Université de Toulouse

# THÈSE

**En vue de l'obtention du**

## DOCTORAT DE L'UNIVERSITÉ DE TOULOUSE

**Délivré par :**

Institut Supérieur de l'Aéronautique et de l'Espace (ISAE)

---

**Présentée et soutenue par :**
**Pierre Roux**
**le** mercredi 18 décembre 2013
**Titre :**

# Analyse statique de systèmes de contrôle commande : synthèse d'invariants non linéaires

---

**École doctorale et discipline ou spécialité :**
ED MITT : Sureté de logiciel et calcul de haute performance

**Unité de Recherche :**
Équipe d'accueil ISAE-ONERA MOIS

**Directeurs de Thèse :**
Mme Virginie Wiels (directrice de thèse)
M. Pierre-Loïc Garoche (co-directeur de thèse)

**Jury :**

| | | |
|---|---|---|
| M. David Monniaux | DR CNRS, Verimag, Grenoble | Rapporteur |
| M. Helmut Seidl | Prof. TU München, Munich | Rapporteur |
| M. Éric Féron | Prof. Georgia Tech, Atlanta | Examinateur |
| M. Éric Goubault | DR CEA, LIST, Palaiseau | Examinateur |
| M. Didier Henrion | DR CNRS, LAAS, Toulouse | Examinateur |
| M. César Muñoz | NASA, LaRC, Hampton | Examinateur |
| Mme Virginie Wiels | MR ONERA, DTIM, Toulouse | Directrice de thèse |
| M. Pierre-Loïc Garoche | IR ONERA, DTIM, Toulouse | Co-directeur de thèse |

# Remerciements

Voici venu le moment de remercier formellement les nombreuses personnes sans qui cette thèse n'existerait pas ou tout au moins ne serait pas celle que vous tenez actuellement entre les mains — ou celle affichée par votre lecteur PDF. Je tenterai par la suite de procéder par ordre chronologique inverse, mais cet ordre n'étant ni total ni formellement défini, j'y ferai quelques entorses.

David Monniaux et Helmut Seidl ont immédiatement accepté d'être rapporteurs de cette thèse. J'en suis très honoré et les remercie pour leurs remarques toujours très pertinentes. Éric Féron, Éric Goubault, Didier Henrion et César Muñoz m'ont fait l'honneur de compléter ce jury en tant qu'examinateurs. Que tous soient ici remerciés pour le temps consacré à mon travail, d'autant plus que cela a souvent nécessité de traverser quelques continents ou océans.

Je suis très redevable à Pierre-Loïc Garoche pour l'encadrement de cette thèse. Il a toujours su m'encourager dans ce travail et son optimisme sans faille a souvent été le bienvenu. Même s'il n'a pas toujours été physiquement présent il s'avérait généralement très réactif, au point de m'en faire parfois perdre toute notion du décalage horaire entre les deux côtés de l'Atlantique. J'ai également apprécié son incroyable efficacité lors des quelques occasions que nous avons eues de coder ensemble, malgré nos conceptions sensiblement différentes en termes de développement logiciel. D'un point de vue moins scientifique, les quelques années passées dans le même bureau furent très agréables. Enfin et surtout il a toujours su par sa curiosité et sa capacité à interagir avec des chercheurs de disciplines parfois très différentes, bien que connexes, être une inépuisable source d'inspiration, de sorte que j'ai souvent eu l'impression d'avoir plusieurs coups de retard entre ce qu'on aurait souhaité faire et ce que j'avais effectivement réalisé. Un grand merci enfin à ma directrice de thèse Virginie Wiels qui a toujours été présente pour répondre à mes, parfois nombreuses, questions administratives et réparer mes quelques gaffes en la matière. J'espère que le fait de ne pas avoir pu prendre une part plus importante dans l'encadrement scientifique de cette thèse ne se sera pas avéré trop frustrant.

Assalé Adjé a bien sûr joué un rôle clef pour les travaux du présent manuscrit en tant qu'auteur de l'itération sur les min politiques pour les gabarits quadratiques. Mais je lui suis également reconnaissant pour le dernier mois de cette thèse durant lequel nous avons partagé un même bureau alors qu'il commençait un postdoc dans la droite ligne de ce travail. Je garderai un bon souvenir de ces quelques dernières semaines et je ne peux que me réjouir de voir le travail poursuivi par quelqu'un de si compétent, tout en espérant avoir l'occasion d'y prendre part, fût-ce de manière mineure.

Éric Féron, en me faisant découvrir l'automatique et les fonctions de Lyapunov quadratiques, a permis à cette thèse de prendre un virage radical au

moment même où elle commençait à s'embourber dans son sujet initial, certes passionnant mais peut-être légèrement trop large ou ambitieux. Qu'il me soit donc permis de lui exprimer ici toute ma gratitude pour ce rôle absolument essentiel au succès de cette thèse. Je suis également redevable à ses thésards Romain Jobredeaux et Tim Wang pour les rudiments d'automatique qu'ils sont parvenus à m'inculquer et pour les nombreux échanges que nous avons eu à ce sujet. J'espère que nous trouverons à l'avenir l'occasion de nouvelles collaborations toutes aussi fructueuses. Enfin, merci au reste de l'équipe ainsi qu'à Vivek et Etienne pour l'accueil chaleureux[1] lors de ces semaines à Atlanta.

Merci aux doctorants, stagiaires et postdocs du DTIM, même si certains m'ont bien enfumé (heureusement uniquement au sens propre du terme), pour les déjeuners passés ensembles. Ces repas ont parfois été le lieu de discussions enflammées, même si certains sujets y étaient un peu trop récurrents à mon goût (désolé d'être toujours aussi incompétent en foot en particulier[2]). La situation géographique de mon bureau m'a également conduit à participer à de nombreuses pauses café avec les permanents de l'étage, j'en garderai le souvenir de moments à la fois instructifs et divertissants oscillant entre les sujets les plus sérieux et les plus loufoques et improbables. Merci à tous pour la bonne ambiance qui règne dans ce laboratoire. Un merci tout particulier à Claire Pagetti, entre autre pour avoir servi de cobaye pour mon cours d'initiation à l'interprétation abstraite et pour ses conseils pertinents pour ma visite de Hanoï.

J'ai eu au cours de cette thèse la chance de participer à quelques enseignements au sein de l'ENSEEIHT. J'ai particulièrement apprécié ma participation aux modules de programmation fonctionnelle et impérative, d'outils mathématiques pour l'informatique et de vérification par analyse statique. Merci aux directeurs de département et aux responsables de modules de m'avoir donné l'occasion d'effectuer ces enseignements, et de m'avoir renouvelé leur confiance plusieurs années consécutives, ainsi qu'aux équipes pédagogiques et aux élèves qui y ont pris part. Ce fut dans l'ensemble une expérience fort agréable.

On pourrait facilement les oublier et ce serait peut être le meilleur des remerciements. De nombreux personnels administratifs ont permis à cette thèse de se dérouler dans de bonnes conditions. Je tiens à remercier tout particulièrement les assistantes du DTIM : Yvonne Lebreton, Lyne Moron et Josette Brial, le service des thèses de l'ISAE en les personnes d'Annie Carles-Bailhe, de Maryse Herbillon et d'Isabelle Zanchetta ainsi que les secrétariats de l'école doctorale : Martine Labruyère et Agnès Requis et du département d'informatique de l'n7 : Blandine Volpato et Muriel De Guibert. Même si l'essentiel des ces « personnels de soutien » effectue un travail extraordinaire, ce n'est pas toujours le cas. Je ne remercie pas les personnes qui se sont avérées incapables d'assurer tant l'efficacité que même la sécurité des déplacements au sein et aux abords de leurs établissements ou qui ont laissé durer plusieurs années une procédure administrative pourtant connue et déjà effectuée à plusieurs reprises.

Bien que sur un sujet légèrement différent de celui de cette thèse, j'ai pu effectuer au cours de ma première année de master un stage au National Institute of Aerospace à Hampton en Virginie. Un grand merci à Radu Siminiceanu pour l'encadrement de ce stage, qui fut pour moi une formidable introduction au model checking symbolique, et pour m'avoir réinvité un an et demi plus tard,

[1] Particulièrement bienvenu avec la climatisation réglée « à l'américaine ».

[2] Et merci à Marc d'avoir à nouveau apporté un peu de culture aéronautique, ça manquait cruellement depuis le départ de Florian.

peu avant le début de cette thèse. Merci également à Alwyn Goodloe, César Muñoz et Gilles Dowek, ainsi qu'à Heber, Mark, Paolo, Taylor, Romain, Vivek et les autres, malheureusement trop nombreux pour être cités tous, pour l'accueil et les moments passés ensemble lors de ces visites au NIA. Merci enfin à l'efficace personnel administratif du NIA grâce auquel ces visites et leur préparation se sont toujours déroulées dans les meilleures conditions.

