}^{m} \beta_j^x \eta_j$$

together with n DS $(d_{\varepsilon_1}, \ldots, d_{\varepsilon_n})$ and m DS $(d_{\eta_1}, \ldots, d_{\eta_m})$ describing the possible random variables (of support $[-1, 1]$) for the noise symbols.

A probabilistic affine form $\hat{x}$ represents the set of all random variables X contained in the DS $\gamma(\hat{x})$, called concretization of $\hat{x}$:

Definition 15 (Concretization of affine forms as a DS) *Let $\hat{x}$ be a probabilistic affine form, its concretization as a DS is:*

$$\gamma(\hat{x}) = \alpha_0^x + \bigoplus_{j=1}^{n} \alpha_j^x d_{\varepsilon_j} + \sum_{k=1}^{m} \beta_k^x d_{\eta_k} \ ,$$

where $\bigoplus$ is the sum of independent DS, $\sum$ is the sum of DS with unknown dependency, and we note αd (resp. $\alpha + d$) where $\alpha \in \mathbb{R}$ and d is a DS $\{\langle \boldsymbol{x_i}, p_i \rangle \mid i = 1, \ldots, q\}$, the result of the multiplication (resp. addition) of a constant by a DS: $\{\langle \alpha \boldsymbol{x_i}, p_i \rangle \mid i = 1, \ldots, q\}$ (resp. $\{\langle \alpha + \boldsymbol{x_i}, p_i \rangle \mid i = 1, \ldots, q\}$).

The interest of affine forms is to be able to represent affine relations that hold between uncertain quantities. We still have this representation, except only *imprecise* affine relations hold, as can be shown in the example below.

Example 16 *Let $\hat{x_1} = 1 + \varepsilon_1 - \eta_1$, $\hat{x_2} = -\frac{1}{2}\varepsilon_1 + \frac{1}{4}\eta_1$, $d_{\varepsilon_1} = \{\langle [-1, 0], \frac{1}{2} \rangle, \langle [0, 1], \frac{1}{2} \rangle\}$,*
$d_{\eta_1} = \{\langle [-\frac{1}{10}, 0], \frac{1}{2} \rangle, \langle [0, \frac{1}{10}], \frac{1}{2} \rangle\}$. Then $\hat{x_1} + 2\hat{x_2} = 1 - \frac{1}{2}\eta_1$, with $d = d_{x_1+2x_2} = \{\langle [\frac{19}{20}, 1], \frac{1}{2} \rangle, \langle [1, \frac{21}{20}], \frac{1}{2} \rangle\}$. Thus the lower probability that $x_1 + 2x_2 \leq \frac{21}{20}$ is 1, and the upper probability that $x_1 + 2x_2 < \frac{19}{20}$ is 0. But for instance, $x_2 + 2x_2 \leq 1$ has upper probability $\frac{1}{2}$ and lower probability 0 and is thus an imprecise relation.

Affine arithmetic operations are exactly the same as in the fully deterministic case. For non-linear operations, we can rely on the affine form calculus, except that we use the available calculus on DS to form a correct DS representing this non-linear part. In this computation, some noise symbols are independent from the others, and we use both independent or unknown dependency versions of the operations on DS.

Noting that the square of a DS $x = \{\langle \mathbf{x}_i, w_i \rangle \mid i = 1, \ldots, p\}$, is the DS $x^2 = \{\langle \mathbf{x}_i^2, w_i \rangle \mid i = 1, \ldots, p\}$, we define the probabilistic affine form $\hat{z} = \hat{x} \times \hat{y}$ as an affine form defined as for classic affine arithmetic, but

where the new symbol η_{m+1} is considered with unknown dependency to the other symbols (characterization of this dependency is left for future work), and has an associated DS $d_{\eta_{m+1}}$ defined by:

$$\begin{aligned} d_{\eta_{m+1}} &= \frac{1}{T}\Big(\sum_{i=1}^{n}\sum_{j=1}^{m}(\alpha_i^x\beta_j^y+\beta_j^x\alpha_i^y)d_{\epsilon_i}\times d_{\eta_j}+\sum_{i=1}^{n}\sum_{j>i}^{n}(\alpha_i^x\alpha_j^y+\alpha_j^x\alpha_i^y)d_{\epsilon_i}\otimes d_{\epsilon_j} \\ &\quad +\sum_{i=1}^{m}\sum_{j>i}^{m}(\beta_i^x\beta_j^y+\beta_j^x\beta_i^y)d_{\eta_i}\times d_{\eta_j}+\frac{1}{2}\Big(\bigoplus_{i=1}^{n}\alpha_i^x\alpha_i^y {d_{\epsilon_i}}^2+\sum_{j=1}^{m}\beta_j^x\beta_j^y {d_{\eta_j}}^2\Big)\Big) \end{aligned}$$

Let us remark that, although affine arithmetic usually increases the computation time compared to interval arithmetic, we here decrease the complexity of arithmetic operations by using affine forms. Actually, we delay most of the DS arithmetic to the concretization function, and we can use the arithmetic of independent DS structures instead of arithmetic with unknown dependency, which is very costly.

Modeling rounding errors In the same way rounding errors can be computed using affine sets, as presented in Chapter 5, they can also be modeled and computed by probabilistic affine forms. We give a quick hint here. We note $f(r^x)$ the nearest floating-point value to real value r^x, and $e(r^x)$ the associated rounding error $e(r^x) = f(r^x) - r^x$. With normalized floating-point numbers and rounding mode to the nearest, we can bound the error committed when rounding the real value r^x to $f(r^x)$ by

$$|f(r^x) - r^x| \leq \delta.2^i \text{ with } i,\ |r^x| \in]2^i, 2^{i+1}], \tag{3.8}$$

and δ is a constant that depends on the floating-point format.

The sum of the DS structures defining $\hat{r}^x$ is a DS structure d_{r^x}. To compute the error when rounding this real number, we thus compute the image by the error function of each focal element, and add the weights when theses images appear several times:

Definition 16 *Suppose that the real value of x is given by the DS structure $d_{r^x} = \left\{\left\langle \boldsymbol{x_k^r}, w_k^r\right\rangle,\ k = 1,\ldots,n\right\}$, then for all $k \in [1,n]$, we define i_k such that $max(|\underline{\boldsymbol{x_k^r}}|, |\overline{\boldsymbol{x_k^r}}|) \in]2^{i_k}, 2^{i_k+1}]$. The rounding error of x is defined by the DS*

$$d_{e^x} = R(\left\{\left\langle [-\delta 2^{i_k}, \delta 2^{i_k}], w_k^e\right\rangle,\ k = 1,\ldots,n\right\}),$$

where R is a reduction operator that bounds the number of focal elements.

Example Consider a given integer n and the sum y_p of p independent variables $x_i,\ i = 1,\ldots,p$, each of them defined in $[-1,1]$ by the DS with $2n$ non-overlapping elements of width $\frac{1}{n}$ and same weight. These variables are first rounded as a floating-point value before being added. The envelop of the rounding error on a variable x_i is given in Figure 3.3. Consider for instance $n = 5$. The focal elements on the real value are materialized on this figure by the grid. The image, by the rounding error function, of the focal elements of r_{x_i} yields as DS for the error

$$d_{e^{x_i}} = \left\{\left\langle\left[-\frac{\delta}{2},\frac{\delta}{2}\right], 3/5\right\rangle, \left\langle\left[-\frac{\delta}{4},\frac{\delta}{4}\right], 1/5\right\rangle, \left\langle\left[-\frac{\delta}{8},\frac{\delta}{8}\right], 1/5\right\rangle\right\}\ .$$

We compute the partial results $y_k = \sum_{0\leq i<k} x_i$ for k from 1 to p, and a new rounding error $e(r^{y_k})$ is associated with each of these partial results. Finally, the rounding result on y_p is thus $e_p^y = \sum_{0\leq i<p} e(r^{x_i}) +$

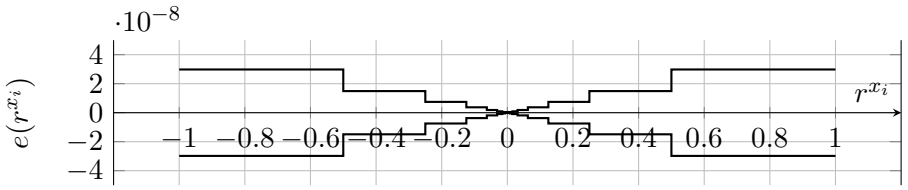


Figure 3.3: Envelop on the rounding error of $r^{x_i} \in [-1, 1]$ to a float (Example 3.2.2).

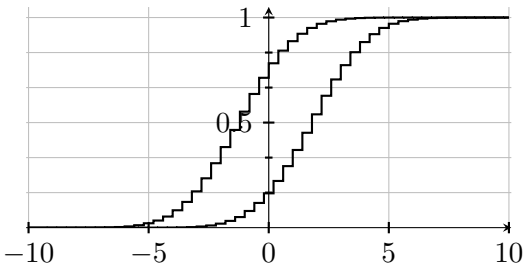


Figure 3.4: P-box for y_p

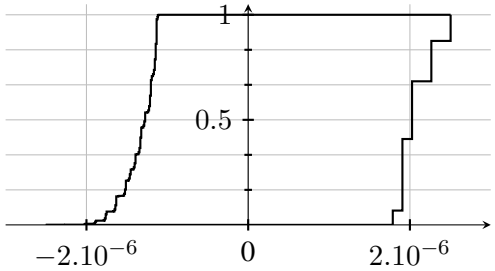


Figure 3.5: P-box for the error on y_p

$\sum_{1 \leq i \leq p} e(r^{y_k})$, where the DS for $e(r^{x_i})$ are all independent between one another, and $e(r^{y_k})$ has unknown dependency to $e(r^{x_j})$ and $e(r^{y_j})$ for all $j \leq k$. It is thus interesting to consider not only independence or dependency, but a generalized notion of dependency matrix between variables or noise symbols.

We present in Figures 3.4 and 3.5 the cumulative distribution functions of the DS obtained for $n = 10$ and $p = 10$. We see in Figure 3.4 that we bound indeed a quasi-Gaussian distribution (sum of independent uniform laws). Figure 3.5 is compatible with the stochastic modeling of errors as popularized in the CESTAC method [155]. The advantage here is that we have guaranteed lower and upper probabilities for rounding errors. Here we can read from Figure 3.5 that the probability that the error is within $[-1.2.10^{-6}, 1.2.10^{-6}]$ is $\frac{1}{2}$ while the probability that the error is in absolute value greater than 2.10^{-6} (which is about the maximal deterministic error one can get) is almost zero. An experiment with CADNA free allows us to estimate that the first 5 to 7 digits are correct in the calculation of the sum, and that the error is less, in absolute value, than 10^{-6} with a probability of a bit more than $\frac{1}{3}$, which is compatible with the results we get here.

Chapter 4

Loops and Fixpoint Computations

Fixpoint computation is a key point for static analysis by abstract interpretation: invariants of a program are obtained as fixpoints of the functional that describes the behavior of the program, and much time and accuracy of an analysis can be lost in loop invariant computation. This chapter, dedicated to the computation of fixpoints occurring in the static analysis of numerical programs, comprises two parts, that discuss two alternatives to solve the problem. The first section describes our use of Kleene-like iterations with zonotopic abstract domains [83, 64, 84]. The use of Kleene-like iterations in static analysis is classic, but we also try here to shortly discuss their convergence for given classes of programs, as one would do for any numerical algorithm. The second section presents an algorithm based on Newton-like iterations called policy iterations, introduced by Howard in 1960 to solve stochastic control problems [100]. We proposed to use policy iterations in the context of static analysis in 2005 [37]. In this article, we demonstrated the interest of this fixpoint iteration used with the abstract domain of intervals, and I will describe it here also in this context. However, it has since been applied for relational abstract domains [59, 2, 60, 61, 62], and we hope in the future to apply it for zonotopic abstract domains.

4.1 Kleene-like iteration schemes

We first briefly recall the classic computation of fixpoints in complete partial orders, based on Kleene's iteration, with widening and narrowing refinements [39], then specialize it to our context of affine sets.

4.1.1 Kleene's iteration sequence, widening and narrowing

We say that a self-map F of a complete partial order[1] $(\mathcal{L}, \leq)$ is monotonic if $x \leq y \Rightarrow F(x) \leq F(y)$. And it is also continuous if and only if for all increasing chains $l_1 \leq l_2 \leq \ldots \leq l_n \leq \ldots$ of $\mathcal{L}$, we have

$$\bigcup_{n \in \boldsymbol{N}} F(l_n) = F(\bigcup_{n \in \boldsymbol{N}} l_n).$$

The least fixpoint of such a monotonic and continuous F can be obtained by computing the sequence: $x^0 = \bot$, $x^{n+1} = F(x^n)$ for $n \geq 0$. If the sequence becomes stationary, i.e., if $x^m = x^{m+1}$ for some m,

[1] a partial order for which every increasing chain has a least upper bound

the limit x^m is the least fixpoint of F. Of course, this procedure may be inefficient, and it may not even terminate in the case of lattices of infinite height, such as the simple interval lattice. For this computation to become tractable, widening and narrowing operators have been introduced [39]. Widening operators are binary operators ∇ on $\mathcal{L}$ which ensure that any finite Kleene iteration $x^0 = \bot$, $x^1 = F(x^0)$, ..., $x^{k+1} = F(x^k)$, followed by an iteration of the form $x^{n+1} = x^n \nabla F(x^n)$, for $n > k$, yields an ultimately stationary sequence, whose limit x^m is a post fixpoint of F, i.e. a point x such that $x \geq F(x)$. Narrowing operators are binary operators Δ on $\mathcal{L}$ which ensure that any sequence $x^{n+1} = x^n \Delta F(x^n)$, for $n > m$, initialized with the above post fixpoint x^m, is eventually stationary, and converges to a fixpoint.

```
void main() {
  int x=0;          // 1
  while (x<100) {   // 2
    x=x+1;          // 3
  }                 // 4
}
```

$$\begin{array}{rcl} x_1 &=& [0,0] \\ x_2 &=&]-\infty, 99] \cap (x_1 \cup x_3) \\ x_3 &=& x_2 + [1,1] \\ x_4 &=& [100, +\infty[\cap (x_1 \cup x_3) \end{array}$$

Figure 4.1: A simple integer loop and its semantic equations

Consider the program at the left of Figure 4.1. The corresponding semantic equations in the lattice of intervals are given at the right of the figure. The intervals $x_1, \ldots, x_4$ correspond to the control points $1, \ldots, 4$ indicated as comments in the C code. The standard Kleene iteration sequence is eventually constant after 100 iterations, reaching the least fixpoint. This fixpoint can be obtained in a faster way by using the classic widening and narrowing operators [39]:

$$[a,b] \nabla [c,d] = [e,f] \text{ with } e = \begin{cases} a & \text{if } a \leq c \\ -\infty & \text{otherwise} \end{cases} \text{ and } f = \begin{cases} b & \text{if } d \leq b \\ \infty & \text{otherwise,} \end{cases}$$

$$[a,b] \Delta [c,d] = [e,f] \text{ with } e = \begin{cases} c & \text{if } a = -\infty \\ a & \text{otherwise} \end{cases} \text{ and } f = \begin{cases} d & \text{if } b = \infty \\ b & \text{otherwise.} \end{cases}$$

For instance, for $k = 10$, the iteration sequence using widening and narrowing takes 12 iterations and reaches the least fixpoint of f: $\forall k \in \{1, \ldots, 9\}, (x_2^k, x_3^k, x_4^k) = ([0, k-1], [1, k], \bot)$, then after widening, $(x_2^{10}, x_3^{10}, x_4^{10}) = ([0, \infty[, [1, \infty[, [100, \infty[)$, and finally, after narrowing, $(x_2^{11}, x_3^{11}, x_4^{11}) = ([0, 99], [1, 100], [100, 100])$.

4.1.2 Kleene-like iteration schemes for affine sets

In the case of affine sets, which do not form a complete partial order, we still use Kleene-like iterations, but with a final widening to intervals (Definition 17). We then propose some variations to improve the (post) fixpoint that is computed.

First variations on the Kleene iteration

We proved [84] that if S is a bounded and countable directed set of affine sets X in $\mathcal{M}(n+1, p) \times \mathcal{M}(m_i, p)$ - we thus suppose we have a bounded number of noise symbols - then there exists a minimal upper bound

for S. As we have only this form of bounded completeness, and not unconditional completeness, our iteration schemes will be parameterized by a large interval I: if the iterates remain bounded, the scheme will eventually converge, else when the current iterate leaves I^p, that is when the range of values taken by at least one of the p variables of the affine sets is not included in the interval I, we end iteration by $\top$. This gives the following iteration scheme:

Definition 17 (Bounded Kleene iteration) *Given an upper-bound operator $\sqcup$, the $\sqcup$-iteration scheme for a continuous and monotonic functional F on affine sets, is as follows:*

- *Start with $X^0 = F(\bot)$*
- *Then iterate: $X^{u+1} = X^u \sqcup F(X^u)$ starting with $u = 0$*
  - *if $\gamma_{lin}(P^{X^{u+1}}) \subseteq \gamma_{lin}(P^{X^u})$ then stop with X^{u+1}*
  - *if $\gamma_{lin}(P^{X^{u+1}}) \not\subseteq I^p$, then end with $\top$*

In this $\sqcup$-iteration scheme, $\gamma_{lin}(P^{X^{u+1}}) \subseteq \gamma_{lin}(P^{X^u})$ can be checked by solving at worse $O(m^2)$ linear programs, and guarantees that X^{u+1} is a post fixpoint of F: combining $X^u \leq X^{u+1}$ with stopping criterion $\gamma_{lin}(P^{X^{u+1}}) \subseteq \gamma_{lin}(P^{X^u})$ ensures that $X^{u+1} \sim X^u$. In practice, we can first use the simpler $O(mp)$ time test: $X_k^{u+1} \leq X_k^u \ \forall k = 1, \ldots, p$. It is only when this test is true that we compute the more costly test $\gamma_{lin}(P^{X^{u+1}}) \subseteq \gamma_{lin}(P^{X^u})$.

But we can also even completely avoid computing this costly test when we use a slightly modified Kleene iteration, with the particular join $\sqcup_C$ operator of Definition 11:

Definition 18 (Bounded component-wise Kleene iteration) *Given the component-wise upper bound operator $\sqcup_C$, we define the $\sqcup_C$-iteration scheme for a continuous and monotonic functional F on affine sets as follows:*

- *Start with $X^0 = F(\bot)$*
- *Then iterate: $X^{u+1} = X^0 \sqcup_C F(X^u)$ starting with $u = 0$*
  - *if $X_k^{u+1} \leq X_k^u$ for all $k = 1, \ldots, p$, then stop with X^{u+1}*
  - *if $\gamma_{lin}(P^{X^{u+1}}) \not\subseteq I^p$, then end with $\top$,*

This scheme converges to a post fixpoint of F: at each iteration, the current iterate is joined with the initial value X^0, so that the noise symbols introduced in the loop are lost, using the join operator of Definition 11. Lemma 5 thus applies, and we can decide the convergence on the affine sets from the convergence on each component independently.

Initial and cyclic unfolding

A loop often takes a few iterations between entering its a permanent behavior: it is thus quite classic to apply an initial unfolding of a loop before actually beginning the Kleene iteration. Moreover, the Kleene iteration on an iterative loop can be seen as a perturbation of the scheme that is indeed executed. However, joining states coming from different iterations can be quite a heavy perturbation, all the more if the join operator

is not perfect. In order to compute a tight post fixpoint of a loop abstracted by a functional F, it is often interesting to perform cyclic unfoldings of the loop, so as to locally compute fixpoints of functionals F^c, that is performing join operations only every c iterates. From there, we deduce the fixpoint of F:

Definition 19 (Kleene iteration with unfoldings) *Let i and c be any positive integers, $\sqcup$ any upper bound operator. The $(i, c, \sqcup)$-iteration scheme for a continuous and monotonic functional F on affine sets, is:*

- *Start with $X^0 = F(\bot)$,*
- *first unroll i times the functional, i.e. compute $X^1 = F^i(X^0)$,*
- *then iterate: $X^{u+1} = X^u \sqcup F^c(X^u)$ starting with $u = 1$,*
- *end when a fixpoint is reached or with $\top$ if $\gamma_{lin}(P^{X^{u+1}}) \not\subseteq I^p$.*

This unrolling scheme can also be applied to the modified iteration defined in Lemma 18. Its use will be exemplified afterwards.

Convergence acceleration and narrowing

The convergence of the schemes of Section 4.1.2 can be accelerated using extrapolation before possibly widening to $\top$: we can use for convergence acceleration any convenient upper bound operator, let us mention two accelerations that we widely use.

When considering numerical programs, the convergence of the fixpoint iteration is sometimes rather long, and even infinitely so, if operating with real numbers. The use of finite precision in the analysis is rather an opportunity in this context, to accelerate convergence by gradually reducing the precision (the size of the mantissa) of the numbers used to compute. This can be thought of as an improvement of the staged widening with thresholds [13], in the sense that thresholds are dynamically chosen along the iterations, depending on the current values of the iterates. Of course, this has to be achieved soundly, with outward rounding if on intervals, or accounting for this loss of precision by new noise terms if on coefficients of affine sets. We used this extrapolation with success in Fluctuat, this was facilitated by the use of the arbitrary precision library MPFR [55].

We combine this with the acceleration obtained by using as upper bound on affine sets, the affine set that keeps, for each variable, only the dependencies that are exactly the same (same coefficient for the same noise symbols) for the two forms. A new perturbation term ensures that the concretization of the variable is the result of the join of the two concretizations. We thus keep dependencies that remain true with the iterations. Of course, this can also be used to define a global acceleration, on the model of the global join operator.

Finally, let us note that loop invariants are elliptic for some interesting classes of control programs, such as order 2 linear recursive filters. It is still possible to abstract these invariants with zonotopes, but the number of faces and noise symbols will grow, and a widening as an enclosing ellipsoid would naturally be interesting. This is linked to the idea of extending affine sets so as to obtain ellipsoidal abstract domains, that we would like to develop in the future, as quickly discussed in the concluding chapter.

4.1.3 Convergence properties for the analysis of linear filters

In this section, we demonstrate the behavior of the Kleene-like fixpoint iteration on iterative schemes that are typical from control software, linear recursive filters. The iteration operates like replacing the numerical scheme by an abstract numerical scheme which has similar convergence properties, and fixpoints can thus be computed in finite time and in a guaranteed manner, accurately. However, joining states perturbs this scheme, so that it is not initially strongly contracting, the join must not be done too often, hence the use of the cyclic unfolding. We will also exemplify the use of extrapolation or widening techniques.

The results presented here are still very partial: they are for a restricted class of programs (linear in particular), and are simple but pessimistic, using old versions of the join operator. Still, they give a first idea of the potential of this abstract domain for fixpoint computation of some classic classes of programs, and they open the way for stronger results in the future.

A class of programs of interest

Let us consider the class of programs obtained by implementing the infinite iteration of a filter of order n with coefficients $a_1, \ldots, a_n, b_1, \ldots, b_{n+1}$ and a new input e between m and M at each iteration :

$$x_{k+n+1} = \sum_{i=1}^{n} a_i x_{k+i} + \sum_{j=1}^{n+1} b_j e_{k+j}, \tag{4.1}$$

starting with initial conditions $x_1, \ldots, x_n, x_{n+1}$. We can rewrite (4.1) as:

$$X_{k+1} = AX_k + BE_{k+1}, \forall k \geq 1, \tag{4.2}$$

with $X_k = {}^t(x_k, x_{k+1}, \ldots, x_{k+n})$, $E_{k+1} = {}^t(e_k, e_{k+1}, \ldots, e_{k+n+1})$,

$$A = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ 0 & 0 & \ldots & 0 \\ \ldots & & & \\ 0 & 0 & \ldots & 1 \\ a_1 & a_2 & \ldots & a_n \end{pmatrix} \quad B = \begin{pmatrix} 0 & 1 & \ldots & 0 \\ 0 & 0 & \ldots & 0 \\ \ldots & & & \\ 0 & 0 & \ldots & 1 \\ b_1 & b_2 & \ldots & b_{n+1} \end{pmatrix}$$

Now, of course, (4.2) has solution

$$X_k = A^{k-1}X_1 + \sum_{i=1}^{k} A^{k-i}BE_i,$$

where X_1 is the vector of initial conditions of this linear dynamical system. If A has eigenvalues (roots of $x^n - \sum_{i=1}^{n} a_i x^{i-1}$) of modulus strictly less than 1, then the term $A^{k-1}X_1$ will tend to zero when k tends toward infinity, whereas the partial sums $\sum_{i=1}^{k} A^{k-i}BE_i$ will tend towards a finite value.

Example 17 *Consider the following filter of order 2:*

$$x_i = 0.7e_i - 1.3e_{i-1} + 1.1e_{i-2} + 1.4x_{i-1} - 0.7x_{i-2},$$

where $x_0 = 0$, $x_1 = 0$, and the e_i are independent inputs between 0 and 1. A typical run of this algorithm, when $e_i = 0.5$ for all i, always stays positive, reaches at most 1.1649, and eventually converges towards 0.8333.

Convergence properties

The only abstract domains known to give accurate results, either deal with filters of order 2 [49] (with an extension on linear recursive filters of any order, given a decomposition in terms of second-order filters), or are specialized for digital filters [131]. Also, convex polyhedra [44] can converge on linear recursive filters, but this is not actually studied in the literature.

We proved [83] that our approach allows us to find good estimates for the real bounds of these general affine recurrences:

Theorem 20 *Suppose iteration (4.1) has bounded outputs, i.e. the (complex) roots of* $x^n - \sum_{i=0}^{n-1} a_{i+1}x^i$ *have modulus strictly less than 1. Then there exists* q *such that the* $(0, q, \nabla)$*-unrolling scheme of Definition 19 (with modified iteration introduced in Definition 18) converges towards a finite over-approximation of the output.*

This results relies on the fact that if A has eigenvalues strictly lower than 1, than we can find q such that A^q has coefficients such that the ℓ_1 norm on each row is strictly lower than 1. From there, even an interval join allows to prove convergence. As affine sets allow to compute exactly A^q for a given q, we can easily conclude.

We carry on with Example 17. Matrix A in this case is

$$A = \begin{pmatrix} 0 & 1 \\ -0.7 & 1.4 \end{pmatrix}$$

the ℓ_1 norm of the rows of A are bigger than 1. Iterating A, we see that:

$$A^5 = \begin{pmatrix} -0.5488 & 0.2156 \\ -0.15092 & -0.24696 \end{pmatrix}$$

is the first iterate of A with ℓ_1 norm of rows less than 1. We know by Theorem 20 that all $(0, k, \nabla)$-iteration schemes, with k greater or equal to 5, will converge to a finite upper approximation of the invariant, which we can estimate to [-1.30,2.82]. Here, the $(0, 5, \nabla)$-iteration scheme gives range [-1.63,3.30], while the $(0, 16, \nabla)$-iteration scheme gives range [-1.30,2.8244]. We thus observe that a large enough unfolding allows us to compute the fixpoint with all the accuracy we want.

The convergence to this invariant is *asymptotic* if computed in real numbers, meaning that we would need an infinite Kleene iteration to reach the invariant. However, when affine sets are implemented using finite precision, the limit is reached in a finite number of steps. For instance, in the case of the $(0, 16, \nabla)$-iteration scheme implemented with floating-point numbers, the fixpoint is reached after 18 iterations. And if we use convergence acceleration by loss of dependencies after one classic normal iteration step, the fixpoint is reached after 4 iterations. This time, convergence is reached in finite time, by construction, and no longer because of the use of finite precision.

We now consider the classic (non specialized) abstract domains that are implemented in the APRON library [103]: boxes, octagons and polyhedra. The unrolled loop already diverges with boxes and octagons, so does the fixpoint computation, naturally. For this 2^{nd} order filter, polyhedra give the least fixpoint for the second order filter, in a time comparable to the time taken by the affine sets abstract domain implemented

in this library, Taylor1+. But when we take a 8^{th} order linear recursive digital filter, polyhedra take an enormous amount of time (we aborted computation after 24 hours). In comparison, the efficiency of Taylor1+ is not really affected by the higher order (ratio to the analysis time of the 2nd order filter between 2 and less than 10, depending on the number of cyclic unfoldings). These results, among others, were presented in [64].

4.2 Policy iteration

One difficulty when using Kleene-like iterations to compute fixpoint, is to design good acceleration and widening/narrowing. Otherwise, the process will be inefficient or inaccurate, or both. When analyzing numerical programs, and in particular if we want refined properties such as tight value and rounding errors ranges, the problem becomes crucial.

We present here a new algorithm, based on policy iteration and not value iteration, for solving our fixpoint equations. Replacing the widening/narrowing process, it can improve both speed and precision of results over Kleene-like iterations. We describe it here in the case of abstraction by intervals, as we first introduced it [37]. But it is more general, and can be applied to a variety of abstract domains, provided that we can find a "selection principle", that is that our loop functionals can be expressed as infimums of a finite set of simpler functionals, on which fixpoint can be efficiently computed. Indeed it has been since applied to relational domains [59, 2], and other ways to use policy iterations have also been proposed since [60, 61, 62]. It is also integrated in the generic abstract domain library Apron (and actually demonstrated for intervals) [153], which can constitute a basis for future experiments with other abstract domains. First ideas to use policy iteration with our zonotopic domains will be mentioned, but are mainly future work.

Policy iteration was first introduced [100] to solve stochastic control problems with finite state and action space. The policy iteration generalizes Newton's algorithm to the equation $x = f(x)$, where f is monotonic, non-differentiable, and convex. It is experimentally efficient, although its complexity is still not well understood theoretically - we suspect the worst case is exponential, as an exponential lower bound was given for parity games by Friedman [57].

4.2.1 Lower selection

We assume that a map f is given as an infimum of a finite set $\mathcal{G}$ of "simpler" maps, and we wish to obtain a fixpoint of f from the fixpoints of the maps of $\mathcal{G}$:

Definition 21 (Lower selection) *We say that a set $\mathcal{G}$ of maps from a set X to a lattice $\mathcal{L}$* admits a lower selection *if for all $x \in X$, there exists a map $g \in \mathcal{G}$ such that $g(x) \leq h(x)$, for all $h \in \mathcal{G}$.*

Setting $f = \inf \mathcal{G}$, we see that $\mathcal{G}$ has a lower selection if and only if for all $x \in X$, we have $f(x) = g(x)$ for some $g \in \mathcal{G}$.

Let us consider the case of static analysis in the lattice of intervals. We define $\mathcal{F}$ the set of finitely generated functions on intervals, composed of expressions with arithmetic or order-theoretic (union, intersection) operations, and $\mathcal{F}_\cup$, the set of finitely generated functions of intervals, composed of expressions with arithmetic operations and unions, but no intersections.

The intersection of two intervals $I = [a, b]$ and $J = [c, d]$, and hence, of two expressions on intervals, is given by:

$$I \cap J = l(I, J) \cap r(I, J) \cap m(I, J) \cap m^{\text{op}}(I, J), \tag{4.3}$$

where $l(I,J) = I$ (l for left), $r(I,J) = J$ (r for right), $m(I,J) = [a,d]$ and $m^{\text{op}}(I,J) = [c,b]$ (m for merge). These operators define the four possible policies on intervals.

To $f \in \mathcal{F}$, we associate a family $\Pi(f)$ of functions of $\mathcal{F}_\cup$ obtained by replacing each occurrence of a term $I \cap J$ by $l(I,J),\ r(I,J),\ m(I,J)$ or $m^{\text{op}}(I,J)$. The set of policies $\Pi(f)$ admits a lower selection. In particular, $f = \inf \Pi(f)$.

4.2.2 Policy iteration

We are interested here in the smallest fixpoint, noted f^-, of f. The following result is a key ingredient of policy iteration:

Theorem 22 *Let $\mathcal{G}$ denote a family of monotonic self-maps of a complete lattice $\mathcal{L}$ with a lower selection, and let $f = \inf \mathcal{G}$. Then $f^- = \inf_{g \in \mathcal{G}} g^-$.*

We now state a general policy iteration algorithm. When it terminates, its output is a fixpoint of $f = \inf \mathcal{G}$.

Algorithm 1 (PI: Policy iteration in lattices) *1.* Initialization. *Set $k = 1$ and select any $g_1 \in \mathcal{G}$.*

2. Value determination. *Compute a fixpoint x^k of g_k.*

3. Compute $f(x^k)$. If $f(x^k) = x^k$, return x^k.

4. Policy improvement. *Take g_{k+1} such that $f(x^k) = g_{k+1}(x^k)$. Increment k and go to Step 2.*

The algorithm does terminate when at each step, the smallest fixpoint of g_k, $x^k = g_k^-$ is selected:

Theorem 23 *Assume that $\mathcal{L}$ is a complete lattice and that all the maps of $\mathcal{G}$ are monotonic. If at each step k, the smallest fixpoint $x^k = g_k^-$ of g_k is selected, then the number of iterations of Algorithm PI is bounded by the height of $\{g^- \mid g \in \mathcal{G}\}$.*

Any $x^k \in \mathcal{L}$ computed by Algorithm PI is a post fixpoint: $f(x^k) \leq x^k$. In static analysis of programs, such a x^k yields a valid, although suboptimal, information.

Although Algorithm PI may terminate with a non minimal fixpoint, it is often possible to check that the output of the algorithm is actually the smallest fixpoint and otherwise improve the results. We consider the special situation where f is such that $\|f(x) - f(y)\|_\infty \leq \|x - y\|_\infty$, for all $x, y \in \mathbb{R}^n$. We say that such maps f have *Property N*.

Theorem 24 *Assume that $\mathcal{G}$ is a finite set of monotone self-maps of $\overline{\mathbb{R}}^n$ that all have Property N, that $\mathcal{G}$ has a lower selection, and let $f = \inf \mathcal{G}$. If Algorithm PI terminates with a finite fixpoint $x^k = g_k^-$ such that there is only one $g \in \mathcal{G}$ such that $f(x^k) = g(x^k)$, then, x^k is the smallest finite fixpoint of f.*

When the policy g such that $f(x^k) = g(x^k)$ is not unique, we can still check whether x^k is the smallest finite fixpoint of f: we scan the set of maps $g \in \mathcal{G}$ such that $g(x^k) = f(x^k)$, until we find a fixpoint associated to g smaller than x^k, or all these maps g have been scanned.

4.2.3 Implementation and experiments for the lattice of intervals

A very simple static analyzer has been implemented. A policy is a table that associates to each intersection node in the semantic abstraction, a value modeling which policy is chosen among l, r, m or m^{op}, in Equation (4.3). There are a number of heuristics that one might choose concerning the initial policy, which should be a guess of the value of $G_1 \cap G_2$ in Equation (4.3).

We now discuss two typical examples, comparing the policy iteration algorithm with Kleene's iteration sequence with widening and narrowing, called Algorithm K here. We use a classic Kleene iterative solver (algorithm K$'$) for solving the reduced fixpoint equations (meaning, without intersections) for each policy. Using Algorithm K$'$ instead of a specific solver is an heuristics, since the convergence result requires that at every value determination step, the smallest fixpoint is computed. We decided to widen intervals, both in Algorithms K and K$'$, after ten standard Kleene iterations.

In the sequel, we put upper indices to indicate at which iteration the abstract value of a variable is shown. Lower indices are reserved to the control point number.

```
1  j=10; // 2
2  while (j >= i) { // 3
3    i = i+2; // 4
4    j = -1+j; // 5
5  } // 6
```

Semantic equations at control points 3 and 6 are

$$\begin{aligned}(i_3, j_3) &= (]-\infty, \max(j_2, j_5)] \cap (i_2 \cup i_5),\ \ [\min(i_2, i_5), +\infty[\cap (j_2 \cup j_5)) \\ (i_6, j_6) &= ([\min(j_2, j_5) + 1, +\infty[\cap (i_2 \cup i_5),\ \ ]-\infty, \max(i_2, i_5) - 1] \cap (j_2 \cup j_5))\end{aligned}$$

The algorithm starts with policy m^{op} for computation of i_3, m for j_3, m for i_6 and m^{op} for j_6. The first iteration using Algorithm K$'$ with this policy, finds the value $(i_6^1, j_6^1) = ([1, 12], [0, 11])$. But the value for j_6 applying again equation 6 using the previous result, is $[0, 10]$ instead of $[0, 11]$, meaning that the policy on equation 6 for j can be improved. The minimum (0) for j at equation 6 is reached as the minimum of the right argument of $\cap$. The maximum (10) for j at equation 6 is reached as the maximum of the right argument of $\cap$. Hence the new policy one has to choose for variable j in equation 6 is r. In one iteration of Algorithm K$'$, one finds the least fixpoint of the system of semantic equations, which is at line 6, $(i_6^2, j_6^2) = ([1, 12], [0, 10])$. This fixpoint is reached by several policies: we apply another policy iteration. This does not improve the result, the current fixpoint is the smallest one. Algorithm PI uses 2 policy iterations and 5 values iterations. Algorithm K takes ten iterations to reach the same result.

Algorithm PI can also be both faster and more precise than Algorithm K, as in the case of the following program:

```
6    k = 9; //2
7    j = -100; //3
8    while (i <= 100) //4 {
9      i = i + 1; //5
10     while (j < 20) //6
11       j = i+j; //7 //8
12     k = 4; //9
13     while (k <=3) //10
14       k = k+1; //11 //12
15   } //13
```

Algorithm PI reaches the following fixpoint, at control point 13, in 13 value iterations, $i = [101, 101]$, $j = [-100, 120]$ and $k = [4, 9]$ whereas Algorithm K only finds $i = [101, 101]$, $j = [-100, +\infty]$ and $k = [4, 9]$ in 62 iterations.

Indeed, after computing after some number of iterations ($i_4 = [0, 9], j_4 = [-100, 28], k_4 = [4, 9]$), Algorithm K applies widening and gets ($i_4 = [0, +\infty], j_4 = [-100, +\infty], k_4 = [4, 9]$).

From here on, there is no way, using further decreasing iterations, to find that j is finite (and less than 20) inside the outer while loop, since this depends on a relation between i and j that cannot be simulated using this iteration strategy.

We will discuss the case of policy iteration for zonotopic abstract domains in the concluding chapter.

Chapter 5

The Fluctuat Static Analyzer

Fluctuat is a static analyzer by abstract interpretation, relying on the general-purpose zonotopic abstract domains described in Chapter 2, and dedicated to the analysis of numerical properties of programs. Written in C++, it takes as input ANSI C or Ada programs, annotated with special assertions and directives.