Cette thèse ne serait bien sûr pas ce qu'elle est sans les années d'étude qui l'ont précédée. J'exprime donc toute ma reconnaissance aux enseignants qui ont su me transmettre leurs connaissances et leur passion pour leur discipline. Parmi eux, je pense tout particulièrement à Helmut Seidl, en particulier pour son cours sur l'itération sur les politiques lors de l'école d'été Marktoberdorf 2011, à Daniel Hirschkoff, tant pour son cours de programmation en L3 que pour son rôle de responsable de la première année de master à l'ENS Lyon, à Jacques Sauloy qui m'a définitivement convaincu de faire de l'informatique en me montrant dans son cours de spé que c'était aussi une discipline théorique avec de beaux résultats mathématiques, à M. Devalette pour m'avoir fait découvrir et apprécier la programmation lors de ma première année de lycée ainsi qu'à Alain Lavignolle et Véronique Lizan pour m'avoir, au cours des ateliers Maths en Jeans qu'ils ont encadré au sein de mon lycée, donné goût à la recherche et montré que les mathématiques n'étaient pas qu'une série de petits exercices d'application du cours. Je dois également remercier l'État français de m'avoir permis d'effectuer, en particulier en les finançant, les susdites études, alors même que mon milieu social d'origine ne m'y prédisposait pas exceptionnellement. J'ai tout spécialement apprécié les – trop courtes – années passées dans le cadre exceptionnel de l'ENS Lyon et j'espère que de nombreuses générations d'étudiants pourront en profiter encore longtemps.

Bien sûr, rien de cela n'aurait été possible sans le soutien indéfectible de ma famille, mes parents et ma soeur qui m'ont toujours accompagné dans mes choix de formation, même si ceux-ci étaient très différents des leurs. Pour ce soutien ainsi que pour l'éducation et la culture qu'ils ont su me transmettre, je leur exprime toute ma gratitude.

Finalement, et c'est peut être le plus important, je remercie mes amis, mes enseignants de musique et camarades d'orchestre et les membres du Simu (que je regrette de ne pas avoir pu visiter plus souvent) pour avoir réussi à me distraire et m'arracher de mon travail au cours de ces années. Cela n'a pas toujours été un exercice facile pour eux, je leur en suis d'autant plus reconnaissant. La musique, les jeux de plateau, les randonnées ou les simples moments de discussion partagés avec eux occupent une place privilégiée dans mes souvenirs. Nombre d'entre eux ont déjà été évoqués, sinon nommés, mais je remercie tout particulièrement Antoine, Romain et Sébastien mais surtout les HEURE et sympathisants RATON-LAVEUR : Aisling, Benjamin, Benoît, Élodie, Gabriel, Guilhem, Guillaume, Hélène, Irène, Jonas, Julien, Laetitia, Mathilde, Mickaël, Mikaël, Nathanaël, Nicolas, Ophélia, Quentin, Sébastien et Tahina.

De même qu'un logiciel développé de manière non formelle contient presque sûrement des bugs, je peux être raisonnablement assuré que ces remerciements contiennent des omissions malheureuses. Je propose plusieurs solutions aux victimes : (1) inscrire votre nom ici ..... sur votre bel exemplaire papier ; (2) m'écrire dans l'espoir que je corrige l'erreur dans la version électronique disponible sur ma page web ; (3) m'envoyer un mail d'insultes. Étant bien entendu que ces solutions ne sont pas exclusives et que je préfère (1) ou (2).

# Contents

*English version*[3] *begins on page 3.*

# Systèmes critiques de contrôle commande

Les commandes de vol numériques sont maintenant un standard dans les grands avions civils. Ces systèmes de contrôle commande sont dit critiques dans la mesure où une défaillance pourrait avoir de graves conséquences, en particulier en termes de vies humaines, d'où les exigences de certification draconienne auxquelles elles sont soumises et un fort intérêt pour la preuve formelle de correction. Ce chapitre présente tout d'abord un bref rappel historique sur l'introduction des systèmes de contrôle commande dans la conception des aéronefs. Après quoi, le lecteur informaticien pourra trouver un panorama très succinct de ce à quoi un système de contrôle commande peut ressembler, quelques mots sur leur aspect critique, d'où le besoin de preuves et finalement le type de propriétés qui doivent être prouvées. Enfin sont introduites les contributions qui apparaîtront dans le reste du document.

**Historiquement,** depuis le premier vol du Wright Flyer en 1903, les pilotes contrôlaient leurs avions par des câbles liant mécaniquement leurs commandes (manche à balai et palonnier) aux gouvernes (ailerons pour l'axe longitudinal (roulis)[4], gouvernes de profondeur pour l'axe latéral (tangage) et gouverne de direction pour l'axe vertical (lacet)) qui entraînent des mouvements de l'avion en perturbant le flot d'air autour. Avec ce système, plus l'avion est gros, plus les gouvernes sont grandes et plus la force à développer par le pilote pour agir sur elles est importante, bien que des gouvernes bien équilibrées (avec des masselottes) et des flettner (petites gouvernes sur le bord des gouvernes principales bougeant dans la direction opposée pour aider à les bouger avec moins d'effort humain)[5] permettent d'économiser les forces du pilote. Les petits appareils d'aviation

[3] This is just a french translation of the first part, to meet some painful french regulation.

[4] Bien que, pour être précis, le Wright Flyer n'était pas équipé d'ailerons (surfaces articulées) mais utilisait un système de déformation de l'aile pour le contrôle longitudinal. Ce système a encore été utilisé sur quelques avions en bois et toile (comme le Blériot XI qui a effectué la première traversée de la Manche) dont les ailes étaient assez flexibles mais a rapidement été abandonné avec l'arrivée de structures plus rigides. Assez paradoxalement, une technique similaire est à nouveau étudiée aujourd'hui par des équipes de la NASA.

[5] Ils ont été mis au point par l'ingénieur allemand Anton Flettner, d'où leur nom.

générale utilisent toujours ce genre de mécanisme[6] mais cela devint un obstacle majeur au développement d'avions commerciaux plus grand dans la deuxième moitié du vingtième siècle. Des vérins hydrauliques ont ensuite été installés pour aider les pilotes à bouger les gouvernes, de manière similaire aux directions assistées introduites ensuite dans les voitures.

Cela a ensuite permis d'introduire un calculateur entre les commandes du pilote et les actuateurs. Le premier avion commercial équipé fut le supersonique franco anglais Concorde, qui prit l'air pour la première fois en 1969, tandis que les premiers appareils militaires produits en série avec des commandes de vol électriques ont été les F15, F16 et Mirage 2000 dans les années 1970. Ces commandes de vol électriques permettent de réaliser des avions intrinsèquement instables qu'un humain serait autrement incapable de piloter. Dans le domaine des avions de combat, elles ont donné naissance à des avions incroyablement manoeuvrables, tandis qu'assouplir la stabilité des avions commerciaux permet de réduire la taille de leurs gouvernes, diminuant à la fois leur poids et leur traînée ce qui conduit à une diminution de la consommation en carburant. Les calculateurs de commandes de vol embarqués dans ces appareils étaient toutefois analogiques[7]. En 1987, moins de vingt ans après Concorde, l'Airbus A320 réalisa son premier vol, devenant le premier avion commercial équipé de commandes de vol numériques[8]. Boeing suivit le même chemin moins de dix ans plus tard avec le premier vol de son B777 en 1994 et de nos jours la conception d'un grand avion commercial ne peut plus s'envisager sans commandes de vol numériques. Elles commencent même à être introduites dans les avions d'affaire avec le Dassault Falcon 7X, qui vola pour la première fois en 2005, et dans les hélicoptères avec le NH90 en 2003[9].

Les commandes de vol numériques peuvent permettre une meilleure précision numérique mais accroissent surtout la flexibilité du système en remplaçant le matériel spécifiquement développé des calculateurs analogiques par du logiciel tournant sur des processeurs du commerce. Ceci permet l'ajout de nouvelles fonctionnalités parmi lesquelles la plus remarquable est probablement la protection de l'enveloppe de vol. Ce système empêche l'avion d'être mis dans une situation dangereuse quoi que fasse le pilote avec ses commandes. En particulier, cela leur permet de réagir rapidement dans une situation d'urgence sans avoir à s'inquiéter de dépasser les limites structurelles ou aérodynamiques de leur avion. Grâce à leur protection de l'enveloppe de vol, on dit parfois des Airbus modernes qu'ils ne peuvent pas décrocher[10]. Ceci est parfois critiqué pour l'effet pervers suivant : puisque l'avion ne peut pas décrocher, on entraîne les équipages à pousser les gaz et tirer le manche quand ils approchent des conditions de décrochage, de façon à éviter le sol. Toutefois, dans de rare cas, ça peut être la pire chose à faire.

[6] C'est même le cas du biturbopropulseur franco italien ATR, l'avion le plus économique en carburant dans sa gamme (vols régionaux court courrier) de part sa relative rusticité.

[7] Bien que le calculateur régulant les entrées d'air du Concorde utilisait déjà une technologie numérique.

[8] La navette spatiale Entreprise vola pour la première fois (en tant que planeur) avec un système de contrôle numérique dix ans plus tôt en 1977 mais ne peut pas vraiment être considérée comme un avion commercial.

[9] Bien que l'hélicoptère vola pour la première fois en 1995 avec des commandes de vol mécaniques classiques du fait de graves retards dans le développement des commandes de vol numériques.

[10] Quand l'angle d'attaque d'une aile augmente, le flux d'air sur sa partie supérieure peut commencer à s'en détacher, induisant une soudaine perte de portance. Cela peut résulter en de sérieuses pertes d'altitude de l'avion qui peut alors frapper le sol s'il en était trop proche.

La perte du vol AF447 et de ses 228 occupants est attribuée à cet effet. Après la perte de tous les capteurs de vitesse (les tubes de Pitot ayant gelé), il devint impossible de conserver la protection de l'enveloppe de vol et les commandes de vol ont basculé dans une mode alternatif dans lequel les pilotes avaient un contrôle plus direct des gouvernes. Volant à haute altitude où la marge entre les conditions de vol normales, la survitesse (qui pourrait finalement détruire l'avion) et le décrochage devient très étroite[11], l'équipage, n'étant pas préparé à une telle situation, a rapidement fait décrocher l'avion qui a finalement péri dans l'océan. Toutefois, mis à part de tels événements exceptionnels, la protection de l'enveloppe de vol a certainement déjà sauvé de nombreuses vies humaines.