Its primary objective is to prove that the computation performed by a program, using finite-precision arithmetic, conforms to what is expected by the programmer, that is the same computation in the idealized real-number semantics. And in the case the analyzer reports that rounding errors produce potentially large errors, it points out the main sources of uncertainties in the analyzed program. But it was gradually used for assessing also more functional properties, thanks to its zonotopic abstract domains. Sensitivity of variables to inputs and parameters deduced from the functional abstraction, can also be used to produce some test cases that aim at confirming a bad behavior, for the value or the error. We focus here on the analysis of programs implemented using integers and floating-point numbers, which is what we mainly studied; however the approach and the abstract domains developed are perfectly well suited to the handling of fixed-point numbers. And a simple abstract domain for the analysis of programs in fixed-point numbers is actually implemented in Fluctuat.

We will quickly outline the main features of Fluctuat, with a special emphasis on the specialization of the zonotopic abstract domains for the analysis of finite precision computations (Section 5.1), before describing its use and results on test cases (Sections 5.2 and 5.3). We will finally conclude by an overview of the key steps of the development of Fluctuat.

5.1 Analyzing finite precision computations

5.1.1 Concrete model

We first recall very briefly some basic properties of floating-point arithmetic, and refer the reader to surveys [68, 132] for more details. The IEEE 754 standard [1] specifies the way most processors handle finite-precision approximation of real numbers known as floating-point numbers. It specifies four rounding modes. When the rounding mode is fixed, it then uniquely specifies the result of rounding of a real number to a floating-point number. Let $\uparrow_\circ : \mathbb{R} \to \mathbb{F}$ be the function that returns the rounded value of a real number r. Whatever the rounding mode and the precision of the floating-point analyzed are, there exist two positive constants δ_r and δ_a (which value depend on the format of the floating-point numbers), such that the error

$r^x - \uparrow_\circ r^x$ when rounding a real number r^x to its floating-point representation $\uparrow_\circ r^x$ can be bounded by

$$|r^x - \uparrow_\circ r^x| \leq max(\delta_r | \uparrow_\circ r^x|, \delta_a). \tag{5.1}$$

For any arithmetic operator on real numbers $\diamond \in \{+, -, \times, /\}$, we note $\diamond_\mathbb{F}$ the corresponding operator on floating-point numbers, with rounding mode the one of the execution of the analyzed program. IEEE-754 standardizes these four operations, as well as the square root: the result of the floating-point operation is the same as if the operation were performed on the real numbers with the given inputs, then rounded. It also makes recommendation about some other operations, such as the trigonometric functions.

Fixed-point numbers are a finite approximation of real numbers with a fixed number of digits before and after the radix. The main differences with floating-point arithmetic are that the range of values is very limited, and that the absolute rounding error is bounded and not the relative error. As the only properties we will use in our abstractions, are a function giving the rounded value $\uparrow_\circ r^x$ of a real number r^x, and a range for the rounding error $r^x - \uparrow_\circ r^x$, all the work below naturally applies to the analysis of fixed-point arithmetic.

The semantics of machine integers does not introduce rounding errors, but there may still be error terms associated to integer variables. First, machine integers have finite range: we consider modular arithmetic, where adding one to the maximum value representable in a given data type gives the minimum value of this data type. The difference with natural integers will be an error term in our model (the "real" value being the natural integer). And of course, the propagation of an error term on a floating-point variable cast to an integer can lead to an error term on this integer variable.

The concrete model on which Fluctuat is based was first sketched in 2001 by Goubault [73], and made more explicit in 2002 by Martel [121]. Its underlying idea is to bound and describe the difference of behavior between the execution of a program in real numbers and in floating-point numbers. For that, the concrete value of a program variable x is a triplet (f^x, r^x, e^x), where $f^x \in \mathbb{F}$ is the value of the variable if the program is executed with a finite-precision semantics, r^x is the value of the same variable if the program is executed with a real numbers semantics, and e^x is the rounding error, that is $e^x = r^x - f^x$. Also, the error term e^x is decomposed along its provenance in the source code of the analyzed program, in order to point out the main sources of numerical discrepancy. The source of the main errors being most of the time localized at a few critical places, this should allow a user to either prove his program correct, or identify in a possibly large program the origin of a problem. Depending on the level of detail required, control points, blocks, or functions of a program can be annotated by a label ℓ, which will be used to identify the errors introduced during a computation. We note $\mathcal{L}$ the set of labels of the program, and $\mathcal{L}^+$ the set of words over these labels, then we express the error term as a sum

$$e^x = \bigoplus_{u \in \mathcal{L}^+} e_u, \tag{5.2}$$

where $e_u \in \mathbb{R}$ is the contribution to the global error of operations involved in word u. A word of length n thus identifies an order n error, which originates from several points in the program.

Several abstractions of this concrete model were successively implemented in Fluctuat, among which three are still used. We are not particularly interested in run-time errors nor exceptions: when an overflow or undefined behavior is encountered, our analysis simply issues a warning and reports $\top$ as a result.

The non relational abstraction is the natural abstraction, with intervals, of the values and the decomposition of error terms of the concrete model. It decomposes first order errors on their provenance, and abstracts all higher order errors by an interval. This was the first abstract domain implemented in Fluctuat, and it

is still occasionally used, when one wants a strict decomposition of the error, and the program analyzed is numerically not too complex. It is quickly described in Section 5.1.2.

The second domain that remains, abstracts by an affine form, the real value, each elementary first order error attached to one control point of the program, and the (agglomerated) higher order error terms. It thus preserves a strict decomposition of the error, and is very accurate. But it is quite costly, and we observed that the higher order term rather perturbs the convergence of fixpoint computations, so that it is better to group it with first order terms.

Finally, the third abstract domain, which is also the latest implemented, is the one which presents the best trade-off between cost and accuracy, and it is used more than 90 % of the time. It is quickly described in Section 5.1.3.

5.1.2 A non relational abstraction

The first natural idea to get an efficient abstract domain of our concrete model, is to abstract the values and errors with intervals, and to agglomerate all errors of order greater than one in a remaining interval term. This is what was first implemented in the Fluctuat static analyzer.

An abstract element x is a triplet

$$x = \left(\boldsymbol{f}^x,\ \boldsymbol{r}^x,\ \bigoplus_{l \in \mathcal{L}} \boldsymbol{e}_l^x \oplus \boldsymbol{e}_{ho}^x \right) \tag{5.3}$$

where interval $\boldsymbol{f}^x$ has floating-point bounds and abstracts the floating-point value, interval $\boldsymbol{r}^x$ has real bounds and abstracts the real value, $\boldsymbol{e}_l^x$ bounds the first-order contribution of control point l on the error between the real and the computed value of variable x, $\boldsymbol{e}_{ho}^x$ the sum of all higher-order contributions (most of the time negligible), and the error intervals have real bounds.

Rounding error abstraction For an interval of real numbers $\boldsymbol{r}^x = [\underline{r^x}, \overline{r^x}]$, we note $\uparrow_\circ \boldsymbol{r}^x = [\uparrow_\circ \underline{r^x}, \uparrow_\circ \overline{r^x}]$. Using bound (5.1), we can over-approximate the rounding error of a real value given in interval $\boldsymbol{r}^x$ to its finite precision representation, by the interval

$$\boldsymbol{e}(\boldsymbol{r}^x) = [-u^x, u^x] \cap (\boldsymbol{r}^x - \uparrow_\circ \boldsymbol{r}^x), \tag{5.4}$$

where $u^x = \max(\delta_r \max(|\uparrow_\circ \underline{r^x}|, |\uparrow_\circ \overline{r^x}|), \delta_a)$. This expresses the fact that we can compute the error as the intersection of the abstraction of bound given by (5.1), and the actual difference between the intervals bounding the real and the finite precision values. The right-hand side or the left-hand side will be more accurate in different cases, depending for instance on the width of the range of values abstracted.

Arithmetic operations Then, using this abstraction of the rounding error with the fact that the arithmetic operations on floating-point operations are correctly rounded (Section 5.1.1), we define the transfer function for expression $z = x \diamond^n y$ at label n. When $\diamond$ is the plus or minus operator, we have:

$$z \ = \ (\boldsymbol{f}^x \diamond_{\mathbb{F}} \boldsymbol{f}^y,\ \boldsymbol{r}^x \diamond \boldsymbol{r}^y,\ \bigoplus_{l \in \mathcal{L} \cup ho} (\boldsymbol{e}_l^x \diamond \boldsymbol{e}_l^y) \oplus \boldsymbol{e}(\boldsymbol{f}^x \diamond \boldsymbol{f}^y)),$$

where the existing errors on the operands are propagated, and a new error term $\boldsymbol{e}(\boldsymbol{f}^x \diamond \boldsymbol{f}^y)$ is associated to label (or control point) n. For the multiplication, we define:

$$z = (\boldsymbol{f}^x \times_{\mathbb{F}} \boldsymbol{f}^y,\ \boldsymbol{r}^x \times \boldsymbol{r}^y,\ \bigoplus_{l \in \mathcal{L} \cup ho} (\boldsymbol{f}^x \times \boldsymbol{e}_l^y + \boldsymbol{f}^y \times \boldsymbol{e}_l^x) \oplus \sum_{(l,k) \in (\mathcal{L} \cup ho)^2} \boldsymbol{e}_l^x \boldsymbol{e}_k^y \oplus\ \boldsymbol{e}(\boldsymbol{f}^x \times \boldsymbol{f}^y)).$$

The first order terms thus express the rounding errors and their propagation through computations. Higher-order errors appear in non affine arithmetic operations, and are non longer associated to a specific control point.

Cast to Integer The cast of a floating-point (or fixed-point) number to an integer, as long as it does not result in an overflow of the integer result, is not seen as an additional error, so that the real value r^x is also cast to an integer. Let us note $(int)(r^x)$ the result of casting a real variable to an integer, and $(int)(\boldsymbol{a})$ its extension to an interval $\boldsymbol{a}$, then we define $i = (int)x$ by:

$$i \ = \ \left((int)(\boldsymbol{f}^x),\ (int)(\boldsymbol{r}^x), \left(\sum_{l \in \mathcal{L} \cup ho} \boldsymbol{e}_l^x + [-1, 1] \right) \cap ((int)(\boldsymbol{r}^x) - (int)(\boldsymbol{f}^x) \right),$$

where the error is fully assigned to the label of the rounding operation: the sources of errors in previous computations are thus lost, we only keep track of the fact that rounding errors on floating-point computation can have an impact on an integer variable.

5.1.3 A zonotopic abstraction

We describe in this section, an abstraction of the triplet (f^x, r^x, e^x) relying on the zonotopic abstract domain of Chapter 2 for the real value r^x and the error e^x, from which the finite precision value of variables is bounded by the reduced product of $r^x - e^x$ with the intersection of a direct interval computation of f^x. More details can be found in [85].

An abstract value now consists of an interval and two affine forms, $x = (\boldsymbol{f}^x, \hat{r}^x, \hat{e}^x)$, and relies on two sets of noise symbols, the ε_i^r that model the uncertainty on the real value, and the ε_i^e that model the uncertainty on the error. The uncertainty on the real value also introduces an uncertainty on the error, which is partially modeled by the affine forms we consider:

$$\begin{aligned} \hat{r}^x &= r_0^x + \sum_i r_i^x\, \varepsilon_i^r \\ \hat{e}^x &= e_0^x + \sum_i e_i^x\, \varepsilon_i^r + \sum_l e_l^x\, \varepsilon_l^e \end{aligned}$$

In the error expression $\hat{e}^x$, $e_l^x\, \varepsilon_l^e$ expresses the uncertainty on the rounding error committed at point l of the program (its center being in e_0^x), and its propagation through further computations, while $e_i^x\, \varepsilon_i^r$ expresses the propagation of the uncertainty on value at point i, on the error term and allows to model dependency between errors and values.

Note that simpler variations of this abstract domain can be used to compute the finite precision value only, basically using only one affine form. Each operation creates a new noise symbol that is used either to over-approximate a non-affine operation, either to add a rounding error, or both.

We now define the transfer function for expression $z = x \diamond^n y$ at label n. When $\diamond$ is the plus or minus operator, we define:

$$\begin{aligned}
\hat{r}^z &= \hat{r}^x \diamond \hat{r}^y, \\
\hat{e}^z &= \hat{e}^x \diamond \hat{e}^y + new_{\varepsilon^e}(\boldsymbol{e}(\gamma(\hat{r}^z - \hat{e}^x \diamond \hat{e}^y) \cap (\boldsymbol{f}^x \diamond \boldsymbol{f}^y))), \\
\boldsymbol{f}^z &= (\boldsymbol{f}^x \diamond_{\mathbb{F}} \boldsymbol{f}^y) \cap \gamma((\hat{r}^z - \hat{e}^z))
\end{aligned}$$

where γ is the interval concretization on affine forms, interval function $\boldsymbol{e}$ is the interval abstraction of the rounding error defined in (5.4), and function new_{ε^e} creates a centered form with a fresh error noise symbol. All operations on affine forms, for real value or error, are those defined in Chapter 2. The new error term is obtained from the estimation of bounds for the real result of the operation on the floating-point operands, $\gamma(\hat{r}^z - \hat{e}^x \diamond \hat{e}^y) \cap (\boldsymbol{f}^x \diamond \boldsymbol{f}^y)$.

For the multiplication, we define:

$$\begin{aligned}
\hat{r}^z &= \hat{r}^x \hat{r}^y, \\
\hat{e}^z &= \hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y + new_{\varepsilon^e}(\boldsymbol{e}(\gamma(\hat{r}^z - (\hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y)) \cap (\boldsymbol{f}^x \boldsymbol{f}^y))), \\
\boldsymbol{f}^z &= (\boldsymbol{f}^x \boldsymbol{f}^y) \cap (\gamma(\hat{r}^z - \hat{e}^z)).
\end{aligned}$$

The propagation of the existing errors, obtained by expressing $\hat{r}^x \hat{r}^y - (\hat{r}^x - \hat{e}^x)(\hat{r}^y - \hat{e}^y)$, is computed with affine arithmetic by $\hat{r}^y \hat{e}^x + \hat{r}^x \hat{e}^y - \hat{e}^x \hat{e}^y$: this is how real noise symbols can appear in affine forms for the error. Note that here, there is no first-order / higher order error distinctions, and that there is only one affine form for the error: thus the origin of the errors is not as strictly decomposed as in the non relational semantics for instance. Still, the noise symbols are associated to control points, which allows us to keep information on the provenance of the uncertainties on the errors. The full decomposition was also implemented, but is very barely used because costly for no gain of accuracy, and with slower convergence in fixpoint iterations.

Some specific properties of floating-point numbers can be used to refine the estimation of the new error in some cases. For instance, the well-known Sterbenz Theorem [154] that says that is x and y are two floating-point numbers such that $\frac{y}{2} \leq x \leq 2y$, then the result of $x - y$ is a floating-point number, so that no new rounding error is added in this case. This allows to refine the subtraction by writing $\hat{e}^z = \hat{e}^x - \hat{e}^x$ when the floating-point values of x and y satisfy the hypotheses of Sterbenz Theorem.

Casts from floating-point value to integers As already stated, the truncation due to the cast is not seen as an error. So, the affine form for the real value is also cast to an integer, which results in a partial loss of relation, of amplitude bounded by 1, that expresses the quantization effect. We give here a simple version, that can be refined in some cases. If the error on x was not zero, a loss of relation also applies. For the floating-point value, the cast directly applied to the interval value will be more accurate than $r^i - e^i$. The cast $i = (int)x$ then writes

$$\begin{aligned}
\hat{r}^i &= \hat{r}^x + new_{\varepsilon^r}([-1, 1]) \\
\hat{e}^i &= \begin{cases} 0 & \text{if } e^x = 0 \\ new_{\varepsilon^e}([-1, 1]) & \text{if } \gamma(e^x) \in [-1, 1] \\ e^x + new_{\varepsilon^e}([-1, 1]) & \text{otherwise} \end{cases} \\
\boldsymbol{f}^i &= (int)\boldsymbol{f}^x
\end{aligned}$$

Affine operations and multiplication over integer variables can be directly extended from the floating-point version (same propagation of existing errors, no additional rounding error, but an additional error term if an overflow occurs). The integer division can be interpreted by a division over real numbers (error propagation with no additional error), followed by a cast to an integer.

Implementation in finite precision One advantage of the abstract domains relying on affine sets is to be easily soundly implemented when using finite precision numbers. The coefficients of the affine sets can be computed with finite precision: the computation is made sound by over-approximating the rounding error committed on these coefficients and agglomerating the resulting error in new noise terms. Keeping track of these particular noise terms is interesting as they quantify the overestimation in the analysis specifically due to the use of finite precision for the analysis.

5.1.4 Order-theoretic operations and unstable tests

The join and widening operators are defined component-wise on the real value, the floating-point value, and the error. This does not raise any particular problem.

But the test interpretation, and thus the meet operation, rely, for the time being, on the assumption that the control flows in the program are the same when considering the finite precision and real values of variables; in this case we say the test is stable. The test conditions are thus interpreted both on real and floating-point values. In the zonotopic abstract domains, the interpretation of tests yields constraints on the noise symbols, so that for the meet operation, we will join the constraints over the noise symbols generated by both $\hat{r}^x \cap \hat{r}^y$ and $\hat{r}^x - \hat{e}^x \cap \hat{r}^y - \hat{e}^y$. We still consider all possible control flows and join the values computed in each branch, so that even when an unstable test can occur, the bounds for the values will be sound. But the error bounds may be unsound, because we do not consider as an error term the difference of values obtained when the control flows diverge.

In practice, there are many potentially unstable tests, but most do not present a discontinuous behavior, in the sense that taking one branch or the other due to a rounding error does not dramatically change the result, and the error bounds will remain sound. Still, this has to be checked. For the time being, in case of an unstable test, a warning is issued by Fluctuat, indicating which test the user should check. We plan to propose in the near future a (dis)continuity analysis relying on the abstract domain of affine sets, that will add a new error term bounding this discontinuity error as an affine set of the existing value and error noise symbols, making the analysis sound even in presence of unstable tests.

5.2 Using Fluctuat

We chose in this document to describe Fluctuat through its use: first through a short description of the different classes of assertions and of the options and parameters of the analysis, then through examples that demonstrate their use. We refer to the user manual [88] for a detailed list of assertions and parameters, and their exact names.

The analysis consists in three phases, that can be iterated: the user first selects the source files to be analyzed, and, if useful, annotates them with the language of directives and assertions of Fluctuat. Then the parameters that are global to the analysis can be set via the user interface (the local parameters are rather set using the directives). Finally the analysis is performed and the results consulted via the user interface

(designed with QT). The user can then refine the analysis or ask for more specific task such as worst case generation.

Recently, an interactive version was developed, that works as an abstract debugger: it allows the user to better understand some results of the analysis, and refine them if needed, by changing some parameters or values during the analysis.

Let us now detail the three phases of the (non interactive) analysis:

5.2.1 Building the analysis project - directives

The directives added in the source code to be analyzed help tune the analysis locally, we will see their use in more details in the examples section. We do not want to be exhaustive here, but the main directives fall in one of the following classes:

Hypotheses on the environment of the program They can specify ranges on values (real or floating-point) of variables, ranges on errors on inputs and parameters, bounds on the variation of a variable between two consecutive calls. A further step has been taken in the direction of taking into account a model of a continuous environment in our analysis, using a guaranteed integration tool to bound the behavior of the continuous environment, see Section 5.3.6. Some special directives in the code then model the interaction with the environment via sensors and actuators.

Local partitioning and collecting strategies They are used to improve the accuracy of the analysis, and are of two kinds. Either a particular partitioning is needed, and known by the user: the user then builds this partitioning in the code with the assertions for the value on the variable to be partitioned, and specifies the analyzer it must collect all results by a special assertion. Or there is no particular partitioning that is expected to improve the accuracy, and we use local partitioning assertions, that specify, the scope of the subdivision, and a number of subdivisions or an accuracy requirements. In the first case, a regular subdivision will be used. In the second case, the analyzer will dynamically adapt the subdivision to the requirement.

Printing information If not specified otherwise, the analyzer prints bounds on values and errors on variables at the end of the analysis. It would be too costly indeed to print all information computed during the analysis. Printing directives thus allow to print value and error information on variables of interest during the analysis: this evolution can then be visualized on graphs with the user interface. Some other directives indicate to print relational information which is not given otherwise, such as sensitivity of an output to the inputs of the program: this in particular permits worst case generation, that will suggest scenarios to try to maximize the value or error of the given output.

As already mentioned, we progressively moved from a use where we only characterized the propagation of errors, to the analysis of more functional behavior, such as sensitivity analysis, or even proof that the program actually computes something at least close to what is expected in term of the real mathematical function. For the time being, the verification of a remarkable identity when available is done by adding variables and computation of that identity in the source code. But it could be achieved in the future using new directives.

5.2.2 Parameterizing the analysis

The options set via the user interface (see Figure 5.1 for the main parameters window), define the global parameterization of the analysis. We try to describe the main possibilities below:

Abstract semantics The different abstract domains presented in Section 5.1 can be chosen via the user interface of Fluctuat. They all rely on the arbitrary precision library MPFR [55], which provides correct rounding for all elementary and library mathematical functions: the precision (length p of the mantissa) of the numbers used can be specified by the user.

Each abstract domain can be used for static analysis (with the iteration strategies mentioned below) or symbolic execution: in the second case, the finite precision value of all variables given in an interval are taken to the center of the interval, and loops are fully unfolded. However, the behavior of the program analyzed around this floating-point scenario can still be studied by adding a perturbation as an error term.

Our alias and array analysis is based on a simple points-to analysis [111]. An abstract element is a graph of locations, where arcs represent the points-to relations, with a label (which we call a selector) indicating which dereferencing operation can be used. Arrays and structs are interpreted in two different ways: either all entries are considered to be separate variables or the same one (for which the union of all possible values is taken). Although these abstractions have proved sufficient for the time being, we may have to refine them in the future.

Iteration strategies Fluctuat implements all the Kleene-like iteration variations and acceleration techniques described in Section 4.1 of Chapter 4. The choice of initial and cyclic unfolding of loops, number of iterations before extrapolating the result can be parameterized in the user interface.

As already mentioned, computations in Fluctuat are performed with user-defined precision p, using the MPFR library. Before eventually using a widening to ensure termination of the analysis, we can use a convergence acceleration operator for the computation of fixpoint, obtained by the progressive reduction of the precision p of the floating-point used for the analysis (in general, the more precision is used for computation, the more iterations are needed for fixpoint convergence): the number of mantissa bits lost at each iteration can be specified by the user.

In the case of nested loops, a depth first strategy is chosen : at each iteration of the outer loop, the fixpoint of the inner loop is computed. In critical embedded systems, recursion is in general prohibited. Hence we chose to use a very simple inter-procedural domain, with static partitioning [104].

Finally, the user can choose in Fluctuat to reduce the level of correlation to be kept: different strategies are proposed to group the noise symbols. For example, correlations can be kept only between the last n iterations of a loop, where n is a parameter of the analyzer. Also, the user can choose to agglomerate, or not, the noise symbols introduced in a loop when getting out of it, the idea being that the correlations introduced inside the loop may be useful for fixpoint convergence inside the loop, but no longer outside the loop. The same thing holds when getting out of a function call.

Subdivisions and refinements The propagation of error bounds in operations in which most relational information are lost, mainly bit-wise operations on integers, is most of the time very inaccurate if on a large set of values: in this case, the user can specify to locally refine the computation of these error bounds, by subdivision of the input values of these integer operations.

Also, global subdivisions (for the whole program) of input variable ranges (in case we want very precise results on non-linear computations) can be specified via the user interface. On difficult cases, combined with symbolic executions, this can be used to have sample values and corresponding analysis results, in order to get a first picture of the behavior of the program, before proceeding to a full static analysis.

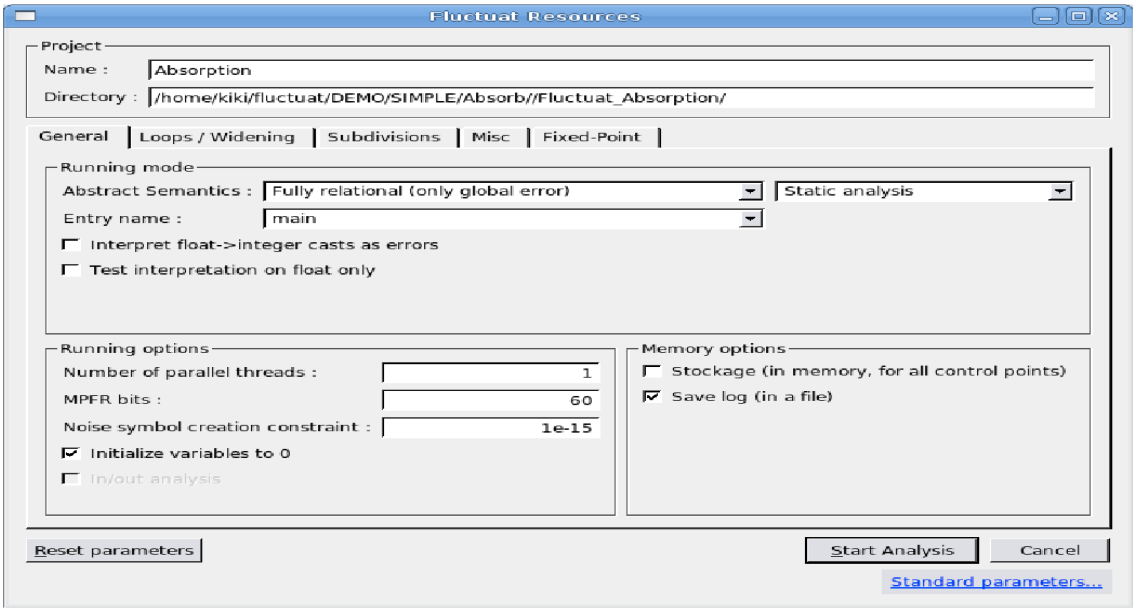


Figure 5.1: Parameters window

5.2.3 Exploitation of results

The graphical interface (see Figure 5.2 for the main window) gives information during and at the end of the analysis:

During the analysis During the analysis, the user has some knowledge about the status of the analysis: in case of loops, he knows in which loop the analysis is, and has an indication of the remaining time based on worst case hypotheses for loops, and number of remaining subdivisions in case of global subdivisions. Also, in case of subdivisions, he has access to the result of the subdivisions already analyzed. And, in any case, a pop-up window appears to warn the user that the analysis may be unsatisfying as soon as a variable is evaluated to top. Finally, the printing assertions can be used to watch the results during the analysis.

All this information allows the user to save some time: he has a first idea without waiting for the end of the analysis, of the time required for the analysis and the accuracy of results.

Warnings The tool warns about possible run-time errors or unspecified behaviors, and their source in the analyzed program. When possible, the program should be fixed for the tool not to issue such warnings, before looking more closely at results. Unstable tests, that is when the floating-point and the real control flows may be different, are also reported: the user must then make sure they have no dangerous impact,

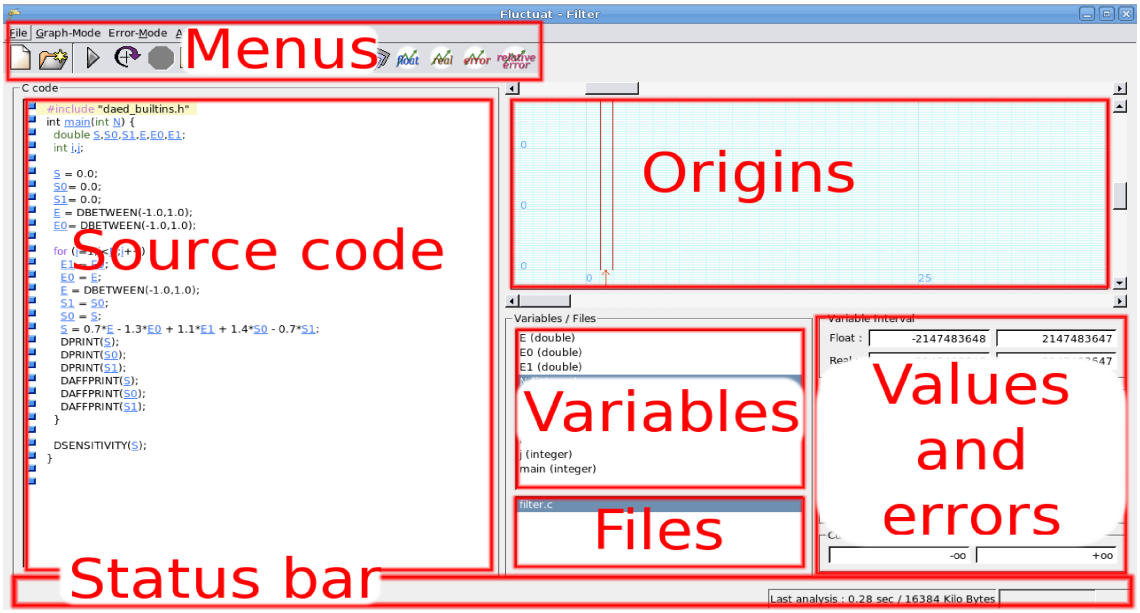


Figure 5.2: Main window

namely that they do not create discontinuities (as already mentioned, we intend to propose a new analysis to compute this information in the near future).

Identifying problems and refining results The interface presents graphically, at the end of the program and for each variable, bounds on the value and on elementary errors and their provenance in the source code: the user can thus quickly identify the main sources of errors. Sensitivity of a given output to inputs can also be printed, and we implemented heuristics to compute scenarios leading to results close to the error and value bounds. Finally, some help is provided for the analysis of portions of large programs: the code which is proved to be dead is indicated, as well as uninitialized variables.

The analyses may still yield abnormally large ranges or errors on output variables (the analyzer points to the main sources of errors). In this case, the user should switch to the symbolic execution mode of the tool, in order to make sure the analyzed program is not trivially unstable. More generally, the accuracy of the static analysis should be assessed through worst-case generation and symbolic execution with (possibly many) subdivisions. The static analysis parameters should then be tuned until results can be compared to that of symbolic execution.

5.3 Examples and case studies

Analysis of control software is the primary target of Fluctuat, at least for now. These software are usually developed in a model-based approach: most of the source code is generated automatically from high-level synchronous data-flow specifications and external basic clocks. And these basic blocks often implement approximations of mathematical functions. When we have a specification for the function, we can in Fluctuat evaluate the accuracy of the approximation, in real and in finite precision. Bounds on the model errors,

which give a functional proof that the program in real numbers indeed computes something close to the mathematical function, are obtained by bounding the real value of the result minus the mathematical function considered. While bounds on the additional error due to finite precision computation are obtained with the error bounds. This gives the user tools to adapt well the model error to the computational error: for instance, there is no need to use double precision numbers in an elementary function if the model error is very high; or conversely, there is no need to use a very fine approximation model if the finite precision computation introduces errors that are larger than this model error. On the contrary, it can be counter-productive and dangerous in some cases, as it may prevent the floating-point program to terminate in all cases: see for instance the square root approximation of section 5.3.5.

No real embedded code is shown here, but all the examples we will develop are strongly inspired by the basic blocks we have encountered in our case studies: indeed, such elementary functions analyses were or are conducted for different groups such as Airbus, Sagem, Continental, Thales and IRSN (the French Institute for Radioprotection and Nuclear Safety), and were a the the origin of several publications [79, 87, 46, 19]. We have also studied longer and more complex pieces of programs, for instance in our studies for Astrium, IRSN, and for Airbus to a lesser extent.

5.3.1 A simple addition...

The simple program below fully demonstrates how tricky floating-point computations can be, already on rather harmless-looking programs:

Listing 5.1: Typical error accumulation

```
1    float t, delta = 0.1;
2    for (int i=0;i<500;i++) {
3      t = t + delta;
4      FPRINT(t);      // assertion to print evolution graphs t(i)
5    }
```

Fluctuat finds out that after 500 such iterations, the relative error between the real and the floating-point computation of variable t is already equal to 3.81×10^{-6}, that is more than an order of magnitude larger than the elementary rounding error for simple precision. This is of course due to the fact that the nice looking constant 0.1 is not exactly representable as a floating-point number. One could then a little too quickly think that the error is quite easy to assess as a linear function of the iterate, the error on delta accumulating at each iteration. But this example on the contrary demonstrates that even on such an simple example the evolution of the error is tricky to assess. Indeed, we should not forget that the addition at each iteration also introduces an error, which becomes more important than the initial error on delta as t grows. In Figure 5.3, we can see that the main source of error is not the assignment `delta = 0.1`: the influence of this statement on the total error on `t` is negligible, as can be seen in the "error at current point" box when selecting this line in the error graph ($-7.45e^{-7}$ whereas the total error on `t` is $1.9e^{-4}$). The difficulty to predict the behavior of the rounding error even on such a simple example can also be observed on the evolution graph for the error on variable t as function of the iteration, shown in Figure 5.4.

5.3.2 A set of order 2 recursive filters

Consider the program of Listing 5.2: assertion `x = DBETWEEN(a,b)` is a Fluctuat directive specifying that double precision variable x can take any value between a and b, and that this initial value is without

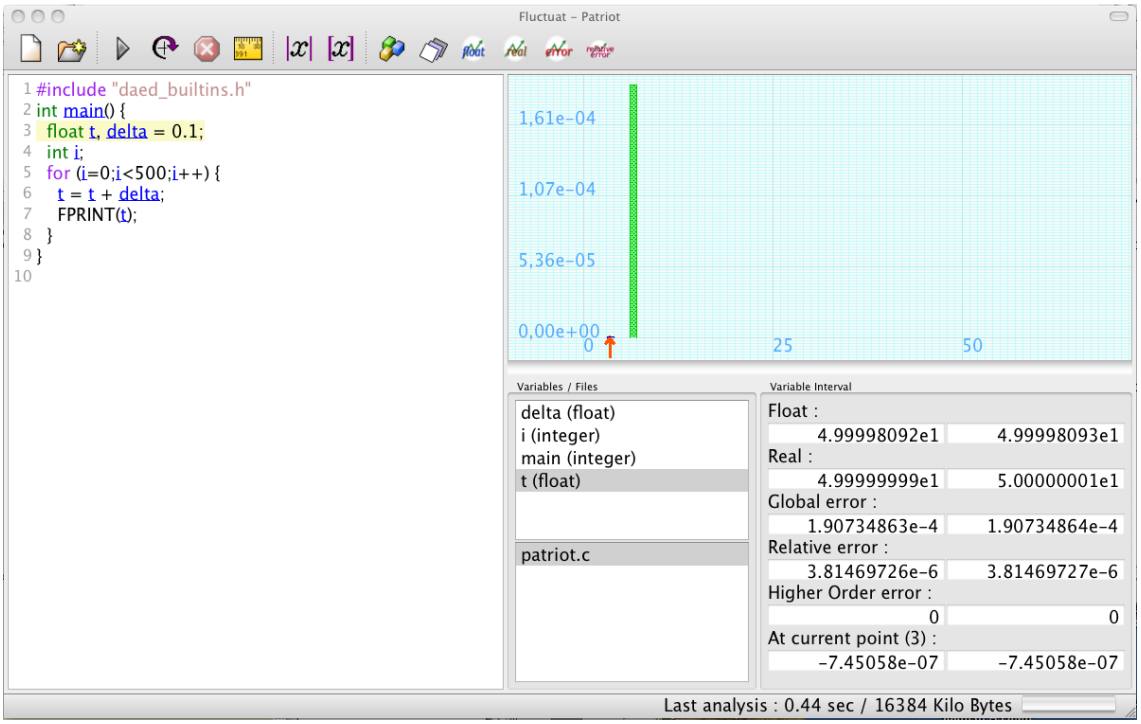
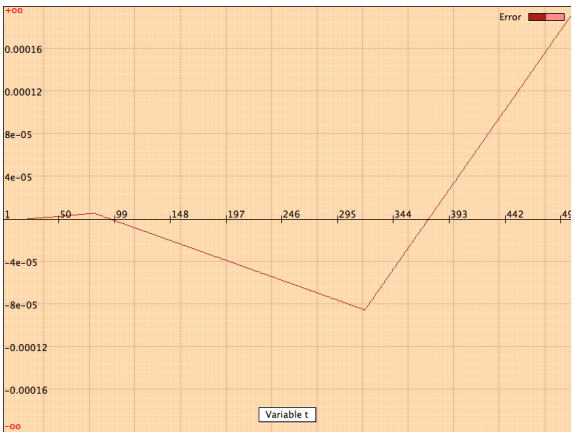


Figure 5.3: Variable t at the end of Listing5.1



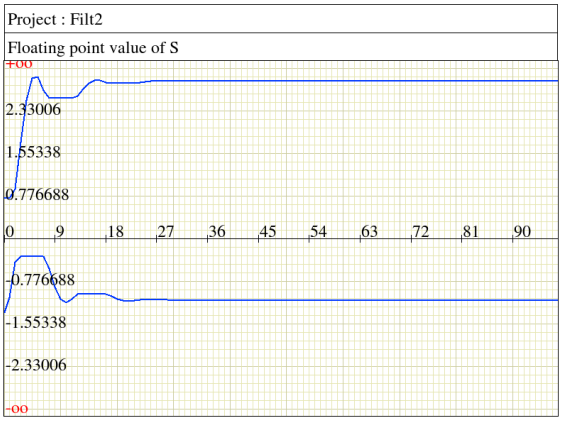


Figure 5.4: Evolution of the error on `t` in Listing 5.1 Figure 5.5: Evolution of the value of `S` in Listing 5.2

error (other assertions exist, that specify errors). This program implements a set of filters (as the filters coefficients are uncertain), run N times, and here for any integer N, with at each iteration a new unknown input E in $[0, 1]$, with unknown dependency to the previous inputs. The poles of the Z-transform of this filter have a module in $[0.932, 0.975]$, thus close to instability.

Listing 5.2: Set of order 2 filters

```
1   int main(void N) {
2   double A1, A2, A3, B1, B2, E, E0, E1, S, S0, S1;
3   A1 = DBETWEEN(0.5, 0.8); A2 = DBETWEEN(-1.5,-1); A3 = DBETWEEN(0.8,1.3);
4   B1 = DBETWEEN(1.39,1.41); B2 = DBETWEEN(-0.71,-0.69);
5   E=DBETWEEN(0,1.0); E0=DBETWEEN(0,1.0);
6   for (i=1;i<=N;i++) {
7     E1=E0; E0=E; E=DBETWEEN(0,1.0);
8     S1=S0; S0=S;
9     S=A1*E+E0*A2+E1*A3+S0*B1+S1*B2;
10  }
11  DSENSITIVITY(S);
12 }
```

Analyzing a particular filter in the previous set We first consider one particular filter among these set of filters, by fixing the coefficients of the filter: $A_1 = 0.7$, $A_2 = -1.3$, $A_3 = 1.1$, $B_1 = 1.4$ and $B_2 = -0.7$. We find the results summarized in Table 5.1. The first two columns of the table indicate the number of cyclic unfolding and the number of iterations after which we begin to widen instead of using the join operator.

c. unfold	widen.	time	mem.	S (float)	S (error)
10	150	92 s	45 Mb	$[-6.30; 7.96]$	$[-5.55 \times 10^{-14}; 5.58 \times 10^{-14}]$
30	40	24 s	45 Mb	$[-5.18; 6.84]$	$[-3.36 \times 10^{-14}; 3.38 \times 10^{-14}]$

Table 5.1: Filter for $A_1 = 0.7$, $A_2 = -1.3$, $A_3 = 1.1$, $B_1 = 1.4$ and $B_2 = -0.7$.