Les calculateurs numériques se sont maintenant répandus dans d'autres systèmes des avions comme les freins[12] ou le contrôle des moteurs[13]. Un dernier exemple de comportement inattendu par l'équipage est survenu lors des essais de réception d'un Airbus A340-600. Au cours d'essais moteur au sol, l'avion a commencé à bouger, l'équipage a alors tenté à la fois de freiner et de tourner pour éviter un mur de déflexion en béton qui entourait la zone d'essais. Malheureusement, le fait de tourner désactivait les freins de la partie centrale du train d'atterrissage principal de façon à économiser les pneus et l'avion n'a pas été capable de s'arrêter avant le mur et a été détruit. Assez paradoxalement, si les personnes à bord étaient allé droit dans le mur, l'avion aurait éventuellement pu s'arrêter avant de le rencontrer ou l'aurait au moins frappé à une vitesse moindre. Encore une fois, dans des conditions normales, ce système a probablement parfaitement rempli son rôle d'économiser les pneus.

**Les systèmes de contrôle commande** sont conçus par des automaticiens. Avant de développer un contrôleur ou régulateur, ils construisent d'abord un modèle du système physique qu'ils veulent réguler. Ils l'appellent *plant* et cela peut être soit un système intrinsèquement stable dont la stabilisation n'est pas jugée suffisamment rapide ou douce[14] ou un système véritablement instable qu'ils souhaitent stabiliser[15]. Cela peut se faire en appliquant les lois fondamentales de la mécanique au système et éventuellement en ajustant le résultat en le comparant au comportement du système réel. Le modèle qui en résulte est ensuite généralement linéarisé autour d'un point de fonctionnement intéressant[16]. En pratique, cela donne des matrices $A_p \in \mathbb{R}^{n\times n}$ et $B_p \in \mathbb{R}^{n\times p}$ telles que l'état

[11] Les jets modernes croisent à haute altitude (plus de 10000 m) où la densité de l'atmosphère est plus faible de façon à réduire la traînée donc la consommation e carburant. Il y a également moins de turbulences à de telles altitude ce qui améliore grandement le confort des passagers.

[12] La toute dernière fonctionnalité, nommée « brake to vacate » permet même aux pilotes de sélectionner une sortie de piste sur un plan de l'aéroport avant l'atterrissage, laissant le système contrôler le freinage de façon à optimiser le confort des passagers et minimiser l'usure des disques de frein.

[13] Généralement appelé FADEC pour contrôle moteur numérique à pleine autorité (Full Authority Digital Engine Control en anglais), il est aussi un élément clef dans l'efficacité énergétique des moteurs modernes en réalisant un contrôle fin de leurs nombreux paramètres qui serait difficile à atteindre même pour les meilleurs mécanicien navigants.

[14] Certains avions commerciaux ou le crochet d'une grue suspendu sous un câble qui se stabilise automatiquement avec un câble vertical sous son propre poids par exemple.

[15] Par exemple un avion de chasse ou un pendule inversé qui peut être maintenu vertical à l'envers en bougeant attentivement sa base (ce qui trouve des applications par exemple dans le Segway).

[16] Typiquement un choix d'altitude, vitesse, poids et centrage pour un avion ou la position verticale pour le pendule inversé.

interne $x_p \in \mathbb{R}^n$ du système[17] suit l'équation différentielle

$$\dot{x}_p = A_p x_p + B_p u_p$$

où $u_p \in \mathbb{R}^p$ est l'entrée du système[18]. À partir de cet état interne, une sortie $y_p$ est calculée :

$$y_p = C_p x_p + D_p u_p.$$

En partant de ce modèle du système physique, les automaticiens utilisent diverses méthodes pour développer un contrôleur. Dans le cas habituel des contrôleurs linéaires, ils définissent des matrices $A_c$ et $B_c$ telles que l'état interne $x_c$ du contrôleur suive l'équation

$$\dot{x}_c = A_c x_c + B_c u_c$$

où $u_c$ est l'entrée du contrôleur, venant typiquement des capteurs en entrée et des commandes[19] $y_d$ et égal à la sortie du système physique moins la commande $(y_p - y_d)$. Une sortie est ensuite calculée

$$y_c = C_c x_c + D_c u_c.$$

allant typiquement aux actuateurs et égale à l'entrée $u_p$ du système physique. Historiquement, quand les seuls outils de calcul disponibles étaient des règles à calcul, des abaques et des assistants humains[20], le développement de contrôleurs se faisait en utilisant des outils graphiques tels les diagrammes de Bode par exemple. Cet ensemble d'outils est connu comme l'*automatique classique*. Avec l'avènement des ordinateurs électroniques, il devint possible d'utiliser des procédures d'optimisation qui donnèrent naissance à des techniques tels les régulateurs LQR ou $H_\infty$ qui peuvent permettre d'atteindre de meilleures performances[21]. Ces techniques sont connues sous le nom d'*automatique moderne*.

Finalement, le contrôleur doit être implémenté. Cela peut se faire mécaniquement[22], avec un circuit électronique analogique comme sur Concorde ou avec du logiciel pour les contrôleurs numériques comme dans les avions de ligne modernes. Par la suite, on s'intéressera uniquement au dernier cas. Pour être implémenté dans du logiciel, le contrôleur doit être discrétisé

$$x_{c_{k+1}} = A'_c x_{c_k} + B'_c u_{c_k}.$$

La valeur de l'état interne $x_c$ est alors recalculée à une fréquence fixée[23]. Cela donne enfin un code qui ressemble au morceau de code C suivant :

[17] Un vecteur avec les positions, vitesses, angles et vitesses angulaires par exemple pour un avion, ou la position horizontale et la vitesse de la base du pendule inversé avec l'angle et la vitesse angulaire du pendule.

[18] La position des gouvernes pour un avion ou la tension en entrée du moteur déplaçant la base du pendule inversé pour continuer avec nos exemples.

[19] Manche et palonnier du pilote sur un avion ou position désirée du pendule inversé.

[20] Avant l'ère des ordinateurs, les bureaux d'études et les centres de recherche étaient équipés de salles remplies de personnes passant leurs journées entières à réaliser des calculs pour les ingénieurs ou chercheurs. À l'époque, la parité n'était pas meilleure qu'aujourd'hui et la plupart de ces personnes étaient des femmes tandis que la plupart des ingénieurs étaient des hommes et il y eut de nombreux cas d'ingénieurs se mariant à leur « calculatrice ».

[21] Par exemple, ils permirent aux dernières versions du lanceur Ariane de mettre en orbite des satellites plus lourd sans nécessiter plus de carburant que des versions précédentes.

[22] Un bon exemple est le régulateur centrifuge inventé par l'anglais James Watt pour contrôler la vitesse des moteurs à vapeur durant la révolution industrielle.

[23] Les fréquences typiques pour les commandes de vol sont de l'ordre de 100 Hz.

```
double x[3] = {0, 0, 0};
double nx[3];
double in;
while (1) {
  in = recuperer_entree();
  nx[0] = 0.9379*x[0] - 0.0381*x[1] - 0.0414*x[2] + 0.0237*in;
  nx[1] = -0.0404*x[0] + 0.968*x[1] - 0.0179*x[2] + 0.0143*in;
  nx[2] = 0.0142*x[0] - 0.0197*x[1] + 0.9823*x[2] + 0.0077*in;
  x[0] = nx[0]; x[1] = nx[1]; x[2] = nx[2];
  attendre_tick_horloge();  // un tick par 10 ms par exemple
}
```

qui est un code hautement numérique, assez différent par exemple du code source d'un système d'exploitation ou d'un compilateur. Bien sûr, le véritable contrôleur des commandes de vol d'un avion est quelque chose de bien plus complexe. Des saturations sont ajoutées sur les sorties de façon à éviter d'envoyer aux actuateurs des ordres hors de leurs capacités, plusieurs contrôleurs sont développés pour un certain nombre de points de vol (définis par une altitude, une vitesse, un poids et un centrage de l'avion) et doucement combinés tous ensembles et des correcteurs sont ajoutés sur les entrées ou les sorties pour prendre en compte divers phénomènes. Finalement, le logiciel peut nécessiter quelques centaines de milliers de lignes de code C.

Enfin, il est intéressant de noter pour les informaticiens que les automaticiens qualifieraient le contrôleur ci dessus comme ayant *trois états*, puisque le vecteur $x_c$ a trois composantes. Évidemment, il a un nombre infini d'états et n'est pas un automate fini à trois états. *Trois variables d'états* pourrait être une formulation plus appropriée.

**Les systèmes critiques** comme les commandes de vol peuvent avoir des effets désastreuses en cas de dysfonctionnement. Deux types de dysfonctionnements différents doivent être distingués : les dysfonctionnements matériel et logiciel. Pour éviter qu'un problème matériel ait des conséquences désastreuses, de multiples exemplaires redondants des appareils plus ou moins susceptibles de dysfonctionner sont embarqués. Par exemple, un schéma courant, est de tripliquer les capteurs en entrée[24] et d'utiliser un voteur choisissant la valeur médiane des trois valeurs de telle sorte qu'un capteur défectueux n'ait aucun impact. De la diversité peut aussi être introduite parmi les différents exemplaires de façon à éviter qu'une erreur de conception affecte tous les exemplaires e même temps[25]. Cet usage de la duplication et de la diversification pour minimiser les problèmes matériels est plus ancien que les commandes de vol numériques. Par exemple, les commandes hydrauliques utilisaient déjà plusieurs circuits hydrauliques, chacun alimentant un ensemble de gouvernes complémentaires et les concepteurs prenaient soin de faire courir les tubulures des différents circuits aussi loin que possible les unes des autres dans l'avion de sorte à limiter le risque que tous les circuits soient endommagés en même temps. Toutes ces stratégies de duplication et de diversification et ces algorithmes de vote soulèvent d'intéressantes questions de vérification. Ce n'est toutefois pas le sujet de cette thèse.