The higher the cyclic unfolding is, the lower the widening threshold can be to obtain convergence of the fixpoint computation. Also, on this example we increased the precision in number of bits of the mantissa of the numbers used for the analysis to 200 bits. This gives more precision, but the computation is slower and converges more slowly to a post fixpoint: for instance, using the default value of 60 mantissa bits in the case when the cyclic unfolding is 10, setting the widening threshold to 40 is enough to ensure convergence, and we get in 8 seconds the same real and floating-point values as before, and an error in $[-6.56 \times 10^{-13}, 6.56 \times 10^{-}13]$.

Completely unfolding the loop on 100 iterations confirms that the bounds found by the analysis are fairly precise: S is found to be in [-1.09, 2.76] with error in $[-1.15 \times 10^{-14}, 1.17 \times 10^{-14}]$. Still, there is room for improvement, with the global join operators on zonotopes presented in Chapter 2 (not yet implemented), or with finer iteration strategies.

Back to the set of filters

The results on the whole set of filters are summarized in Table 5.2. When considering the set of filters, with uncertain coefficients, the computations are no longer affine, and we thus subdivide some coefficients to

improve the accuracy. First column indicates how much we subdivide coefficients `B1` and `B2`. On the first line, we see the results for the full static analysis for `N` unknown. The second and third line show the results of the loop completely unfolded on the first 200 iterations (hence the analysis is particularized to `N`=200), first without subdividing coefficients `B1` and `B2`, and then, subdividing them. The results of the static

#B1,B2	c.	widen.	time	mem.	S (float)	S (error)
1, 1	200	20	430 s	50 Mb	$[-17.1; 18.6]$	$[-8.0 \times 10^{-14}; 8.0 \times 10^{-14}]$
1, 1	fully	no	6 s	53 Mb	$[-5.63; 7.13]$	$[-2.9 \times 10^{-14}; 2.9 \times 10^{-14}]$
10, 10	fully	no	1347 s	53Mb	$[-4.81; 6.33]$	$[-2.6 \times 10^{-14}; 2.6 \times 10^{-14}]$

Table 5.2: Results on the order 2 filter of Listing5.2

analysis (i.e. fixpoint calculation) are not too over-approximated compared to the results of the unfolded loop, which, as can be seen on the evolution graph of Figure 5.5, are quite stable on the last iterations.

Fluctuat, using the fact that affine sets give linearized functions of the outputs, can also automatically generate worst-case scenario that should reach a value close to the maximal (or minimal) bound on the value of some variable x specified by a `DSENSITIVITY(x)` directive in the source code. This works very well for the values, and is more heuristic for the errors though still satisfying in most cases: the affine forms for the errors are defined partially on value noise symbols, establishing a correlation between errors and values of inputs, that allow to partially control these errors. When asking here for a worst case scenario, leading to a maximum value of S, the analyzer proposes $A1 = 0.8$, $A2 = -1$, $A3 = 1.3$, $B1 = 1.41$, $B2 = -0.69$, along with values for the successive inputs E (see Figure 5.6): when executed with Fluctuat, this leads to $S = 5.36$. Proceeding in several steps to generate worst-case scenarios when the computation are non linear lead to better scenarios in general: for instance here, if we fix the coefficients of the filters as computed above, and only then ask for values for the inputs E, then we reach the larger value $S = 5.65$. We are actually quite close to the upper bound on the fixpoint given in Table 5.2.

5.3.3 An interpolated sine approximation

We use the example in this section to describe at the same time the results that can be obtained with Fluctuat, and the methodology to obtain these results: we detail the different steps that allow to progressively refine the results, starting from a program almost without directives, and with default analysis parameters.

Embedded programs classically use interpolation tables to approximate trigonometric functions, such as the sine of an angle expressed in degrees. For instance, one may build an array of 361 double precision numbers providing approximations of the sine function for every half degree:

$$\forall k = 0, 1, \ldots 360, \left| T[k] - \sin\left(\frac{k}{2}\right)\right| < 10^{-4},$$

and implement an approximation of sine by interpolating between the points of this table for angles between 0 and 180 degrees and using the identities

$$\forall x \in \mathbb{R},\ \forall k \in \mathbb{Z},\ \sin(x) = -\sin(-x) = \sin(x + 360k) = \sin\left(x - 360\left\lfloor\frac{x + 180}{360}\right\rfloor\right)$$

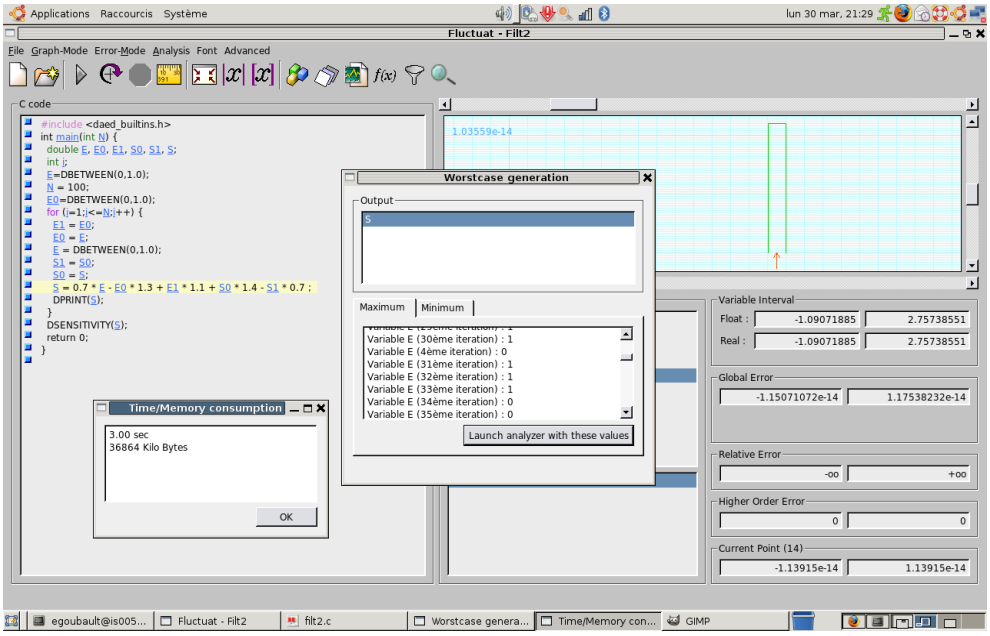


Figure 5.6: Worst case scenario on value of S in Listing 5.2

Let us try to run an analysis for the full range of double numbers, for the following naïve implementation of the sine operator:

Listing 5.3: Sine approximation

```
1  extern const double T[361];
2  double SIN_0_180(double x) {
3   double dx, i_dx, v_inf;
4   double v_sup; int i;
5   dx=2*x; i=dx; i_dx=i;
6   v_inf=T[i]; v_sup=T[i+1];
7   return v_inf + (dx - i_dx)
8          * (v_sup - v_inf); }
9
10 double SIN_180(double x) {
11  if (x<0)
12   return -SIN_0_180(-x);
13  else
14   return SIN_0_180(x); }
15
16 double SIN_POS(double x) {
17  if (x>180) return SIN_180(x
18        -360.*(int)((x+180.)/360.));
19  else       return SIN_180(x); }
20
21 double SIN(double x) {
22  if (x<0) return -SIN_POS(-x);
23  else     return SIN_POS(x); }
```

Fluctuat, in 1.5 seconds, issues infinite values and errors for the result of the sine function, and warns that the `(int)((x+180.)/360.)` cast may be undefined. Indeed this implementation uses a conversion

from double to int, so it is only valid for inputs x such that $\left\lfloor \frac{|x|+180}{360} \right\rfloor$ can be represented in the type `int`. In our 32 bits case, this is the $]-773094113100, 773094113100[$ interval. This is not an issue, provided that call contexts for `SIN` in the whole control program are guaranteed to fall into this last interval.

The analysis costs for the reduced domain are unchanged, but its result is still disappointing: the computed over-approximation of the value (and error) interval for the result of the sine algorithm in floating-point numbers is $[-3.09 \times 10^{12}, 3.09 \times 10^{12}]$, whereas it is $[-1.5, 1.5]$ in the real-number semantics. Besides, Fluctuat warns the user that the program test in function `SIN_180` is unstable: it stresses here that whenever the floating-point execution takes the `(x<0)` branch, the real execution is sure to execute the `(x>=0)` branch. This requires careful attention.

As a matter of fact, whenever `x` is very close to (but smaller than) a number of the form $180 + 360 \times k$, for some $k \in \mathbb{N}$, then a rounding error occurs when evaluating expression `(x+180.)/360.` in floating-point arithmetic. The result is rounded to the `double` representing $k + 1$. In all such cases, the `x - 360.*(int)((x+180.)/360.)` expression has value close to (though below) 180 in real numbers, but close to (though below) -180 in floats. In such a case, expression $T[i+1]$ in function `SIN_0_180` attempts to access array `T` out of bounds, as signaled by Fluctuat. We thus fix the naïve implementation of the sine operator: we add a 362^{nd} array element, such that $T[0] = T[360] = T[361] = \sin(0) = \sin(\pm 180) = 0$.

Custom non uniform subdivisions Now we have fixed the implementation, we would like to have an idea of the accuracy that can be ultimately expected from the analysis. For that, we switch to the symbolic execution mode of Fluctuat with same range and subdivisions, thus choosing a sample of inputs in the input range. As a result, we get a useful (though unsound for the whole range of values) estimate of the error range for y: $[-1.05 \times 10^{-16}, 1.05 \times 10^{-16}]$.

We thus look for a more precise way to perform the static analysis: we write a new main function implementing a custom irregular subdivision, with a singleton for every interpolation point of the interpolation table:

$$[-180, 180.5[= \bigcup_{i=-360}^{360} \left\{ \frac{i}{2} \right\} \cup \left] \frac{i}{2}, \frac{i+1}{2} \right[$$

This improves the accuracy of the analysis by a "manual" disjunctive analysis.

Listing 5.4: Custom non uniform subdivision

```
1  double x=-180., y=0., xi, yi; int i;
2  for (i=-360; i<=360; i++) {
3   xi = i*0.5;  yi = SIN(xi);
4   x = DCOLLECT(x, xi);  y = DCOLLECT(y, yi);
5   xi = DBETWEEN(DSUCC(i*0.5), DPREC((i+1)*0.5)) ;
6   yi = SIN(xi);
7   x = DCOLLECT(x, xi);  y = DCOLLECT(y, yi); }
```

We then collect back the values that variables can take on the full domain using directive `DCOLLECT`, see Listing 5.4. We could also have used a variation of this assertion `DCOLLECT_WARN`, that checks that there is no hole in the values collected, and that we have thus subdivided the whole initial range.

The static analysis is run unrolling the main loop. It takes 97 seconds and 36 MB, and ensures that $x \in [-1.8 \times 10^2, 1.805 \times 10^2]$ and $y \in [-1.0, 1.0]$ with errors in $[-4.97 \times 10^{-16}, 4.97 \times 10^{-16}]$. The imprecision generated by the floating-point implementation is thus proved negligible compared to the accuracy of the interpolation table.

We may also run sensitivity analyses for all $[\frac{i}{2}, \frac{i+1}{2}[$ sub-intervals of the $[-180, 180[$ range. Merging the 720 results, we can guarantee $|\frac{\Delta y}{\Delta x}| \leq 0.0176$. This is of course a very satisfactory result, as we are approximating a real-valued sine function expressed in degrees, i.e. such that

$$\left|\frac{d}{dx}\left(\sin\left(\frac{180}{\pi}x\right)\right)\right| = \left|\frac{180}{\pi}\cos\left(\frac{180}{\pi}x\right)\right| \leq \frac{180}{\pi} \sim 0.0175$$

Approximate functional proof of special identities We want here to check that the approximate operator satisfies, in real and in floating-point numbers, some properties close to classic properties of the real sin function. For instance here:

$$\sin(x) = -\sin(-x) = sin(180 - x) \quad \sin(x)^2 + \sin(x - 90)^2 = 1$$

For that, between lines 6 and 7 in Listing 5.4, we add the following lines:

```
1  c1 = SIN(xi-90);
2  z1 = y1 + SIN(-xi);
3  z2 = y1-SIN(180-xi);
4  z3 = y1*y1+c1*c1-1.0;
```

Fluctuat proves that the real values of $z1$ and $z2$ are in $[-4.34 \times 10^{-18}, 4.34 \times 10^{-18}]$ and their floating-point value in $[-5.44 \times 10^{-16}, 5.44 \times 10^{-16}]$. Property $\sin(x) = -\sin(-x) = sin(180-x)$ holds almost exactly for the approximate operator, and the rounding error is predominant here.

But the real value of $z3$ is in $[-1.32 \times 10^{-4}, 1.21 \times 10^{-4}]$ with negligible error, bounded by $[-8.44 \times 10^{-16}, 7.93 \times 10^{-16}]$. Here, the approximation errors seems large. And indeed, if asked for a "worst case", Fluctuat delivers as best guess for reaching the maximum of $z3$, $x = 127.5$. Executing the code with this value, we get $z3 = 1.21 \times 10^{-4}$ confirming the supremum bound found by static analysis. This is due to an error of about 10^{-4} on the corresponding table entries: the algorithm is actually most probably computing something quite close to the sine function, with approximately the 10^{-4} absolute precision expected, mostly due to the algorithm (the interpolation), and not to its finite precision implementation.

5.3.4 Cosine approximation

For the sine approximation, we proved that some inequality approximately holds. We an also use Fluctuat to compare the implementation of a mathematical function to the actual function, like here for a cosine approximation. In this example, we will even prove that the implementation 5.5, which is a variation of that proposed in `http://www.netlib.org/fdlibm/k_cos.c`, actually approximates very closely the cosine function.

Listing 5.5: Cosine approximation

```
1  include ``math.h''
2  double C1, C2, C3, C4, C5, C6;
3  C1=4.16666666666666019037e-02;
4  C2=-1.38888888888741095749e-03;
5  C3=2.48015872894767294178e-05;
6  C4=-2.75573143513906633035e-07;
7  C5=2.08757232129817482790e-09;
8  C6=-1.13596475577881948265e-11;
9
10 double mcos(double x) {
11   double a,hz,z,r,qx,zr;
12         z  = x*x;
13         if (x<0) x = -x;
14         hz = 0.5*z;
15         r  = z*(C1+z*(C2+z*(C3+z*(C4+z*(C5+z*
               C6)))));
16         zr = z*r;
17         if (x < 0x1.33333p-2)
```

```
            /* if |x| < 0.3 */
18          return 1. - (hz - zr);
19        else {
20          if(x > 0.78125) {          /* x
                > 0.78125 */
21            qx = 0.28125;
22          } else {
23            float f = x;
24            qx = 0.25f * f;
25          }
26          hz = hz-qx;
27          a  = 1.-qx;
28          return a - (hz - zr);
29        }
30  }
31
32  void main(void)
33  {
34    double x,y,z;
35    double pi = 3.141592653589793238462;
36    x = DBETWEEN(0.5,0.75);
37    y = mcos(x);
38    z = y - cos(x);
39  }
```

We study function `mcos` on the range [0.5,0.75], this is added using the DBETWEEN assertion again. And we add line `z = y - cos(x)`, where cos is the mathematical cosine function, in order to prove that function `mcos` actually computes something very close to the mathematical function. Bounds on the value of z will bound the approximation error, while bounds on its error will bound the error due to finite precision.

Fluctuat with default parameters gives in 0.01 sec, z in $[-7.51 \times 10^{-3}, 7.71 \times 10^{-3}]$ with computation error bounds $[-2.7 \times 10^{-16}, 2.7 \times 10^{-16}]$. Subdivisions on the input allow to refine this estimation: for instance, in 10 sec, 16Mb and 1000 subdivisions, we have z in $-7.5 \times 10^{-9}, 7.5 \times 10^{-9}]$, with an error in $[-2.62 \times 10^{-16}, 2.62 \times 10^{-16}]$. And n 4 hours of computation, we obtain that the real value of z is in $[-1.88 \times 10^{-15}, 1.88 \times 10^{-15}]$, with an error due to finite precision in $[-3.45 \times 10^{-16}, 2.56 \times 10^{-16}]$. Subdividing even more allows to keep reducing the real value of z, that is the error due to the approximation scheme, while the error due to finite precision remains of the same order, so that the mcos function indeed approaches the cosine very closely. But the computation times become unreasonable.

Proving such refined properties is not a central use of Fluctuat for the time being: the zonotopic abstract domains are well suited to computations that do not involve too many non-linear computations, and for large ranges of values. But this example demonstrates that if time is not an issue, like on elementary symbols that need to be analyzed once, such use is conceivable, whereas it is not with a non relational domain such as intervals.

5.3.5 An iterative algorithm for square root approximation

We finally consider an example that illustrates the difficult case when the use of finite precision does not only affect the accuracy of the result, but also the behaviour of the program. The function below computes by the Householder method the inverse of the square root of the input I. The loop stops when the difference between two successive iterates is below a criterion that depends on a value `eps`.

```
1     xn = 1.0/I; i = 0; residu = 2.0*eps;
2     while (fabs(residu) > eps) {
3       xnp1 = xn*(1.875+I*xn*xn*(-1.25+0.375*I*xn*xn));
4       residu = 2.0*(xnp1-xn)/(xn+xnp1);
5       xn = xnp1; i++; }
6     O = 1.0/xnp1;  sbz = O- sqrt(I);
```

This algorithm is known to quickly converge when in real numbers, whatever the value of `eps` is. However, in finite precision, the stopping criterion must be defined with caution: for instance, executed in simple precision, with `eps`=10^{-8}, the program never terminates for some inputs, for instance $I = 16.000005$.

Analyzed with Fluctuat for an input in $[16, 16.1]$, we obtain that the number of iterations i of the algorithm is potentially unbounded. We also obtain that $residu$ has a real value bounded in $[-3.66\times10^{-9}, 3.67\times10^{-9}]$ and a floating-point value is bounded in $[-2.79\times10^{-7}, 2.79\times10^{-7}]$ (50 subdivisions), so that the real value satisfies the stopping criterion but not the floating-point value, which is also signaled by an unstable test on the loop condition. Now, analyzing the same program but now for double precision variables, Fluctuat is able to prove in 3 seconds that the number of iterations is always 6 (both the real value and the floating-point value satisfy the stopping criterion). Also, bounding the variable `sbz`, Fluctuat is also able to bound the method error, so that we prove that the program indeed computes something that is close to the square root: the real and float value of `sbz` are found in $[-1.03 \times 10^{-8}, -1.03 \times 10^{-8}]$, with an error due to the use of finite precision in $[-1.05 \times 10^{-14}, 1.05 \times 10^{-14}]$.

5.3.6 Analysis of hybrid systems

An emerging trend in software verification is to extend the analysis of programs to take into account their interaction with the external world in a more refined manner than just specifying ranges of values for inputs and outputs of the programs. We extended Fluctuat in this direction and introduced new directives to describe the interaction of the program with a continuous environment, modeled as a set of switched ODEs, given by the user as C++ functions. The directives model the action of sensors (with which the program reads the value of a physical variable, solution of an ODE) and actuators (with which the program acts on the system's behavior, that is modifies modes or parameters of the switched ODEs). Then Fluctuat calls a guaranteed ODE solver to bound the behavior of the continuous environment (we used the C++ library GRKLib [23]).

In our model, time is computed by the program: Fluctuat initializes the set of ODEs by sending bounds on initial values, modes and parameters to the environment. Then any action by the program is also sent to the environment (the ODE solver) when encountering the corresponding actuator directives in the program. When Fluctuat encounters a directive requesting a sensor value in the program, it calls the guaranteed ODE solver, specifying the time at which it wants to read bounds on the continuous value of the given quantity, and the integration is performed on the fly. This extension is described in more details in [22].

We illustrated its use on a simplified version of the MSU main control loop, part of the Automated Transfer Vehicle (ATV) control software, represented Figure 5.7. The external environment (relating the position of the ATV to accelerations due to thrusters) is modeled by an ODE of dimension 7. At each cycle of the program, the vehicle's configuration (the position, speed, acceleration etc.) is read from sensors (assertion `HYBRID_DVALUE` in the program), then the program computes the command sent to actuators to achieve the desired thrust with the engines (assertion `HYBRID_PARAM` in the program, that modifies the parameters of the ODE system modeling the environment).

We studied the behavior of the system on 25 seconds, starting from one point, and could prove that the values converge towards the the safe state that the ATV is supposed to reach in this escape mode. More details and analysis results can be found in [19].

5.3.7 Float to fixed-point conversion: bit-width optimisation

There is a need in digital processing for automatic floating-point to fixed-point transformation of programs, guaranteeing output accuracy while minimizing the cost of implementation, cost for instance defined by the sum of word lengths of all variables. We developed and implemented in Fluctuat in 2005, during the M2

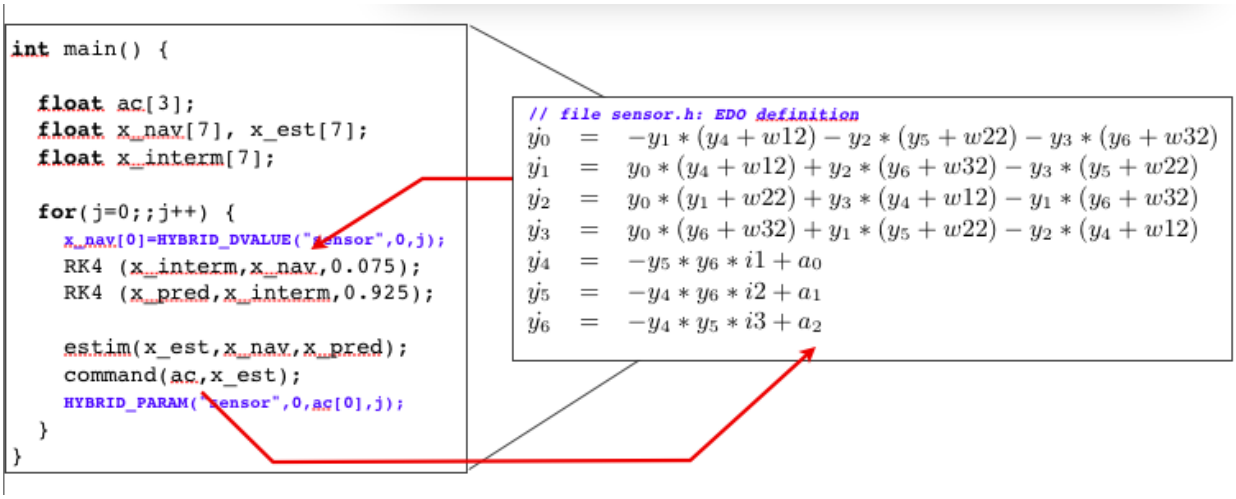


Figure 5.7: Hybrid system modeling: the ATV control software

internship of Jérôme Mascunan [122], the counterpart of the non relational abstract domain for floating-point numbers, but for fixed-point numbers. Special Fluctuat assertions, give the size of word length and integer word length for each variable. Then, we started working on a small optimization tool, which, given a C program written in floating-point numbers and an aim in terms of accuracy compared to the accuracy of the floating-point program, should try to compute optimal (smallest) fixed point formats for each variable. The aim is typically to achieve some percentage of the accuracy of the floating-point program; constraints can also be added on the fixed point formats (such as fixed word length), so that the optimization can be eased. The tool relies on the dependencies between variables computed by the abstract domain on the floating-point program, as heuristics to decide the format of which variable to modify in order to act on a given variable's precision.

This work is very preliminary, and relies on non relational abstractions of the values and errors. We intend to improve our conversion algorithm, using the dependencies we are now able to compute, in the DEFIS ANR project. We shall compare our approach and results to other approaches [117, 159], which also use value and fixed point error models relying on affine arithmetic for bit-width optimization, one by simulated annealing, the other by a greedy search algorithm. In digital processing, a commonly used criteria for the accuracy of a fixed-point implementation, is the signal-to-quantization-noise ratio, elementary quantization errors being considered as uniformly distributed white noises. The CAIRN Inria team, which leads the DEFIS project, uses a partially analytical approach to compute such ratios [123], and optimize the floating-point to fixed-point conversion [124]; their noise propagation models based on perturbation theory and Taylor developments resemble in some way to our own propagation model. We intend to exchange information with this approach.

5.4 Looking back at the last ten years

Finally, let us quickly come back on the now more than 10 years of existence of Fluctuat, the development and transfer of which constantly accompanied and benefited from the research work presented in this

document. It is a research prototype, but we have always had a high concern for usability by reasonably non-expert users: the evolutions were equally driven by our progress on the zonotopic abstract domains, and requirements and observations on the industrial source codes that were submitted to us.

Propagation of rounding errors, proof of concept (2001-2003) We started from an already existing simple abstract interpreter developed at CEA [75], in which we first implemented the non-relational domain of Section 5.1.2. Our main industrial supports at that time were Airbus, through the Daedalus European IST project, and the IRSN (French Institute for Radioprotection and Nuclear Safety). In both cases, the objective was to be able to numerically study large programs, and identify loss of accuracy due to floating-point numbers. We designed the user interface that represents the decomposition of elementary rounding errors as histograms, and links the error bars to their provenance in the source code. Soon, the need for information about intermediary steps in the analysis appeared. We added the first printing assertions and evolution graphs. Case studies concentrated on the propagation of perturbations around representative scenarios, the symbolic execution mode was introduced and much used. The analyzer allowed us to point out some classic floating-point issues such as oscillation problems due to sampling and catastrophic elimination.

First steps for a methodology, zonotopic domains (2004-2005) I would say the present Fluctuat takes much of its roots from this period. We were encouraged by our users to put some effort on the interpretation of large pieces of software. This involves being able to interpret most most ANSI C constructs, at least those present in embedded code (no recursion): alias/arrays, casts from floats to integers by bit-wise operations, and more generally all integers and bit-wise operations, including modular arithmetic. They take time, as their design for our model of propagation of uncertainties is not obvious. Interpreting large parts of full software also means giving the user some tools to interpret results, identify some parts that are more difficult, and analyze them separately. We sorted the large amount of information at the end of the analysis, by issuing warnings for the main points to be checked by the user, and displaying the error bars by decreasing amplitude. We also started printing more information about the current state of the analysis, so that the user needs not necessary wait for the end of a possibly long analysis. We also added a basic in/out analysis, in order to point out variables that need to be initialized with assertions, or dead code information (code that is proved to be dead): this is especially useful when isolating a part of program. This methodology for the analysis of large programs was experimented on an extract of Airbus programs.

We also naturally felt the need for a more accurate analysis of the critical places. We began implementing a first version of our zonotopic abstract domain, relational on the values only at that time (not on the errors) [81] (a hint on its extension to the errors was also presented around the same time [80]). With this new abstract domain came new assertions, specifying the real value, but also bounds on the gradient for recurrent inputs, in order to avoid non realistic behaviours. Meanwhile, we introduced the possibility in the analyzer to use regular subdivisions of some inputs - and for the whole program - to improve the accuracy for non linear computations. With this new abstract domain, we started interesting ourselves to the elementary symbols which are present in most control software, typically interpolation operators, linear recursive filters, and approximations of elementary functions such as trigonometric functions. This work opened the way for some future joint publications with users, and future transfer to industrial users. We obtained at that time first results of the analyzer proving termination of an algorithm in real but not in floating-point numbers, because of the loop exit condition.

We finally started to spend time on different iteration strategies for fixpoint computations, with initial

and cyclic unfolding, widening by reduction of the precision, strategies for nested loops and function calls, and for the level of dependency kept in loops or function calls.

Towards functional proof and maturity of the analyzer (2006-2009) Building on previous experience, we developed during this time the zonotopic abstract domains for both values and errors that are still presently used in Fluctuat, although these abstract domains were improved since. These new domains allowed us to compute accurate bounds on the error and the floating-point value as well, when they were previously accurate on the real value only, with a drift of the error that prevented fixpoint computation. We also further exploited these domains, with sensitivity analysis and worst case generation. These were used in particular for the case studies, that were mostly, but not exclusively, focused at elementary symbols analysis. We also started to prove functional properties on these symbols. As more refined properties were proved, we generalized the use of custom irregular subdivisions. The case studies lead at this time to several joint publications with the users [87, 46, 19]. We also produced for Airbus a report on the analysis with Fluctuat of A380 elementary symbols, that prepared the way for use of the analyzer on field.

Towards industrial transfer and new uses (2008-present) We kept improving the abstract domains (join and meet over affine sets in particular), but the accent was also put on cleaning the older parts, like the front-end, improving the usability and defining new usages, in particular in view of a potential valorisation. Karim Tekkal joined us (2008-2011) for a Digiteo maturation operation first, then more explicitly with the objective of a start-up creation (project since abandoned). Franck Védrine, who had previously worked on the TWO analyzer, Fluctuat's ancestor, also joined us in 2008, and committed in the valorisation project. These two arrivals brought a welcome impetus! A first extension to hybrid systems [22] was developed, with Olivier Bouissou, which we applied to the analysis of the ATV escape program [19]. On the application's side, we considered again larger programs, but also with new difficulties, such as the quaternions of the ATV study. Worst case generation was extended to error maximization, subdivisions were refined for local, possibly irregular, subdivisions. The proof of functional properties was more systematically explored and more user-oriented assertions were added, for instance to compare or replace parts of programs, or ask for different safety checks by the analyzer.

At the same time, we started thinking about possible interactions with other tools: to mention a few of them, partial information was integrated in SCADE in the INTERESTED project; we are thinking of possible interfaces with the Frama-C platform via the interpretation of the ACSL language; in the DEFIS ANR project, we are discussing interactions around the float to fixed-point conversion of programs. For these purposes, a library was created in order to call and interact more smoothly with the heart of Fluctuat. It was used in particular to design an interactive version of Fluctuat [76].

Chapter 6

Conclusion and Perspectives

I synthesized in this document my research work at CEA LIST since 2001, first in the Software Safety Laboratory (LSL) led by Jacques Raguideau, then in the Laboratory for the Modelling and Analysis of Systems in Interaction (LMeASI) led by Eric Goubault. I focused on the validation of numerical programs, drawing inspiration from such different fields as theoretical computer science, software engineering, control theory, numerical analysis, reliable computing and computer arithmetic. My main contributions on this subject are the following. I first introduced, with Eric Goubault, a class of new abstract domains relying on a parameterization by noise symbols: the abstract domain that was studied the more extensively was the zonotope (or affine sets) domain, but many extensions can be derived, as partly described in Chapter 3. I also contributed to the introduction of a new general method for the solution of fixpoint equations occurring in static analysis; and meanwhile studied and implemented Kleene-like iterations in the context of zonotopic abstractions. I finally adapted the zonotopic abstract domains for the analysis of finite precision computations, and implemented these abstract domains in the Fluctuat analyzer. More generally, I have been the main developer of the Fluctuat analyzer during several years, and realized or participated in many case studies (Airbus, IRSN, Hispano-Suiza, EADS, Continental, etc), that drove us to the present possibilities of the tool.

Let me draw some perspectives on these different themes over the next few years:

6.1 Zonotopic abstract domains and extensions

We set up a formal framework for a fast and accurate abstract analysis based on affine sets, which is quite mature now. Improvements of affine sets, such as a better join operator in the case we do not exactly have common affine relations, would still be very valuable. But most future work concern the extensions of this abstract domain presented in Chapter 3, or even other possible extensions, we give a few leads hereafter.

Under-approximations We have proposed in 2007 [82] a first practical, tractable abstract semantics, based on generalized affine sets, for under-approximating the values of program variables. One current objective is to extend this method to higher-order Taylor forms, for instance by using zonotopic coefficients instead of interval coefficients in our generalized affine forms. This also remains to be compared to other existing methods for under-approximations, mostly introduced in control theory (and called inner-approximations in that context), with the idea to improve our first results. Most other methods for under-approximation use bisections algorithms, that allow to build box pavings that are included in the image of

a vector-valued function. We have recently focused on the approach developed by Goldsztejn et al. [71], which was defined only for functions whose domain and co-domain have same dimensions. For applications to static analysis or robust control, this is usually not the case. Olivier Mullier, in his PhD thesis, which I co-supervise, proposed an extension to the general case, and first results were presented at SWIM'11 [135]. Our interest also goes to the nested inner-approximations for polynomial matrix inequalities [98], even though these methods may prove intractable for our applications, or to inner-approximations by ellipsoids [109], two methods that manage to give somewhat dual over and under-approximations, as we are aiming at. Our generalized affine sets for under-approximation are somehow dual to over-approximation, as the same forms can be interpreted both as under or over-approximation. But the under-approximations we obtain do not have a geometric concretization as zonotopes. Actually, we are currently trying to characterize their concretization, or at least find convex shapes that can be enclosed.

We then hope to use our under-approximation methods in both contexts of static analysis of programs and robust control of uncertain dynamical systems, this is again among the objectives of O.Mullier's PhD thesis. For instance, we want to obtain an over- and under-approximation of the set of states that are reachable by a physical system described by a discrete-time dynamical equation involving uncertain parameters, with constraints on the inputs of the system (bounds, energy...). This problem is similar to a viability problem [6], with uncertainties in the system dynamics. This step will then allow us to consider the synthesis of commands (input sequences) of the system allowing to lead it from a state to another, with a guaranteed robustness towards incoherent parameters. Still, we did not find yet a completely satisfying way to apply our under-approximations described in Section 3.1, which for the time being can solely be interpreted as AE formulas, to these robust control approach.

Finally, the under-approximating abstract semantics we introduced is for the time being applied to real-valued variables, and does not address yet floating-point variables, which are discrete by nature. As a partial solution, one can think for instance of adding information about the minimal size of the gaps between two floating-point values in the resulting interval.

P-boxes and affine arithmetic We have for now defined an arithmetic relying on p-boxes and affine arithmetic. There still remains to apply this to the static analysis of programs: we must in particular define set-theoretic operations such as union and intersection, probably by generalizing those we introduced for affine sets. Another direction is to maintain a better abstraction of the dependencies between variables, during computation. In the current work, we decomposed the dependency relationships between variables on a basis of independent noise symbols (ε_i) and noise symbols with unknown dependency (η_j). This is a particular abstraction of some form of a correlation matrix between the noise symbols, which we hope to refine in the future. Finally, the probabilistic affine forms we have introduced should be linked with particular Dempster-Shafer structures on $\mathbb{R}^p$, whose focal elements are zonotopes (particular central symmetric polytopes), parametrized by affine forms. These in turn might generalize to higher-order Taylor models as used by Makino and Berz [120] instead of order one models such as zonotopes.

A generalization of zonotopes and ellipsoids: super-ellipsoids The use of ellipsoids has been widely studied in set estimation, control, and reachability analysis [137, 108], and also applied in the context of static analysis by Feret [49]. Also, as already stated, among other interests, ellipsoids have been used to provide both under and over-approximations, as we intend to. However, ellipsoidal calculus [110] is rather intricate and costly. It could thus be interesting to investigate a variation of our abstract domain of

affine sets based on ellipsoids, or combining zonotopes and ellipsoids. Indeed, as we noted in a NSV'11 presentation [86], in the same way a zonotope is the image by an affine transformation of an hypercube, an ellipsoid is the affine transformation of an n-dimensional disk, and can thus be parameterized by an affine form, just like zonotopes, except that the vector of noise symbols is bounded in l_2 norm instead of in l_∞ norm. And this can generalize to other norms, yielding super-ellipsoids. Some arithmetic operations could thus be achieved using the same arithmetic as on affine forms, avoiding a costly ellipsoidal calculus. And we would need much fewer terms to compute some loop invariants. This of course remains to be further developed and experimented.

Other approximations We believe that we should be able to extend our zonotopic approach to deal with higher-order Taylor models [120], where we use a polynomial approximation of functions with monomials on noise symbols ε^α with $|\alpha| \leq n$. Other ways to get higher-order approximations could also rely on affine sets with zonotopic coefficients, probably in the line of the affmat objects of Combastel [36]. From our first observations, this second approach would in particular probably be more suited for building under-approximations.

But, while these forms are probably well suited to bound rounding errors, which are small compared to values, they may not be suitable to bound the values of variables: most of the time in static analysis, we want to study the behavior of the program for large ranges of values. In this case, other rigorous approximations may be more interesting, such as for instance those relying on Chebyshev interpolation polynomials. The idea of using better approximation polynomials, for example Chebyshev truncated series, in the context of validated computations, was already introduced in 1982 [48] under the name ultra-arithmetic. As discussed by Makino and Berz [120], the arithmetic proposed there has some drawbacks that make it less accurate than Taylor models. However, new Chebyshev interpolation models were proposed, for instance by Brisebarre et al. [24], that could be of interest to us.

Finally, a track that we intend to follow in the future, and believe very promising, builds on ideas of Sankaranarayanan [146]: he uses changes of bases to transform a non linear dynamical system into a linear one, that can be solved much more easily, by classic methods. We would like to extend these ideas to discrete systems, and automatically compute change of bases that would allow us to abstract systems that require non-linear invariants, by systems that could be solved efficiently with a linear abstract domain such as affine sets. These changes of bases could be polynomial, but we plan to possibly also include rational functions, logarithm or exponential.

6.2 Fixpoint computations

Kleene-like iterations One future direction is to use modified iteration schemes as a mean to compute global join operations, that preserve relations between variables and noise symbols, but in a more general way than as proposed in Section 2.2.3. For instance, there are cases when there is no common affine relationship, so that the global join of Section 2.2.3 does not apply, but still, this component-wise computation loses some crucial correlations. Informally, the idea of this modified iteration is, when dependencies between variables are discovered by our affine sets, to make the union only on part of these variables, and propagate it to the other dependent variables. This is close to the global join that we defined, but would apply more widely than just the case of exact common relationships.

Another direction relies on the fact that typical critical real-time systems, as found in automotive, aeronautic, and aerospace applications, usually consist in a huge iterated loop. But these systems do not run forever, and this loop will not be executed more than a (possibly very large) number of times. And when implemented with finite precision numbers, it may be for instance that there is a drift between the real and the finite precision, that cannot be bounded for any number of iterations. We would still like to characterize and bound this drift, and more generally bound the abstract values of the program variables, using this bound on the number of iterations. As our global join operators explicit the implicit relation between program variables (and thus for instance a loop counter, but not exclusively so), we think we could then use some abstract accelerations [72, 150] to accurately bound the behavior of such programs. This would also be close to the work of Feret [50] that defines an abstract domain relating variable values to a clock counter by affine expressions, and uses this relation to give bounds on the behavior that depend on the clock, but adapted to our abstract domain, and, we hope, of use on a larger class of programs.