[24] Comme les tubes de Pitot mesurant la vitesse de l'avion par exemple.

[25] Par exemple sur l'A320, les ordinateurs des commandes de vol sont de deux types différents. Du fait d'un problème de climatisation dans la baie avionique où ils se trouvaient, tous les ordinateurs d'un des deux types ont brûlé au cours d'un vol d'essais. Heureusement, l'autre type permettait encore à l'avion de voler en toute sécurité.

Le logiciel constitue aussi une source potentielle d'erreurs. C'est pourquoi il doit satisfaire à des exigences draconiennes [RTC92]. Ces exigences sont actuellement mises en oeuvre par des tests intensifs. On pourrait s'en satisfaire dans la mesure où après des décennies d'utilisation quotidienne de milliers d'avions, aucune perte de vie humaine ne peut être imputée à une erreur logicielle dans un système de commandes de vol numérique. De tels processus sont toutefois terriblement lourds et coûteux[26]. De plus, le nombre de scénarios de test étant bien trop grand, le test exhaustif est impossible et il n'y a aucune garantie que les cas de test choisis n'échouent pas à mettre un bug en évidence[27]. Tout cela justifie l'intérêt à la fois des industriels et des communautés académiques pour les méthodes formelles capables de fournir plus ou moins automatiquement des preuves mathématiques de correction. Parmi elles, cette thèse va particulièrement se concentrer sur l'interprétation abstraite, une méthode efficace pour générer automatiquement des preuves de propriété numériques qui sont essentielles dans notre contexte.

Deux types de propriétés sont d'un intérêt majeur[28] : la *stabilité en boucle fermée* et la *stabilité en boucle ouverte*. La stabilité en boucle fermée, illustrée sur la Figure 1, exprime que le contrôleur réussit à stabiliser la plant (en supposant que son modèle simule fidèlement le système physique).[29] La stabilité en boucle ouverte est illustrée sur la Figure 2. Cette propriété signifie que toutes les variables du contrôleur sont bornées (en supposant que ses entrées sont bornées). Cela peut à première vue sembler être une propriété plus artificielle puisqu'un système peut être stable en boucle fermée sans l'être en boucle ouverte. En effet, si le contrôleur n'est pas stable (en boucle ouverte), tant que le système physique ne lui fournit pas en entrée des données qui le feraient diverger, le système entier reste stable (en boucle fermée). Toutefois, la stabilité en boucle ouverte reste une propriété intéressante. En particulier, un contrôleur stable en boucle ouverte ne divergera pas catastrophiquement s'il subit un dysfonctionnement temporaire de ses capteurs d'entrée (i.e., coupant la boucle de la Figure 1) rendant alors son retour en fonctionnement normal bien moins improbable quand les capteurs sont de retour. C'est pourquoi la stabilité en boucle ouverte est souvent exigée des systèmes critiques. Il est intéressant de noter que la notion actuelle de stabilité en boucle ouverte diffère sensiblement de celle à laquelle sont habitués les automaticiens. Cette dernière comprend la plant qui peut être intéressante pour le développement du contrôleur mais est complètement inutile quand on vérifie un contrôleur déjà développé. C'est même un avantage des propriétés en boucle ouverte de ne pas faire d'hypothèses sur la plant. Ce document traite principalement de stabilité en boucle ouverte, bien que la stabilité en boucle fermée constituerait certainement une extension très intéressante.

[26] Par exemple, sur le B777, la vérification compte pour 70% des coûts de développement logiciel (qui représentent eux même un tiers des coûts totaux de développement de l'avion) [FBG+].

[27] D'après Edsger W. Dijkstra : "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." [Dij72].

[28] Au moins si on s'intéresse à la stabilité, d'autres propriétés comme les performances sortant du cadre de cette thèse.

[29] Dans le cas d'un avion, ça signifierait que son attitude reste entre certaines bornes (pas de vitesse verticale excessive par exemple), pour le pendule inversé qu'il reste proche de sa position verticale.

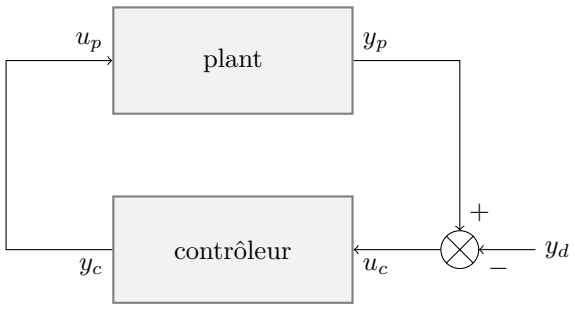



FIGURE 1 – Stabilité en boucle fermée : la commande $y_d$ étant bornée, let états internes $x_p$ et $x_c$ et les sorties $y_p$ et $y_c$ de la plant et du contrôleur restent bornés.


FIGURE 2 – Stabilité en boucle ouverte : l'entrée $u_c$ du contrôleur étant bornée, son état interne $x_c$ et sa sortie $y_c$ restent bornés.

## Contributions

Le reste de cette première partie de ce document (traduite en français) ne contient aucune contribution mais introduit plutôt quelques notions qui seront ensuite utilisées à travers le reste de la thèse. Plus précisément, le chapitre suivant (page xix) introduit la notion d'invariant inductif et le besoin d'outils d'inférence d'invariants. Suit une brève introduction à une de ces techniques d'inférence d'invariants, à savoir l'interprétation abstraite (page xxiii). Enfin, un dernier chapitre (page xxxiii) présente un cour état de l'art des techniques d'inférence d'invariants pour la stabilité en boucle ouverte des systèmes de contrôle commande linéaires.

Le document est ensuite organisé en trois parties (non traduites), chacune correspondant à une contribution distincte. Il est bien connu des automaticiens que les contrôleurs linéaires sont stables si et seulement si ils admettent un invariant quadratique (géométriquement parlant, un ellipsoïde). Ils appellent ces invariants *fonctions de Lyapunov quadratiques* et une première partie (Partie II) propose de calculer automatiquement de tels invariants pour des contrôleurs donnés comme une paire de matrices $A'_c$ and $B'_c$. Ceci est fait en utilisant des outils d'optimisation de programmation semi-définie. Il est intéressant de noter que les aspects virgule flottante sont pris en compte, qu'ils affectent les calculs effectués par le programme analysé ou par les outils utilisés pour l'analyse.

Toutefois, le véritable but est d'analyser des programme implémentant des contrôleurs (et non des paires de matrices), incluant éventuellement des remises à zéro ou des saturations, donc non purement linéaires. Les techniques d'*itération sur les politiques* sont des techniques d'analyse statique récemment développées et bien adaptées à cet usage. Toutefois, elles ne se marient pas facilement avec le paradigme classique de l'interprétation abstraite. La partie suivante (Partie III) essaye de proposer une interface propre entre les deux mondes.

Enfin, la dernière partie (Partie IV) est un travail plus prospectif sur l'usage

de l'optimisation globale polynomiale basée sur les polynômes de Bernstein pour calculer des invariants polynomiaux sur des systèmes polynomiaux.

# Invariants inductifs

Ce chapitre introduit la notion d'invariant $k$-inductif et la méthode de vérification par $k$-induction et montre comment elle peut profiter d'une méthode d'inférence automatique d'invariants. Le reste du document va ensuite se concentrer sur cette génération d'invariants dans le cas de coeurs de contrôle commande numériques.

Un modèle très simple de système de transition est tout d'abord introduit. Malgré sa simplicité, il permet de modéliser les systèmes de contrôle commande numériques de cette thèse tout en autorisant la discussion sur les invariants menée dans ce chapitre.

**Definition 1** (Système de transition)**.** *Étant donné un ensemble $s$, une paire $(I, T)$ est appelée un* système de transition *si $I \subseteq S$ et $T \subseteq S \times S$. Le sous ensemble $I$ de $S$ est appelé ensemble d'*états initiaux *tandis que la relation $T$ sur $S$ est appelée* relation de transition*.*

*Par la suite, on notera $T(X)$ l'ensemble des états accessibles en une transition $T$ depuis un état dans $X$ :*

$$T(X) := \{s' \in S \mid \exists\, s \in X, (s, s') \in T\}.$$

La définition suivante donne une sémantique aux systèmes de transition.

**Definition 2** (Espace d'états accessibles)**.** *Un ensemble $R \subseteq S$ est appelé* espace d'états accessibles *d'un système de transition $(I, T)$ sur $S$ s'il est le plus petit ensemble (pour l'ordre $\subseteq$) satisfaisant :*

$$\begin{cases} I \subseteq R \\ T(R) \subseteq R. \end{cases} \tag{1}$$

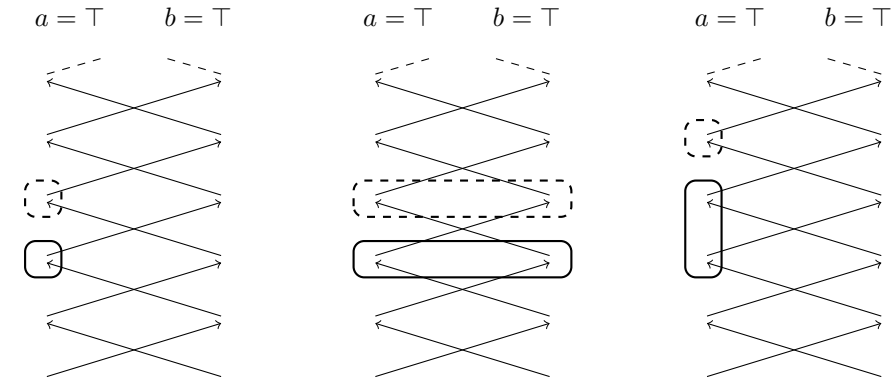
**Property 1.** *Tout système de transition a un unique espace d'états accessibles.*

*Démonstration.* Nous devons prouver l'existence et l'unicité d'un tel $R$. Comme tout ensemble de sous ensembles, $(\mathcal{P}(S), \subseteq)$ est un treillis complet. De plus, la fonction $f : X \in \mathcal{P}(S) \mapsto I \cup T(X)$ est croissante sur ce treillis. Alors, par le théorème de Knaster-Tarski [Kna28, Tar55], $R$ est l'unique plus petit point fixe de la fonction $f$. □

**Example 1.** *Étant donné $S := \mathbb{Z}^2$ :*

$$I := \{\, (0,0) \,\}$$
$$T := \left\{ ((x,y),(x',y')) \;\middle|\; \begin{array}{l} (x \leqslant 0 \wedge x' = x \wedge y' = y) \\ \vee\, (x \geqslant 1 \wedge x' = x \wedge y' = y + 1) \end{array} \right\}$$

*définit un système de transition. Son espace d'états accessibles est le singleton $R := \{\, (0,0) \,\}$.*




(a) $P := \{(a, b) \mid a = \top\}$ n'est pas inductif.

(b) $P' := \{ (\top, \top) \}$ est inductif.

(c) $P$ est 2-inductif.

FIGURE 3 – Illustration de l'Exemple 3.

**Definition 3** (Invariant)**.** *Une propriété $P \subseteq S$ est dite* invariant *d'un système de transition si $R \subseteq P$, i.e., si tout état accessible satisfait la propriété $P$.*

**Definition 4** (Invariant inductif)**.** *$P \subseteq S$ est dit* invariant inductif *d'un système de transition quand on a à la fois $I \subseteq P$ et $T(P) \subseteq P$.*

**Property 2.** *Tout les invariants inductifs sont des invariants.*

Il est intéressant de noter que la réciproque n'est pas vraie en toute généralité, comme démontré par le contre exemple suivant.

**Example 2.** *Sur le système de transition de l'Exemple 1, $P := \{(x, y) \mid y = 0\}$ est un invariant. Toutefois il n'est pas inductif puisque $T(P) = \{(x, y) \mid 0 \leqslant y \leqslant 1\}$. $P' := \{ (0, 0) \}$ est un invariant inductif.*

Ceci illustre un fait apparaissant couramment quand on essaye de prouver un résultat mathématique par récurrence. Il est souvent nécessaire de considérer une version plus forte du résultat pour que la preuve par récurrence fonctionne. Une astuce habituelle est de supposer que la propriété est vrai plusieurs fois consécutives (plutôt que seulement une fois) avant de prouver qu'elle l'est la fois suivante.

**Definition 5** (Invariant $k$-inductif)**.** *$P \subseteq S$ est dit $k$-*inductif *si :*

$$\bigcup_{0 \leqslant i \leqslant k-1} T^i(I) \subseteq P \quad \textit{and} \quad T^k\left( \bigcap_{0 \leqslant i \leqslant k-1} \{ s \in S \mid T^i(\{ s \}) \subseteq P \} \right) \subseteq P.$$

Fondamentalement, une propriété $P$ est $k$-inductive si au plus $k-1$ transition depuis un état initial mènent toujours à un état dans $P$ et si une transition supplémentaire après $k-1$ transition dans $P$ reste toujours dans $P$. Il est intéressant de noter que le cas particulier de la 1-induction correspond à la notion de récurrence telle que définie dans la Définition 4.

L'Exemple 2 montre qu'un invariant n'est pas nécessairement inductif. Il existe même des invariants qui ne sont $k$-inductifs pour aucun $k$. Toutefois, certains invariants $k$-inductifs ne sont pas $k-1$ inductifs.

**Example 3** (Lacets)**.** *Étant donné* $S := \mathbb{B}^2 = \{\bot, \top\}^2$*, on définit* $(I, T)$ *avec* $I := \{(\top, \top)\}$ *et* $T := \{((a, b), (a', b')) \mid (a = \bot \vee b' = \top) \wedge (b = \bot \vee a' = \top)\}$*. L'espace d'états accessibles de ce système de transition est* $R := \{(\top, \top)\}$*.*

*Bien que* $P := \{(a, b) \mid a = \top\}$ *est un invariant de ce système de transition, il n'est pas inductif. La propriété renforcée* $P' := \{(a, b) \mid a = \top \wedge b = \top\}$ *est inductive mais* $P$ *peut aussi être prouvé par* 2*-induction. Tout ceci est illustré sur la Figure 3 qui justifie le nom de* lacets *souvent donné à cet exemple courant.*

Ceci est utilisé par la procédure de $k$-induction pour tenter de prouver qu'une propriété $P$ est un invariant d'un système de transition $(I, T)$.

**Definition 6** (Procédure de $k$-induction)**.**

1. *Mettre* $k$ *à 1.*
2. *Tester s'il existe* $s_0, \ldots, s_{k-1}$ *satisfaisant la formule suivante :*

$$s_0 \in I \wedge \left( \bigwedge_{0 \leqslant i \leqslant k-2} (s_i, s_{i+1}) \in T \right) \wedge \neg (s_{k-1} \in P)\,. \tag{2}$$

3. *S'ils existent, alors renvoyer* `Faux` *(et* $s_0, \ldots, s_{k-1}$ *constituent un contre exemple).*
4. *Sinon, tester s'il existe* $s_0, \ldots, s_k$ *satisfaisant :*

$$\left( \bigwedge_{0 \leqslant i \leqslant k_1} s_i \in P \right) \wedge \left( \bigwedge_{0 \leqslant i \leqslant k_1} (s_i, s_{i+1}) \in T \right) \wedge \neg (s_k \in P)\,. \tag{3}$$

5. *Si cette formule n'est pas satisfiable, alors renvoyer* `Vrai` *(nous avons prouvé que* $P$ *est* $k$*-inductive).*
6. *Sinon, incrémenter* $k$ *et retourner en 2.*

En pratique, la satisfiabilité des formules 2 and 3 est testée en utilisant un solveur SMT [BSST09]. Ces formules sont souvent appelées respectivement *base* and *step*. Ainsi, un *contre exemple de step* est un modèle de 3.

Cette procédure ne termine pas. Toutefois, dans les cas où elle termine, elle prouve soit que $P$ est un invariant ou donne un contre exemple (tout au moins en supposant que la logique sous jacente est décidable et que le solveur SMT répond toujours satisfiable (avec un modèle de la formule) ou insatisfiable). Dans ce dernier cas, c'est équivalent à un test — particulièrement coûteux — de model checking borné [Bie09]. Ainsi, cette procédure présente un intérêt pratique seulement quand $P$ est $k$-inductive (avec $k$ assez petit pour que les formules 2 et 3 puissent être prouvées insatisfiables par un solveur SMT).

**Remark 1** (Sur les invariants et invariants inductifs)**.** *Dans la suite de ce document, le terme « invariant » sera couramment utilisé à la place de « invariant inductif ». Dans les rare cas où l'invariant considéré ne serait pas inductif, cela sera explicitement précisé.*

# Interprétation abstraite

L'interprétation abstraite est un cadre formel pratique et polyvalent pour définir des analyses statiques de programmes [Cou99, CC77, CC79, CC92].

Le but de ce chapitre n'est certainement pas de donner une vue d'ensemble complète de l'interprétation abstraite mais plutôt de rendre ce document raisonnablement auto suffisant en introduisant un langage jouet et les notations utilisées par la suite.

## Un langage impératif jouet

À travers ce document, un langage impératif jouet très classique sera utilisé pour illustrer nos analyses.

Les programmes de ce langage manipulent seulement des nombres réels. De plus, pour garder les choses aussi simples que possible par la suite, le langage contient aussi peu de constructions que possible. Il pourrait être étendu avec des appels de fonctions, des variables de différent types comme les entiers ou les booléens, des pointeurs, des structures de données complexes,... Toutefois, il est déjà Turing complet et ces fonctionnalités additionnelles seraient d'un intérêt douteux pour introduire les analyses numériques réalisées tout au long de ce document.

Enfin, il est intéressant de noter que les logiciels de contrôle commande sont idéalement écrits dans un langage synchrone comme Lustre ou Scade [BCE+03, HCRP91]. Toutefois, l'interprétation abstraite de programmes écrits dans ces langages peut être efficacement réalisée en appliquant d'abord un processus de compilation approprié produisant des programmes impératifs [Jea00, Jea03]. Ainsi, cette thèse ne traitera que de tels programmes impératifs.