Policy iteration for zonotopic abstract domains Policy iteration is a general algorithm that can be applied to many abstract domains - and has indeed been applied to several - provided that our loop functionals can be expressed as infimums of a finite set of simpler functionals. We started investigating two directions in order to design a policy iteration algorithm for our affine sets abstract domains, but these are still preliminary work. These two directions respectively define as policies, choices for the join or for the meet of two affine sets.

The meet operation in affine sets is expressed as constraints on the noise symbols, that can be interpreted in any abstract domain. If interpreted in intervals, we are then brought back to a variation of the policy iteration defined in Section 4.2. A policy is then, for each bound of each noise symbol occurring in the test, a choice between the old bound and the new bound generated by the constraint. Still, this idea remains to be studied in depth, as its use is more problematic than the simple policy iteration on intervals.

During the internship of Mickael Buchet, we started some work on a simple idea in order to define policies on the join operator: we noted that the join operator of Definition 11 allows us to define policies for the coefficients of the noise symbols. Indeed, $\text{argmin}_{|.|}(x, y)$ is either x, or y, or 0, which can be chosen as policies. However, the problem is that this does not necessarily define a descending policy iteration: the change of policy does not necessarily conduct to an improvement of the post fixpoint. Still, a first algorithm was designed that allowed to avoid cycles, and the implementation gave promising results.

6.3 Fluctuat and its applications

We intend of course to gradually integrate the new abstract domains and fixpoint iteration techniques previously described to the Fluctuat analyzer. A specific need linked to the analysis of the impact of finite precision, and that appeared in the use of Fluctuat, is the analysis of the behavior of program along the path conditions:

Discontinuity analysis As mentioned in Section 5.1.4, we are currently exploring the use of affine sets to make our analysis sound in presence of unstable tests. This is done by studying the conditions and the potential discontinuity of the program when the floating-point execution takes one path and the real execution takes another path, and adding to the error on variables, a discontinuity term bounding this difference. This

will also give a (dis)continuity analysis in the sense of Chaudhuri et al. [26, 27], that is say if a program represents a continuous function of its inputs, and is thus robust to uncertainties on these inputs.

Modular analysis As mentioned in Chapter 2, we started to experiment, for the time being on a separate prototype, a modular abstraction based on zonotopic abstract domains [89]. It would be interesting to integrate in the future this possibility to Fluctuat, in order to allow the analysis of larger pieces of programs, as a modular analysis can prove much cheaper than the full monolithic analysis. The functional abstraction by zonotopes is indeed well suited to a summary-based approach, where summaries can be instantiated to a given call at low cost and with very correct accuracy. Then, the use of policy iteration can accelerate the incremental analysis of a large piece of program: the policy giving the fixpoint of a given piece of program can be used as a very good initial guess for the same piece of program but with a small modification. Still, the use of modularity strongly depends on the structure of the program to analyze, and this remains to be further studied on actual applications.

Exchanging information We initiated some thinking on the possible interactions of Fluctuat with other tools. We took first steps already in this direction, for instance with the extension in HybridFluctuat to hybrid systems, where some directives in the source code are used to call a guaranteed differential equations solver. This can be further developed in several directions, first by studying the propagation of the imprecision in the continuous environment, but also by bounding the method error due to an approximate ODE solver in a program, thanks to a call to a guaranteed ODE solver. This latter extension requires additional assertions to compare a piece of code to a specification, that would in this case be a continuous model.

Some new directives could also be used in order to exchange information with other analysis tools, such as the provers integrated in the Frama-C platform[1]. This platform contains a chain of tools, that work on programs annotated by behavioral properties expressed in the ACSL language [8], that can be used to specify computations both in real and floating-point numbers, and to speak about mathematical functions. This chain can be used for the formal verification of numerical programs, as described for instance in Boldo and Marche [17]. Exchanges between Fluctuat and such framework could benefit to both: for instance, automatically synthesized invariants of Fluctuat could be used as first annotation for the provers, partially saving the user a tedious manual phase; in the other hand, some refined properties that can be proved on a piece of program, can be used either as an annotation of the program analyzed by Fluctuat, or directly as a property inside Fluctuat. For instance, the property that if x and y are two floating-point values such that $\frac{y}{2} \leq x \leq 2y$, then their floating-point subtraction is exact, which was discovered by Sterbenz in the 1970s [154], is a property used by the abstract domains of Fluctuat. But we could imagine to use more such properties, that would be specified in the program analyzed, and interpreted and used by Fluctuat. More generally, interpreting a specification language sufficiently general and expressive to define complex numerical behavioral properties of C programs, is a goal that facilitates the constitution of benchmarks for numerical program analysis, which is of course desirable.

We also hope to improve in the future the results of our analysis by automatically calling constraint solvers, as already partially demonstrated by Ponsini et al. [138]. Interactions with other tools could also for instance include interactions with a compiler, or an analyzer of compiled code, in order for instance to consider more refined information such as the actual order of execution of floating-point operations.

[1] http://frama-c.cea.fr/

Application fields The problematic of the validation of safety-critical systems is crucial, and justifies in itself long-term researches for the validation of numerical properties of programs. We will naturally keep developing our methods and tools in directions that come up from industrial test cases.

But in the long term, we can hope these advances will also benefit to less critical numerical systems. For instance, they can benefit to the conception of embedded programs, possibly in fixed point numbers, that optimize some aspects such as energy consumption under some given accuracy requirements. This is the problem we will be addressing in the DEFIS ANR project.

Another challenge is to analyze numerically intricate programs coming from the area of simulation, where there is a high demand for such tools.

In scientific computing programs, different kinds of approximations occur: first the mathematical model for the physical phenomenon of interest is already an approximation; then an approximate algorithm is used to solve that model: accuracy often depends on discretization steps for space and time, then linear systems are often solved by approximate iterative methods, which accuracy depends on a stopping criterion; and finally, this algorithm is implemented in finite precision. Ideally, all these approximations should be of the same order of magnitude. The approximation in the mathematical model is an input of the analysis: we can add uncertainties to the inputs and parameters of the program, that reflect this approximation, and analyze their propagation with Fluctuat. Additionally, when physical identities are known, that are true in the physical world, functional proof can be used to see whether these identities hold in the implementation, or else quantify the uncertainty or error on them: but in this case we may not distinguish the model error from the algorithm error. When one has access both to the implementation and the mathematical model, a guaranteed solution of the continuous model can be used to quantify the method error (and possibly propagate it with the analyzer as a special error). This is indeed part of what we started to do in the extensions of Fluctuat for hybrid systems, using a guaranteed ODE solver to bound the solutions of differential equations. Then the user can compare the orders of magnitude of the respective uncertainties on the model (propagation on the outputs of uncertainties on inputs and parameters), on the algorithm (functional property / physical identity), and on the finite precision computation (rounding error on the outputs). In most cases we have little latitude on the error due to finite precision: we can only decide to use either float or double precision numbers, depending on the adequacy between method and computation error. But in the general case, the accuracy of the algorithm can be adapted if necessary, for instance modifying a discretization step, the stopping criterion of an iterative algorithm, or even the level of approximation, so that for equivalent accuracy the computation time will be optimal.

Scientific computing programs present the interest for numerical verification concerning finite precision implementations, that (pen and paper) studies on algorithms in finite precision have long been conducted, see Higham's book [99] to mention only one reference, and can be used to validate the interest of our results. Theoretical results about what can and cannot be computed accurately and efficiently such as recently developed by Demmel [47], are also very valuable to guide our work. However, automatically getting accurate error bounds on credible programs is a difficult task, at which we still aim. For instance, we obtained some first results on the solution of a small linear systems with small perturbations, by a conjugate gradient algorithm, that confirmed (part) of the results of the mathematical studies on the behavior of such algorithms in finite precision as described in Meurant's book [125]. But automatizing such studies for real-scale simulation programs remains a long-term challenge for program verification.

Bibliography

[1] Standard for floating-point arithmetic (ieee 754-2008), http://dx.doi.org/10.1109/ieeestd.2008.4610935, 2008.

[2] A. Adjé, S. Gaubert, and E. Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. *Logical Methods in Computer Science*, 8(1), 2011.

[3] T. Alamo, J.M. Bravo, and E.F. Camacho. Guaranteed state estimation by zonotopes. *Automatica*, 41(6):1035 – 1043, 2005.

[4] X. Allamigeon, S. Gaubert, and E. Goubault. Inferring min and max invariants using max-plus polyhedra. In *Proceedings of SAS'08, Static Analysis Symposium*, volume 5079 of *Lecture Notes in Computer Science*, pages 189–204. Springer, 2008.

[5] M. Althoff, O. Stursberg, and M. Buss. Computing reachable sets of hybrid systems using a combination of zonotopes and polytopes. *Nonlinear analysis: hybrid systems*, 4(2):233–249, 2010.

[6] J.P. Aubin and H. Frankowska. *Set-Valued Analysis*. Birkhäuser, Boston, 1990.

[7] A. Ayad and C. Marché. Multi-prover verification of floating-point programs. In *Proceedings of IJCAR'10, 5th Joint Conference on Automated Reasoning*, volume 6173 of *Lecture Notes in Computer Science*, pages 127–141. Springer, 2010.

[8] P. Baudin, J.-C. Filliâtre, C. Marché, B. Monate, Y. Moy, and V. Prevosto. *ACSL: ANSI/ISO C specification language*, 2008.

[9] F.L. Bauer. Computational graphs and rounding error. *SIAM Journal of Numerical Analysis*, 11:87–96, 1974.

[10] D. Berleant and C. Goodman-Strauss. Bounding the results of arithmetic operations on random variables of unknown dependency using intervals. *Reliable Computing*, 4(2):147–165, May 1998.

[11] D. P. Bertsekas, A. Nedic, and A. E. Ozdaglar. *Convex Analysis and Optimization*. Athena Scientific, 2003.

[12] M. Berz. From taylor series to taylor models. In *AIP Conference Proceedings 405*, pages 1–23, 1997.

[13] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Design and implementation of a special-purpose static program analyzer for safety-critical real-time embedded software. In *The Essence of Computation: Complexity, Analysis, Transformation*, number 2566 in LNCS, pages 85–108. Springer, 2002.

[14] S. Boldo. Floats and ropes: a case study for formal numerical program verification. In *Proceedings of ICALP'09, Colloquium on Automata, Languages and Programming*, volume 5556 of *Lecture Notes in Computer Science*, pages 91–102. Springer, 2009.

[15] S. Boldo and J.-C. Filliâtre. Formal verification of floating-point programs. In *Proceedings of ARITH'07, 18th IEEE Symposium on Computer Arithmetic*, pages 187–194. IEEE Computer Society, 2007.

[16] S. Boldo, J.-C. Filliâtre, and G. Melquiond. Combining coq and gappa for certifying floating-point programs. In *Proceedings of Calculemus/MKM*, volume 5625 of *Lecture Notes in Computer Science*, pages 59–74. Springer, 2009.

[17] S. Boldo and C. Marché. Formal verification of numerical programs: from c annotated programs to mechanical proofs. *Mathematics in Computer Science*, 5:377–393, 2011.

[18] S. Boldo and G. Melquiond. Flocq: a unified library for proving floating-point algorithms in Coq. In *Proceedings of the 20th IEEE Symposium on Computer Arithmetic*, 2011.

[19] O. Bouissou, E. Conquet, P. Cousot, R. Cousot, J. Feret, K. Ghorbal, E. Goubault, D. Lesens, L. Mauborgne, A. Mine, S. Putot, and X. Rival. Space software validation using abstract interpretation. In *Proceedings of DASIA'09, Conference on Data Systems in Aerospace*, 2009.

[20] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic, and applications to static analysis of programs. In *Presented at SCAN 2010, 14th GAMM-IMACS International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*, 2010.

[21] O. Bouissou, E. Goubault, J. Goubault-Larrecq, and S. Putot. A generalization of p-boxes to affine arithmetic. *Computing*, 94(2-4):189–201, 2012.

[22] O. Bouissou, E. Goubault, S. Putot, K. Tekkal, and F. Védrine. Hybridfluctuat: a static analyzer of numerical programs within a continuous environment. In *Proceedings of CAV'09, 21st Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 620–626. Springer, 2009.

[23] O. Bouissou and M. Martel. GRKLib: a guaranteed runge-kutta library. In *Follow-up of International Symposium on Scientific Computing, Computer Arithmetic and Validated Numerics*. IEEE Press, 2007.

[24] N. Brisebarre and M. Joldeċs. Chebyshev interpolation polynomial-based tools for rigorous computing. In *Proceedings of ISSAC'10, International Symposium on Symbolic and Algebraic Computation*, pages 147–154. ACM, 2010.

[25] V. A. Carreño and P. S. Miner. Specification of the IEEE-854 floating-point standard in HOL and PVS. In *HOL95: 8th Workshop on Higher-Order Logic Theorem Proving and Its Applications*, Aspen Grove, UT, 1995.

[26] S. Chaudhuri, S. Gulwani, and R. Lublinerman. Continuity analysis of programs. In *Proceedings of POPL'10, 37th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 57–70, 2010.

[27] S. Chaudhuri, S. Gulwani, R. Lublinerman, and S. NavidPour. Proving programs robust. In *Proceedings of ISGSOLT/FSE'11, 9th ACM SIGSOFT Symposium on the Foundations of Software Engineering*, pages 102–112, 2011.

[28] L. Chen, A. Miné, and P. Cousot. A sound floating-point polyhedra abstract domain. In *Proceedings of APLAS'08, 6th Asian Symposium on Programming Languages and Systems*, volume 5356 of *Lecture Notes in Computer Science*, pages 3–18. Springer, 2008.

[29] L. Chen, A. Miné, J. Wang, and P. Cousot. Interval polyhedra: an abstract domain to infer interval linear relationships. In *Proceedings of SAS'09, 16th Static Analysis Symposium*, volume 5673 of *Lecture Notes in Computer Science*, pages 309–325. Springer, 2009.

[30] L. Chen, A. Miné, J. Wang, and P. Cousot. Linear absolute value relation analysis. In *Proceedings of ESOP'11, 20th European Symposium on Programming*, volume 6602 of *Lecture Notes in Computer Science*, pages 156–175. Springer, 2011.

[31] R. Clarisó and J. Cortadella. The octahedron abstract domain. In *Proceedings of SAS'04, 11th Symposium on Static Analysis*, volume 3148 of *Lecture Notes in Computer Science*, pages 312–327. Springer, 2004.

[32] H. Collavizza, M. Rueher, and P. Van Hentenryck. Cpbpv: a constraint-programming framework for bounded program verification. *Constraints*, 15(2):238–264, 2010.

[33] J. L. D. Comba and J. Stolfi. Affine arithmetic and its applications to computer graphics. In *Proc. SIBGRAPI'93 — VI Simpósio Brasileiro de Computaccão Gráfica e Processamento de Imagens (Recife, BR)*, pages 9–18, 1993.

[34] C. Combastel. A state bounding observer based on zonotopes. In *Proceedings of European control conference. Cambridge, UK*, 2003.

[35] C. Combastel. A state bounding observer for uncertain non-linear continuous-time systems based on zonotopes. In *Proceedings of 44th IEEE Conference on Decision and Control*, 2005.

[36] C. Combastel. On computing envelopes for discrete-time linear systems with affine parametric uncertainties and bounded inputs, 2011.

[37] A. Costan, S. Gaubert, E. Goubault, M. Martel, and S. Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *Proceedings of CAV'05, 17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 462–475. Springer, 2005.

[38] P. Cousot and R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proceedings of POPL'77, 4th ACM Symposium on Principles of Programming Languages*, pages 238–252. ACM Press, 1977.

[39] P. Cousot and R. Cousot. Comparison of the Galois connection and widening/narrowing approaches to abstract interpretation. JTASPEFL '91, Bordeaux. *JTASPEFT/WSA*, 74:107–110, 1991.

[40] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 2(4):511–547, 1992.

[41] P. Cousot and R. Cousot. Compositional separate modular static analysis of programs by abstract interpretation. In *Proceedings of SSGRR'01, Second International Conference on Advances in Infrastructure for E-Business, E-Science and E-Education on the Internet*. Scuola Superiore G. Reiss Romoli, 2001.

[42] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X Rival. The astrée analyzer. In *Proceedings of ESOP'05, 14th European Symposium on Programming*, volume 3444 of *Lecture Notes in Computer Science*, pages 21–30. Springer, 2005.

[43] P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. Combination of abstractions in the astrée static analyzer. In *Proceedings of ASIAN'06, 11th Asian Computing Science Conference*, volume 4435 of *Lecture Notes in Computer Science*, pages 272–300. Springer, 2006.

[44] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Proceedings of POPL'78, 5th ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM Press, 1978.

[45] L. H. de Figueiredo and J. Stolfi. Affine arithmetic: concepts and applications. *Numerical Algorithms*, 37:147–158, 2004.

[46] D. Delmas, E. Goubault, S. Putot, J. Souyris, K. Tekkal, and F. Védrine. Towards an industrial use of fluctuat on safety-critical avionics software. In *Proceedings of FMICS'09, 14th Workshop on Formal Methods for Industrial Critical Systems*, volume 5825 of *Lecture Notes in Computer Science*, pages 53–69. Springer, 2009.

[47] J. Demmel. The complexity of accurate floating-point computation. In *Proceedings of the 2002 International Congress of Mathematicians*, pages 697–706, 2002.

[48] C. Epstein, W.L. Miranker, and T.J. Rivlin. Ultra-arithmetic i: Function data types. *Mathematics and Computers in Simulation*, 24(1):1 – 18, 1982.

[49] J. Feret. Static analysis of digital filters. In David A. Schmidt, editor, *Proceedings of ESOP'04, 13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 33–48. Springer, 2004.

[50] J. Feret. The arithmetic-geometric progression abstract domain. In *Proceedings of VMCAI'05, 6th Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 42–58. Springer, 2005.

[51] S. Ferson. *RAMAS Risk Calc 4.0 software: risk assessment with uncertain numbers*. Lewis Publishers, Boca Raton, Florida, 2002.

[52] S. Ferson, V. Kreinovich, L. Ginzburg, D. Myers, and K. Sentz. Constructing probability boxes and dempster-shafer structures. Technical report, Sandia National Laboratories, 2003.

[53] S. Ferson, R.B. Nelsen, J. Hajagos, D.J. Berleant, J. Zhang, W.T. Tucker, L.R. Ginzburg, and W.L. Oberkampf. Dependence in probabilistic modelling, dempster–shafer theory and probability bounds analysis. Technical report, Sandia National Laboratories, 2004.

[54] P. C. Fischer. Automatic propagated and round-off error analysis. In ACM, editor, *Preprints of papers presented at the 13th national meeting of the Association for Computing Machinery*, 1958.

[55] L. Fousse, G. Hanrot, V. Lefvre, P. Plissier, and P. Zimmermann. Mpfr: A multiple-precision binary floating-point library with correct rounding. *ACM Trans. Math. Softw*, 33, 2007.

[56] M. J. Frank, R. B. Nelsen, and B. Schweizer. Best-possible bounds for the distribution of a sum a problem of kolmogorov. *Probability Theory and Related Fields*, 74:199–211, 1987.