### Syntaxe

Un programme $p$ du langage est un énoncé $stm$ dans la grammaire suivante :

$$
\begin{aligned}
stm ::= {} & stm; stm \mid v := expr \mid v := ?(r, r) \\
& \mid \textbf{if } expr \leqslant r \textbf{ then } stm \textbf{ else } stm \textbf{ fi} \\
& \mid \textbf{while } expr \leqslant r \textbf{ do } stm \textbf{ od} \\
expr ::= {} & v \mid r \mid expr + expr \mid expr - expr \mid expr \times expr
\end{aligned}
$$

avec $v \in \mathbb{V}$, un ensemble de variables, et $r \in \mathbb{R}$. $?(r_1, r_2)$ représentant le choix aléatoire d'un nombre réel entre $r_1$ et $r_2$ (pratique pour simuler des entées).

**Example 4.** *La Figure 4 présente un programme dans ce langage.*

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

FIGURE 4 – Exemple de programme.

**Remark 2.** *Le programme de la Figure 4 est typique des programmes de contrôle commande : quelque énoncés d'initialisation suivis par une boucle infinie. Dans cette boucle, quelques variables d'entrées (la variable in ici) sont acquises (typiquement par des capteurs mesurant certaines valeurs sur le système physique contrôlé) et un état interne (les variables xi ici) sont mises à jour. Cet état interne peut ensuite être utilisé pour calculer quelque valeurs de sortie (typiquement pour les envoyer à un actuateur agissant sur le système physique). Tout cela étant répété à une période fixée. De tels programmes sont souvent appelés* système réactifs.

*Tout au long de ce document, ce type de programme de la forme*

$$s;\ \textbf{while}\ -1 \leqslant 0\ \textbf{do}\ \mathrm{s}'\ \textbf{od}$$

*où $s$ et $s'$ sont des énoncés et $s$ ne contient aucune boucle while, seront très couramment considérés.*

## Sémantique collectrice

**Definition 7** (Sémantique des expressions $[\![e]\!]$)**.** *La sémantique $[\![e]\!](\rho) \in \mathbb{R}$ d'une expression $e$ dans un environnement $\rho : \mathbb{V} \to \mathbb{R}$, est définie comme :*

$$\begin{array}{l}
[\![v]\!](\rho) := \rho(v) \\
[\![r]\!](\rho) := r \\
[\![e_1 \diamond e_2]\!](\rho) := [\![e_1]\!](\rho) \diamond [\![e_2]\!](\rho) \qquad \text{for } \diamond \in \{+, -, \times\}.
\end{array}$$

**Definition 8** (Sémantique des énoncés $[\![s]\!]$)**.** *Cela permet de donner une sémantique $[\![s]\!](R) \subseteq (\mathbb{V} \to \mathbb{R})$ pour un énoncé $s$ et un ensemble d'environnements $R \subseteq (\mathbb{V} \to \mathbb{R})$ :*

$$\begin{array}{l}
[\![s_1; s_2]\!](R) := [\![s_2]\!]([\![s_1]\!](R)) \\
[\![v := e]\!](R) := \{\rho[v \leftarrow [\![e]\!](\rho)] \mid \rho \in R\} \\
[\![v := ?(\mathrm{r}_1, \mathrm{r}_2)]\!](R) := \{\rho[v \leftarrow r] \mid \rho \in R, r \in \mathbb{R}, r_1 \leqslant r \leqslant r_2\} \\
[\![e \bowtie r]\!](R) := \{\rho \in R \mid [\![e]\!](\rho) \bowtie r\} \text{ for } \bowtie \in \{>, \geqslant, <, \leqslant\} \\
[\![\textbf{if } \mathrm{e} \leqslant \mathrm{r} \textbf{ then } \mathrm{s}_1 \textbf{ else } \mathrm{s}_2 \textbf{ fi}]\!](R) := [\![s_1]\!]([\![e \leqslant r]\!](R)) \cup [\![s_2]\!]([\![e > r]\!](R)) \\
[\![\textbf{while } \mathrm{e} \leqslant \mathrm{r} \textbf{ do } \mathrm{s} \textbf{ od}]\!](R) := [\![e > r]\!](\mu\,(X \mapsto R \cup [\![s]\!]([\![e \leqslant r]\!](X)))).
\end{array}$$

*avec $\mu\, f$ le plus petit point fixe de la fonction $f$.*

*Démonstration.* Pour que $[\![s]\!]$ soit bien défini, l'existence du plus petit point fixe doit être prouvée. Cela peut se faire en prouvant que $[\![s]\!]$ est croissante, i.e., $[\![s]\!]$ est bien définie et pour tout $R, R' \subseteq (\mathbb{V} \to \mathbb{R})$, si $R \subseteq R'$ alors $[\![s]\!](R) \subseteq [\![s]\!](R')$.

La preuve étant menée par induction structurelle sur les énoncés $s$, le seul cas non trivial est celui de la boucle while.

Ainsi, en considérant une expression $e$, un réel $r$ et un énoncé $s$ et en supposant que $[\![s]\!]$ est croissante, nous devons prouver que $[\![\textbf{while e} \leqslant \textbf{r do s od}]\!]$ est bien définie sur $\mathcal{P}(\mathbb{V} \to \mathbb{R})$ et que pour tout $R, R' \subseteq (\mathbb{V} \to \mathbb{R})$ tel que $R \subseteq R'$, nous avons

$$[\![\textbf{while e} \leqslant \textbf{r do s od}]\!](R) \subseteq [\![\textbf{while e} \leqslant \textbf{r do s od}]\!](R').$$

Remarquons tout d'abord que $(\mathcal{P}(\mathbb{V} \to \mathbb{R}), \subseteq)$ est un treillis complet (puisque pour tout ensemble $S$, l'ensemble des ses parties $\mathcal{P}(S)$ muni de l'ordre $\subseteq$ est un treillis complet). De plus, par hypothèse d'induction, la fonction $f_R : X \mapsto R \cup [\![s]\!]([\![e \leqslant r]\!](X))$ est croissante sur ce treillis. Alors, d'après le théorème de Knaster-Tarski [Kna28, Tar55], le plus petit point fixe $\mu\, f_R$ est défini de manière unique comme

$$\mu\, f_R = \bigcap \{X \in \mathcal{P}(\mathbb{V} \to \mathbb{R}) \mid f_R(X) \subseteq X\}$$

et de même pour $R'$. Puisque $R \subseteq R'$, par définition de $f_R$ and $f_{R'}$, pour tout $X$, $f_R(X) \subseteq f_{R'}(X)$ ce qui implique que $\{X \mid f_{R'}(X) \subseteq X\} \subseteq \{X \mid f_R(X) \subseteq X\}$. Ainsi $\mu\, f_R \subseteq \mu\, f_{R'}$, d'où le résultat final. □

**Definition 9** (Sémantique des programmes $[\![p]\!]$)**.** *Enfin, la sémantique d'un programme $p$ est donnée par la sémantique $[\![p]\!](\mathbb{V} \to \mathbb{R})$ de $p$ partant d'un état inconnu.*

Cette sémantique dénotationnelle est très classique et peut se trouver dans des livres de cours sur la sémantique des langages de programmation [Win93][30].

Il est intéressant de noter que cette sémantique est donnée avec des opérations sur les nombres réels $\mathbb{R}$ alors qu'un véritable programme calculerait en utilisant des nombres à virgule flottante. Ce sujet sera brièvement abordé plus tard mais est actuellement grandement laissé pour de futurs développements.

**Remark 3.** *Le cas courant des coeurs de contrôleur*

$$s;\ \textbf{while } -1 \leqslant 0 \textbf{ do s}' \textbf{ od}$$

*où $s$ est un énoncé sans boucle peut être vu comme un système de transition, comme défini au Chapitre 2, avec*

$$I := [\![s]\!](\mathbb{V} \to \mathbb{R}) \quad \textit{and} \quad T := \{(X, [\![s']\!](X)) \mid X \in \mathcal{P}(\mathbb{V} \to \mathbb{R})\}.$$

*En fait, sont espace d'états accessibles $R$ est donné par*

$$R := \mu\,(X \mapsto [\![s]\!](\mathbb{V} \to \mathbb{R}) \cup [\![s']\!](X)).$$

*Cela donne un sens aux notions d'invariant et d'invariant inductif sur ce type de programmes. Ainsi, l'ensemble $R$, également appelé* ensemble d'états accessibles en tête de boucle *sera considéré à la place de la sémantique du programme entier $[\![s;\ \textbf{while } -1 \leqslant 0 \textbf{ do s}' \textbf{ od}]\!]$ qui n'est autre que l'ensemble vide $\emptyset$ puisque le programme ne termine jamais.*

[30] Il faut noter toutefois que tous les auteurs ne s'accordent pas sur la définition de pre et post point fixes d'une fonction $f$. Certains [Win93] définissent un pre point fixe comme un point $x$ tel que $f(x) \leqslant x$ tandis que dans la communauté interprétation abstraite [Cou99, CC77, CC79, CC92], il est couramment défini comme un point $x$ tel que $x \leqslant f(x)$ (et de manière duale pour les post point fixes). Dans la suite de ce document, la seconde convention est adoptée.

## Domaines abstraits

La sémantique concrète précédente n'étant pas calculable[31], l'idée de base de l'interprétation abstraite est de calculer une sémantique abstraite. Cette sémantique abstraite est conçue comme une surapproximation calculable de la sémantique concrète.

Les *domaines abstraits* sont les briques de base des interprètes abstraits. Ils sont basés sur un *treillis complet* $\left(\mathcal{D}^\sharp, \sqsubseteq^\sharp_\mathcal{D}\right)$ (l'exposant $.^\sharp$ est couramment utilisé pour distinguer les objets abstraits de leur contrepartie concrète (ici le treillis complet $(\mathcal{D}, \sqsubseteq_\mathcal{D}) := (\mathcal{P}(\mathbb{V} \to \mathbb{R}), \subseteq)$)). La borne supérieure (least upper bound en anglais, également appelée union) du treillis, sa borne inférieure (greatest lower bound en anglais, aussi appelée intersection), ses plus petit et plus grand éléments sont respectivement notés $\sqcup^\sharp_\mathcal{D}$, $\sqcap^\sharp_\mathcal{D}$, $\bot_\mathcal{D}$ et $\top_\mathcal{D}$. Des versions binaires des bornes supérieures et inférieures seront couramment utilisées (i.e., $\sqcup^\sharp_\mathcal{D} \{ x^\sharp, y^\sharp \}$ sera noté $x^\sharp \sqcup^\sharp_\mathcal{D} y^\sharp$). Pour alléger les notations, l'indice $_\mathcal{D}$ sera souvent omis quand il peut être facilement inféré à partir du contexte. Chaque élément de $\mathcal{D}^\sharp$ appelé une *valeur abstraite* va « abstraire » un élément concret de $\mathcal{D}$. Ce que « abstraire » signifie est formellement défini par une fonction de $\mathcal{D}^\sharp$ dans $\mathcal{D}$ appelée une *fonction de concrétisation*.

**Definition 10** (Fonction de concretisation)**.** *Une* fonction de concrétisation *est une fonction $\gamma_\mathcal{D} : \mathcal{D}^\sharp \to \mathcal{D}$ du domaine abstrait $\mathcal{D}^\sharp$ vers le concret $\mathcal{D}$. Cette fonction doit être croissante :*

$$\forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp,\ x^\sharp \sqsubseteq^\sharp_\mathcal{D} y^\sharp \Rightarrow \gamma_\mathcal{D}(x^\sharp) \sqsubseteq_\mathcal{D} \gamma_\mathcal{D}(y^\sharp).$$

Ainsi, dans notre cas, une valeur abstraite $x^\sharp$ de $\mathcal{D}^\sharp$ est associée à une valeur concrète $\gamma_\mathcal{D}(x^\sharp) \subseteq (\mathbb{V} \to \mathbb{R})$, i.e., un ensemble d'environnements $\rho : \mathbb{V} \to \mathbb{R}$ associant une valeur dans $\mathbb{R}$ à chaque variable dans $\mathbb{V}$. La contrainte de monotonicité permet de donner un sens concret à l'ordre abstrait $\sqsubseteq^\sharp_\mathcal{D}$ : quand une valeur abstraite $x^\sharp$ est plus petite qu'une autre $y^\sharp$ dans $\mathcal{D}^\sharp$ (i.e., $x^\sharp \sqsubseteq^\sharp_\mathcal{D} y^\sharp$), alors $x^\sharp$ représente moins d'environnements que $y^\sharp$ (i.e., $\gamma_\mathcal{D}(x^\sharp) \subseteq \gamma_\mathcal{D}(y^\sharp)$). En d'autres termes, $x^\sharp \sqsubseteq^\sharp_\mathcal{D} y^\sharp$ signifie que l'abstraction $x^\sharp$ est *plus précise* que $y^\sharp$.

**Example 5** (Domaine des intervalles)**.** *Un domaine abstrait très courant est le* domaine des intervalles *: $\mathcal{I}^\sharp := \mathbb{V} \to I$ avec $I := \bot_I \cup \left\{ [a, b] \;\middle|\; a \in \overline{\mathbb{R}}, b \in \overline{\mathbb{R}}, a \leqslant b \right\}$ où $\overline{\mathbb{R}} := \mathbb{R} \cup \{ -\infty, +\infty \}$ et $\leqslant$ est l'ordre habituel sur $\overline{\mathbb{R}}$. Un ordre $\sqsubseteq^\sharp_\mathcal{I}$ est défini sur $\mathcal{I}^\sharp$ par $x^\sharp \sqsubseteq^\sharp_\mathcal{I} y^\sharp$ quand pour tout $v \in \mathbb{V}$ :*

- *soit $x^\sharp(v) = \bot_I$;*
- *soit $x^\sharp(v) = [a, b]$ et $y^\sharp(v) = [c, d]$ et $c \leqslant a$ et $b \leqslant d$.*

*Équipé de cet ordre, $\mathcal{I}^\sharp$ est un treillis complet. Étant donné un sous ensemble $S$ de $I$, $\sqcup^\sharp_I S$ est défini comme :*

$$\sqcup^\sharp_I S := \begin{cases} \bot_I & \text{quand } S = \emptyset \text{ ou } S = \{ \bot_I \} \\ \left[\inf \left\{ a \in \overline{\mathbb{R}} \;\middle|\; [a, b] \in S \right\}, \right. \\ \left. \sup \left\{ b \in \overline{\mathbb{R}} \;\middle|\; [a, b] \in S \right\}\right] & \text{sinon} \end{cases}$$

[31] Autrement, le problème de l'arrêt (sur notre langage de programmation Turing complet) serait résolu simplement en testant si la sémantique des programmes est $\emptyset$ ou non.

*ce qui permet de définir la borne supérieure* $\sqcup_{\mathcal{I}}^{\sharp}$ *de* $\mathcal{I}^{\sharp}$ *:*

$$\sqcup_{\mathcal{I}}^{\sharp} S := \left( v \mapsto \sqcup_{I}^{\sharp} \left\{ x^{\sharp}(v) \mid x^{\sharp} \in S \right\} \right).$$

*Les extremums* $\bot_{\mathcal{I}}$ *et* $\top_{\mathcal{I}}$ *sont les fonctions associant à toute variable respectivement* $\bot_I$ *et* $[-\infty, +\infty]$.

*La fonction de concrétisation* $\gamma_{\mathcal{I}} : \mathcal{I}^{\sharp} \to 2^{(\mathbb{V} \to \mathbb{R})}$ *est donnée par :*

$$\gamma_{\mathcal{I}}(x^{\sharp}) := \left\{ \rho : \mathbb{V} \to \mathbb{R} \;\middle|\; \forall v \in \mathbb{V}, \exists a, b \in \overline{\mathbb{R}}, x^{\sharp}(v) = [a, b] \wedge a \leqslant \rho(v) \leqslant b \right\}.$$

*Il est intéressant de noter que* $\gamma_{\mathcal{I}}(x^{\sharp}) = \emptyset$ *quand* $x^{\sharp}(v) = \bot_I$ *pour n'importe quelle variable* $v \in \mathbb{V}$.

## Opérateurs abstraits

La section précédente définissait seulement une abstraction. Le but étant d'utiliser cette abstraction pour calculer une surapproximation de la sémantique concrète (qui n'est pas calculable), cela nécessite des opérations sur cette abstraction. Les *opérateurs abstraits* sont des fonctions sur un domaine abstrait qui miment les véritables opérations comme les affectations et les gardes de façon à permettre le calcul dans le domaine abstrait d'une surapproximation de la sémantique concrète définie précédemment (page xxiv).

**Definition 11** (Opérateurs abstraits)**.** *Étant donné une variable* $v \in \mathbb{V}$ *et une expression* $e$*, un opérateur abstrait pour l'affectation* $v{:=}e$ *est une fonction* $[\![v{:=}e]\!]^{\sharp} : \mathcal{D}^{\sharp} \to \mathcal{D}^{\sharp}$*. De manière similaire, des opérateurs abstraits pour l'affectation aléatoire* $[\![v{:=}?(r_1, r_2)]\!]^{\sharp} : \mathcal{D}^{\sharp} \to \mathcal{D}^{\sharp}$ *et les gardes* $[\![e \leqslant r]\!]^{\sharp} : \mathcal{D}^{\sharp} \to \mathcal{D}^{\sharp}$ *sont définis.*

*Ces opérateurs sont dits* corrects *par rapport à leur alter ego concret s'ils remplissent la condition suivante :*

$$\forall x^{\sharp} \in \mathcal{D}^{\sharp}, [\![v{:=}e]\!] \left( \gamma_{\mathcal{D}}(x^{\sharp}) \right) \subseteq \gamma_{\mathcal{D}} \left( [\![v{:=}e]\!]^{\sharp}(x^{\sharp}) \right).$$

*Cette condition est appelée* condition de correction*. Des conditions similaires sont définies pour l'affectation aléatoire et les gardes.*

Intuitivement, la condition de correction exprime le fait que l'opérateur abstrait n'oublie aucun comportement de son alter ego concret. Plus précisément, si un environnement $\rho$ peut être obtenu par l'affectation $v{:=}e$ depuis un environnement représenté par $x^{\sharp}$ (i.e., dans $\gamma_{\mathcal{D}}(x^{\sharp})$), c'est à dire $\rho \in [\![v{:=}e]\!] \left( \gamma_{\mathcal{D}}(x^{\sharp}) \right)$, alors cet environnement est nécessairement représenté par le résultat de l'opérateur abstrait $[\![v{:=}e]\!]^{\sharp}$ sur $x^{\sharp}$, c'est à dire $\rho \in \gamma_{\mathcal{D}} \left( [\![v{:=}e]\!]^{\sharp}(x^{\sharp}) \right)$.