[57] O. Friedmann. An exponential lower bound for the parity game strategy improvement algorithm as we know it. In *Proceedings of LICS'09, 24th Annual IEEE Symposium on Logic in Computer Science*, pages 145–156, 2009.

[58] IMTEC-92-26 GAO Report. Patriot missile defense, software problem led to system failure at dhahran, saudi arabia. Technical report, 1992.

[59] S. Gaubert, E. Goubault, A. Taly, and S. Zennou. Static analysis by policy iteration on relational domains. In *Proceedings of ESOP'07, 16th European conference on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 237–252. Springer, 2007.

[60] T. Gawlitza and H. Seidl. Precise fixpoint computation through strategy iteration. In *Proceedings of ESOP'07, 16th European conference on Programming*, volume 4421 of *Lecture Notes in Computer Science*, pages 300–315. Springer, 2007.

[61] T. Gawlitza and H. Seidl. Precise relational invariants through strategy iteration. In *Proceedings of CSL'07, 21st International Workshop on Computer Science Logic*, volume 4646 of *Lecture Notes in Computer Science*, pages 23–40. Springer, 2007.

[62] T. Gawlitza and H. Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *Proceedings of SAS'10, 17th Static Analysis Symposium*, volume 6337 of *Lecture Notes in Computer Science*, pages 271–286. Springer, 2010.

[63] K. Ghorbal. Static analysis of numerical programs: constrained affine sets, 2011.

[64] K. Ghorbal, E. Goubault, and S. Putot. The zonotope abstract domain taylor1+. In *Proceedings of CAV'09, 21st Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 627–633. Springer, 2009.

[65] K. Ghorbal, E. Goubault, and S. Putot. A logical product approach to zonotope intersection. In *Proceedings of CAV'10, 22nd Conference on Computer Aided Verification*, volume 6174 of *LNCS*, pages 212–226, 2010.

[66] A. Girard. Reachability of uncertain linear systems using zonotopes. In *Proceedings of HSCC'05, 8th International Workshop on Hybrid Systems: Computation and Control*, volume 3414 of *Lecture Notes in Computer Science*, pages 291–305. Springer, 2005.

[67] A. Girard and C. Le Guernic. Zonotope/hyperplane intersection for hybrid systems reachability analysis. In *Proceedings of HSCC'08, 11th Workshop on Hybrid Systems: Computation and Control*, volume 4981 of *Lecture Notes in Computer Science*, pages 215–228. Springer, 2008.

[68] D. Goldberg. What every computer scientist should know about floating-point arithmetic. *ACM Comput. Surv.*, 23(1):5–48, 1991.

[69] A. Goldsztejn. Modal intervals revisited: a mean-value extension to generalized intervals. In *Proceedings of QCP-2005, Quantification in Constraint Programming*, 2005.

[70] A. Goldsztejn. Modal intervals revisited part 1: a generalized interval natural extension. 2005.

[71] A. Goldsztejn and L. Jaulin. Inner approximation of the range of vector-valued functions. *Reliable Computing*, 1:23, 2010.

[72] L. Gonnord and N. Halbwachs. Combining widening and acceleration in linear relation analysis. In *Proceedings of SAS'06, 13th International Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 144–160. Springer, 2006.

[73] E. Goubault. Static analyses of the precision of floating-point operations. In *Proceedings of SAS'01, 8th Static Analysis Symposium*, volume 2126 of *Lecture Notes in Computer Science*, pages 234–259. Springer, 2001.

[74] E. Goubault, T. Le Gall, and S. Putot. An accurate join for zonotopes, preserving affine input/output relations. *Electronic Notes in Theoretical Computer Science*, 287:65–76, 2012. Proceedings of NSAD'12, 4th Workshop on Numerical and Symbolic Abstract Domains.

[75] E. Goubault, D. Guilbaud, A. Pacalet, B. Starynkévitch, and F. Védrine. A simple abstract interpreter for threat detection and test case generation. In *Proceedings of WAPATV'01 (ICSE'01)*, 2001.

[76] E. Goubault, T. Le Gall, S. Putot, and F. Védrine. Interactive analysis in fluctuat, 2012.

[77] E. Goubault, M. Martel, and S. Putot. Asserting the precision of floating-point computations: a simple abstract interpreter. In *Proceedings of ESOP'02, 11st European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 209–212. Springer, 2002.

[78] E. Goubault, M. Martel, and S. Putot. Static analysis-based validation of floating-point computations. In *Dagstuhl Seminar Numerical Software with Result Verification, LNCS*, volume 2991 of *Lecture Notes in Computer Science*, pages 306–313. Springer, 2003.

[79] E. Goubault, M. Martel, and S. Putot. Some future challenges in the validation of control systems. In *Proceedings of ERTS'06, Embedded Real-Time Systems*, 2006.

[80] E. Goubault and S. Putot. Weakly relational domains for floating-point computation analysis. In *Proceedings of NSAD'05, Workshop on Numerical and Symbolic Abstract Domains*, 2005.

[81] E. Goubault and S. Putot. Static analysis of numerical algorithms. In *Proceedings of SAS'06, 13th Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 18–34. Springer, 2006.

[82] E. Goubault and S. Putot. Under-approximations of computations in real numbers based on generalized affine arithmetic. In *Proceedings of SAS'07, 14th Static Analysis Symposium*, volume 4634 of *Lecture Notes in Computer Science*, pages 137–152. Springer, 2007.

[83] E. Goubault and S. Putot. Perturbed affine arithmetic for invariant computation in numerical program analysis. *CoRR*, abs/0807.2961, 2008.

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

[85] E. Goubault and S. Putot. Static analysis of finite precision computations. In *Proceedings of VMCAI'11, 12th Conference on Verification, Model Checking, and Abstract Interpretation*, volume 6538 of *Lecture Notes in Computer Science*, pages 232–247. Springer, 2011.

[86] E. Goubault and S. Putot. Superellipsoids: a generalization of the interval, zonotope and ellipsoid domains, 2011.

[87] E. Goubault, S. Putot, P. Baufreton, and J. Gassino. Static analysis of the accuracy in control systems: principles and experiments. In *Proceedings of FMICS'07, 12th Workshop on Formal Methods for Industrial Critical Systems*, volume 4916 of *Lecture Notes in Computer Science*, pages 3–20. Springer, 2007.

[88] E. Goubault, S. Putot, K. Tekkal, and F. Vedrine. Fluctuat user manual: static analysis of finite precision computations. Technical report, CEA, 2012.

[89] E. Goubault, S. Putot, and F. Vedrine. Modular static analysis with zonotopes. In *Proceedings of SAS'12, 19th Static Analysis Symposium*, volume 7460 of *Lecture Notes in Computer Science*, pages 24–40. Springer, 2012.

[90] L. J. Guibas, A. Nguyen, and L. Zhang. Zonotopes as bounding volumes. In *Proceedings of SODA'03, 14th ACM-SIAM Symposium on Discrete Algorithms*, pages 803–812, 2003.

[91] S. Gulwani and A. Tiwari. Combining abstract interpreters. In *Proceedings of PLDI'06, Conference on Programming Language Design and Implementation*, pages 376–386. ACM Press, 2006.

[92] E. Hansen. Interval arithmetic in matrix computations, part i. *Journal of the Society for Industrial and Applied Mathematics: Series B, Numerical Analysis*, 2(2):308–320, 1965.

[93] E. Hansen. On solving systems of equations using interval arithmetic. *Math. Comp*, 22:374–384, 1968.

[94] E. Hansen. A generalized interval arithmetic. In *Proceedings of the International Symposium on Interval Mathematics*, pages 7–18. Springer, 1975.

[95] E. Hansen. *Global optimization using interval analysis*. M. Dekker, New York, 1992.

[96] E. Hansen and R. Smith. Interval arithmetic in matrix computations, part ii. *SIAM Journal on Numerical Analysis*, 4(1):1–9, 1967.

[97] J. Harrison. Floating-point verification in hol. In *Proceedings of TPHOLs'95, 8th Workshop on Higher Order Logic Theorem Proving and Its Applications*, volume 971 of *Lecture Notes in Computer Science*, pages 186–199. Springer, 1995.

[98] D. Henrion and J. B. Lasserre. Inner approximations for polynomial matrix inequalities and robust stability regions. *CoRR*, abs/1104.4905, 2011.

[99] N. J. Higham. *Accuracy and stability of numerical algorithms*. Society for Industrial and Applied Mathematics, 1996.

[100] R. Howard. *Dynamic programming and Markov processes*. Wiley, 1960.

[101] B. Jeannet and al. Newpolka library. `http://www.inrialpes.fr/pop-art/people/bjeannet/newpolka`.

[102] B. Jeannet, D. Gopan, and T. Reps. A relational abstraction for functions. In *Proceedings of NSAD'05, Int. Workshop on Numerical and Symbolic Abstract Domains*, 2005.

[103] B. Jeannet and A. Miné. Apron: A library of numerical abstract domains for static analysis. In *Proceedings of CAV'09, 21st International Conference on Computer Aided Verification*, volume 5643 of *Lecture Notes in Computer Science*, pages 661–667. Springer, 2009.

[104] N. D. Jones and S. S. Muchnick. A flexible approach to interprocedural data flow analysis and programs with recursive data structures. In *Proceedings of POPL'82, 9th Symposium on Principles of Programming Languages*, pages 66–74. ACM, 1982.

[105] W. Kahan. The improbability of probabilistic error analyses for numerical computations, 1998.

[106] E. Kaucher. Interval analysis in the extended interval space ir. *Computing*, Suppl. 2:33–49, 1980.

[107] W. Kühn. Zonotope dynamics in numerical quality control. In *Visualization and Mathematics*, pages 125–134. Springer, 1998.

[108] A. B. Kurzhanski and I. Valyi. Ellipsoidal techniques for dynamic systems: the problem of control synthesis. *Dynamics and Control*, 1:357–378, 1991.

[109] Alexander B Kurzhanski and Pravin Varaiya. Ellipsoidal techniques for reachability analysis: internal approximation. *Systems Control Letters*, 41(3):201–211, 2000.

[110] A. B. Kurzhanskii and I. Vlyi. *Ellipsoidal calculus for estimation and control*. Systems and control : foundations and applications. Birkhuser, Boston, MA, 1997.

[111] W. Landi and B. G. Ryder. A safe approximate algorithm for interprocedural aliasing. In *Proceedings of PLDI'92, ACM SIGPLAN Conference on Programming language design and implementation*, pages 235–248. ACM, 1992.

[112] P. Langlois. Automatic linear correction of rounding errors. *BIT*, 41(3):515–539, 2001.

[113] P. Langlois and F. Nativel. Reduction and bounding of the rounding error in floating-point arithmetic. *C.R. Acad. Sci. Paris, Série 1*, 327:781–786, 1998.

[114] P. Langlois and F. Nativel. When automatic linear correction of rounding errors is exact. *C.R. Acad. Sci. Paris, Série 1*, 328:543–548, 1999.

[115] J. Larson and A. Sameh. Efficient calculation of the effects of roundoff errors. *ACM Trans. Math. Softw.*, 4(3):228–236, 1978.

[116] V. Laviron and F. Logozzo. Subpolyhedra: a (more) scalable approach to infer linear inequalities. In *Proceedings of VMCAI'09, 10th Conference on Verification, Model Checking, and Abstract Interpretation*, volume 5403 of *Lecture Notes in Computer Science*, pages 229–244. Springer, 2009.

[117] D.-U. Lee, A. Abdul Gaffar, W. Luk, and O. Mencer. MiniBit: Bit-width optimization via affine arithmetic. In *Proceedings of DAC'05, Design Automation Conference*, pages 837–840, 2005.

[118] S. Linnainmaa. Taylor expansion of the accumulated rounding error. *BIT Numerical Mathematics*, 16:146–160, 1976.

[119] F. Logozzo and M. Fähndrich. Pentagons: a weakly relational abstract domain for the efficient validation of array accesses. In *Proceedings of SAC'08, ACM Symposium on Applied Computing*, pages 184–188. ACM Press, 2008.

[120] K. Makino and M. Berz. Taylor models and other validated functional inclusion methods. *International Journal of Pure and Applied Mathematics*, 4(4):379–456, 2003.

[121] Matthieu Martel. Propagation of roundoff errors in finite precision computations: a semantics approach. In *Proceedings of ESOP'02, 11th European Symposium on Programming*, volume 2305 of *Lecture Notes in Computer Science*, pages 194–208. Springer, 2002.

[122] J. Mascunan. Analyse statique de programmes en virgule fixe, 2005.

[123] D. Menard, R. Rocher, and O. Sentieys. Analytical fixed-point accuracy evaluation in linear time-invariant systems. *IEEE Transactions on Circuits and Systems I: Regular Papers*, 55(10):3197–3208, 2008.

[124] D. Menard, R. Serizel, R. Rocher, and O. Sentieys. Accuracy constraint determination in fixed-point system design. *EURASIP J. Embedded Syst.*, pages 1:1–1:12, 2008.

[125] G. Meurant. *The lanczos and conjugate gradient algorithms; from theory to finite precision computation*. SIAM, 2006.

[126] C. Michel, M. Rueher, and Y. Lebbah. Solving constraints over floating-point numbers. In *Proceedings of CP'01, 7th Conference on Principles and Practice of Constraint Programming*, volume 2239 of *Lecture Notes in Computer Science*, pages 524–538. Springer, 2001.

[127] W. Miller. Software for roundoff analysis. *Transactions of the ACM on Mathematical Software*, 1:108–128, 1975.

[128] A. Miné. A new numerical abstract domain based on difference-bound matrices. In *Proceedings of PADO, 2nd Symposium on Programs as Data ObjectsO*, volume 2053 of *Lecture Notes in Computer Science*, pages 155–172. Springer, 2001.

[129] A. Miné. The octagon abstract domain. In *Proceedings of WCRE'10, Eighth Working Conference on Reverse Engineering*, pages 310–, 2001.

[130] A. Miné. Relational abstract domains for the detection of floating-point run-time errors. In *Proceedings of ESOP'04, 13th European Symposium on Programming*, volume 2986 of *Lecture Notes in Computer Science*, pages 3–17. Springer, 2004.

[131] D. Monniaux. Compositional analysis of floating-point linear numerical filters. In *Proceedings of CAV'05, 17th International Conference on Computer Aided Verification*, volume 3576 of *Lecture Notes in Computer Science*, pages 199–212. Springer, 2005.

[132] D. Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3):1–41, 2008.

[133] R. E. Moore. *Interval analysis*. Prentice-Hall, Englewood Cliffs N. J., 1966.

[134] R.E. Moore. Automatic error analysis in digital computation. Technical report space div. report lmsd84821, Lockheed Missiles and Space Co., 1959.

[135] O. Mullier, E. Goubault, M. Kieffer, and S. Putot. Under-approximation of the range of vector-valued functions extended. In *SWIM'11 Small Workshop on Interval Methods, June 14-15, 2011*, 2011.

[136] A. Neumaier. The wrapping eefect, ellipsoid arithmetic, stability and confidence regions, 1993.

[137] A. I. Ovseevich and F. L. Chernousko. On optimal ellipsoids approximating reachable sets. *Problems of Control and Information Theory*, 16:125–134, 1987.

[138] O. Ponsini, C. Michel, and M. Rueher. Refining abstract interpretation based value analysis with constraint programming techniques. In *Proceedings of CP'2012, Conference on Principles and Practice of Constraint Programming*, volume 7514 of *Lecture Notes in Computer Science*, pages 593–607. Springer, 2012.

[139] V. Pratt. Anatomy of the pentium bug. In *TAPSOFT '95: Theory and Practice of Software Development, volume 915 of Lecture Notes in Computer Science*, pages 97–107, 1995.

[140] PPL project. The Parma Polyhedra Library. `http://www.cs.unipr.it/ppl/`.

[141] H. M. Regan, S. Ferson, and D. Berleant. Equivalence of methods for uncertainty propagation of real-valued random variables. *Int. J. Approx. Reasoning*, 36(1):1–30, 2004.

[142] RTCA. Do-178b, software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission for Aeronautics, 1992.

[143] RTCA. Do-178c, software considerations in airborne systems and equipment certification. Technical report, Radio Technical Commission for Aeronautics, 2011.

[144] S. M. Rump. Verification methods: rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, 2010.

[145] D. M. Russinoff. A mechanically checked proof of IEEE compliance of the floating point multiplication, division and square root algorithms of the AMD-K7 processor. *LMS Journal of Computation and Mathematics*, 1:148–200, 1998.

[146] S. Sankaranarayanan. Automatic abstraction of non-linear systems using change of bases transformations. In *Proceedings of HSCC'11, 14th Conference on Hybrid Systems: Computation and Control*, HSCC '11, pages 143–152. ACM Press, 2011.

[147] S. Sankaranarayanan, H. B. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In *Proceedings of VMCAI'05, 6th Conference on Verification, Model Checking, and Abstract Interpretation*, volume 3385 of *Lecture Notes in Computer Science*, pages 25–41. Springer, 2005.

[148] D. A. Schmidt. A calculus of logical relations for over- and underapproximating static analyses. *Sci. Comput. Program.*, 64(1):29–53, 2007.

[149] D.A. Schmidt. Underapproximating predicate transformers. In *Proceedings of SAS'06, 13th International Static Analysis Symposium*, volume 4134 of *Lecture Notes in Computer Science*, pages 127–143. Springer, 2006.

[150] P. Schrammel and B. Jeannet. Extending abstract acceleration methods to data-flow programs with numerical inputs. *Electronic Notes in Theoretical Computer Science*, 267(1):101 – 114, 2010. Proceeding of NSAD'10, Second International Workshop on Numerical and Symbolic Abstract Domains.

[151] G. Shafer. *A Mathematical theory of evidence*. Princeton University Press, Princeton, 1976.

[152] A. Simon, A. King, and J. M. Howe. Two variables per linear inequality as an abstract domain. In *Proceedings of LOPSTR'02, 12th Workshop on Logic Based Program Synthesis and Tranformation*, volume 2664 of *Lecture Notes in Computer Science*, pages 71–89. Springer, 2002.

[153] P. Sotin, B. Jeannet, F. Védrine, and E. Goubault. Policy iteration within logico-numerical abstract domains. In *Proceedings of ATVA'11, 9th International Symposium on Automated Technology for Verification and Analysis*, volume 6996 of *Lecture Notes in Computer Science*, pages 290–305. Springer, 2011.

[154] P. H. Sterbenz. Floating point computation. Prentice Hall, 1974.

[155] J. Vignes. Review on stochastic approach to round-off error analysis and its applications. *Mathematics and Computers in Simulation*, 30(6):481–491, 1988.

[156] J. von Neumann and H. H. Goldstine. Numerical inverting of matrices of high order. *Bulletin of the AMS*, 53(11):1021–1099, 1947.

[157] J. H. Wilkinson. *Rounding errors in algebraic processes*. Englewood Cliffs, New Jersey: Prentice Hall, 1963.

[158] R. C. Williamson and T. Downs. Probabilistic arithmetic I: numerical methods for calculating convolutions and dependency bounds. *Journal of Approximate Reasoning*, 1990.

[159] L. Zhang, Y. Zhang, and W. Zhou. A fast and flexible accuracy-guaranteed fractional bit-widths optimization approach. In *Proceedings of ISCAS'09, IEEE International Symposium on Circuits and Systems*, pages 1517 –1520, 2009.

[160] G. M. Ziegler. *Lectures on Polytopes (updated seventh printing)*. Number 152 in Graduate Texts in Mathematics. Springer-Verlag, 2007.