Étant donnés ces opérateurs abstraits, une sémantique abstraite des énoncés $[\![s]\!]^{\sharp}$ peut être définie qui correspond principalement à la sémantique concrète précédemment donnée (page xxiv) dans laquelle les opérateurs abstraits remplacent leur alter ego concret :

$$\begin{aligned}
&[\![s_1; s_2]\!]^{\sharp}(x^{\sharp}) := [\![s_2]\!]^{\sharp} \left( [\![s_1]\!]^{\sharp}(x^{\sharp}) \right) \\
&[\![\textbf{if } \mathrm{e} \leqslant \mathrm{r} \textbf{ then } \mathrm{s}_1 \textbf{ else } \mathrm{s}_2 \textbf{ fi}]\!]^{\sharp}(x^{\sharp}) := \\
&\qquad\qquad [\![s_1]\!]^{\sharp} \left( [\![e \leqslant r]\!]^{\sharp}(x^{\sharp}) \right) \sqcup^{\sharp} [\![s_2]\!]^{\sharp} \left( [\![-e \leqslant -r]\!]^{\sharp}(x^{\sharp}) \right) \qquad (4) \\
&[\![\textbf{while } \mathrm{e} \leqslant \mathrm{r} \textbf{ do } \mathrm{s} \textbf{ od}]\!]^{\sharp}(x^{\sharp}) := \\
&\qquad\qquad [\![-e \leqslant -r]\!]^{\sharp} \left( \mu \left( X \mapsto x^{\sharp} \sqcup^{\sharp} [\![s]\!]^{\sharp} \left( [\![e \leqslant r]\!]^{\sharp}(X) \right) \right) \right).
\end{aligned}$$

En supposant que les opérateurs abstraits sont corrects, cette sémantique est alors correcte par rapport à la sémantique concrète :

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![s]\!]\big(\gamma_{\mathcal{D}}(x^\sharp)\big) \subseteq \gamma_{\mathcal{D}}\big([\![s]\!]^\sharp(x^\sharp)\big)\,.$$

La sémantique abstraite d'un programme $p$ est finalement $[\![p]\!]^\sharp(\top)$.

Si les opérateurs abstraits et le plus petit point fixe $\mu$ peuvent être calculés efficacement, cette sémantique abstraite donne un algorithme pratique pour réellement calculer une surapproximation de la sémantique concrète. Si la sémantique abstraite est assez précise pour rester incluse dans une propriété $P$, elle donne alors une méthode pour prouver que cette propriété est vérifiée par le programme. Bien sûr, la méthode n'est pas complète[32] et la réciproque n'est pas vraie, le fait que la sémantique abstraite ne soit pas incluse dans $P$ peut soit signifier que la propriété $P$ est fausse soit que la sémantique abstraite est une surapproximation trop grossière de la sémantique concrète. Toutefois, réellement calculer le plus petit point fixe $\mu$ est loin d'être trivial et sera l'objet de la section suivante.

**Example 6** (Opérateurs abstraits pour le domaine des intervalles)**.** *Les opérations abstraites $+^\sharp$, $-^\sharp$ et $\times^\sharp : I \times I \to I$ peuvent être définies pour les opérations arithmétiques :*

$$+^\sharp : (x,y) \mapsto \begin{cases} \bot_I & \text{si } x = \bot_I \text{ ou } y = \bot_I \\ [a+c, b+d] & \text{si } x = [a,b] \text{ et } y = [c,d] \end{cases}$$

$$-^\sharp : (x,y) \mapsto \begin{cases} \bot_I & \text{si } x = \bot_I \text{ ou } y = \bot_I \\ [a-d, b-c] & \text{si } x = [a,b] \text{ et } y = [c,d] \end{cases}$$

$$\times^\sharp : (x,y) \mapsto \begin{cases} \bot_I & \text{si } x = \bot_I \text{ ou } y = \bot_I \\ [\min(ab,ac,ad,bd), \\ \ max(ab,ac,ad,bd)] & \text{si } x = [a,b] \text{ et } y = [c,d]. \end{cases}$$

*Elles permettent de donner une sémantique abstraite pour les expressions :*

$$\begin{array}{l} [\![v]\!]^\sharp(x^\sharp) := x^\sharp(v) \\ [\![r]\!]^\sharp(x^\sharp) := [r,r] \\ [\![e_1 \diamond e_2]\!]^\sharp(x^\sharp) := \diamond^\sharp\big([\![e_1]\!]^\sharp(x^\sharp), [\![e_2]\!]^\sharp(x^\sharp)\big) \qquad \text{pour } \diamond \in \{+,-,\times\} \end{array}$$

*ce qui permet finalement de définir des opérateurs abstraits pour le domaine des intervalles :*

$$[\![v{:=}e]\!]^\sharp(x^\sharp) := \begin{cases} \bot_{\mathcal{I}} & \text{si } x^\sharp = \bot_{\mathcal{I}} \\ x^\sharp\big[v \mapsto [\![e]\!]^\sharp(x^\sharp)\big] & \text{sinon} \end{cases}$$

$$[\![v{:=}?(r_1,r_2)]\!]^\sharp(x^\sharp) := \begin{cases} \bot_{\mathcal{I}} & \text{si } x^\sharp = \bot_{\mathcal{I}} \text{ or } r_1 > r_2 \\ x^\sharp[v \mapsto [r_1,r_2]] & \text{sinon} \end{cases}$$

$$[\![e \leqslant r]\!]^\sharp(x^\sharp) := x^\sharp.$$

*Pour prouver que ces opérateurs abstraits sont corrects par rapport à leurs alter egos concrets, les opérations arithmétiques sont d'abord prouvées correctes*

[32] Comme mentionné plus haut, cela résoudrait sinon le problème de l'arrêt.

*(i.e., en définissant $\gamma_I : I \to 2^{\mathbb{R}}$ comme la fonction associant à $\bot$ l'ensemble vide $\emptyset$ et à $[a,b]$ l'ensemble $\{x \in \mathbb{R} \mid a \leqslant x \leqslant b\}$, la propriété suivante est vérifiée : $\forall x^\sharp, y^\sharp \in I, \left\{x \diamond y \;\middle|\; x \in \gamma_I\left(x^\sharp\right), y \in \gamma_I\left(y^\sharp\right)\right\} \subseteq \gamma_I\left(\diamond^\sharp\left(x^\sharp, y^\sharp\right)\right)$). Puis la correction de la sémantique abstraite des expressions peut être établie par induction structurelle sur les expressions (i.e., avec le même $\gamma_I$ : $\forall x^\sharp \in \mathcal{I}^\sharp, [\![e]\!]\left(\gamma_{\mathcal{I}}(x^\sharp)\right) \subseteq \gamma_I\left([\![e]\!]^\sharp(x^\sharp)\right)$) ce qui permet finalement de prouver la correction de l'opérateur abstrait d'affectation. La correction des opérateurs abstraits d'affectation aléatoire et de garde peut être obtenue plus directement.*

*Il est intéressant de noter que, bien que correct, l'opérateur abstrait pour les gardes n'est pas très précis (par exemple, $[\![x \leqslant 0]\!]^\sharp([-\infty, +\infty])$ pourrait être défini comme $[-\infty, 0]$ plutôt que $[-\infty, +\infty]$). Une garde abstraite plus précise est habituellement définie en utilisant une sémantique arrière des expressions et des réductions itératives [Cou99, Gra92].*

## Itérations de Kleene et élargissement

Étant donné des opérateurs abstrait calculables, le seul élément manquant pour être capable d'effectivement calculer une sémantique abstraite d'un programme est une façon efficace de calculer l'opérateur de plus petit point fixe $\mu$ apparaissant dans la définition de la sémantique des boucles while. En pratique, atteindre exactement le plus petit point fixe étant trop difficile, seule une surapparoximation de celui ci est calculée. Cela mène à une surapparoximation supplémentaire de la sémantique abstraite mais ne porte pas atteinte à sa correction.

Dans la suite de cette section, la fonction $f_{x^\sharp} : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ est supposée donner la sémantique abstraite d'une garde et d'un corps de boucle à partir d'une valeur abstraite $x^\sharp$ (i.e., pour une boucle "**while** e $\leqslant$ r **do** s **od**", $f_{x^\sharp} := y^\sharp \mapsto x^\sharp \sqcup^\sharp [\![s]\!]^\sharp\left([\![e \leqslant r]\!]^\sharp(y^\sharp)\right)$). Le but est alors de calculer une surapproximation de $\mu\, f_{x^\sharp}$, i.e., un $y^\sharp \in \mathcal{D}^\sharp$ tel qu'il existe un point fixe $z^\sharp$ ($f_{x^\sharp}(z^\sharp) = z^\sharp$) plus petit que $y^\sharp$ ($z^\sharp \sqsubseteq^\sharp_{\mathcal{D}} y^\sharp$).

Quand le treillis $\mathcal{D}^\sharp$ satisfait la condition d'absence de chaîne strictement croissante infinie (i.e., toute suite $\left(x^\sharp_n\right)_{n\in\mathbb{N}}$ telle que pour tout $i \in \mathbb{N}$, $x^\sharp_i \sqsubseteq^\sharp_{\mathcal{D}} x^\sharp_{i+1}$, est stationnaire), la suite $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ est stationnaire (car $f^0_{x^\sharp}(\bot) = \bot \sqsubseteq^\sharp_{\mathcal{D}} f_{x^\sharp}(\bot)$ et par récurrence immédiate en utilisant la croissance de $f_{x^\sharp}$, la suite $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ est croissante, donc stationnaire). Cela signifie qu'il existe un $N \in \mathbb{N}$ tel que $f_{x^\sharp}\left(f^N_{x^\sharp}(\bot)\right) = f^N_{x^\sharp}(\bot)$. $f^N_{x^\sharp}(\bot)$ est alors un point fixe de $f_{x^\sharp}$ et par définition une surapproximation de son plus petit point fixe : $\mu\, f_{x^\sharp} \sqsubseteq^\sharp_{\mathcal{D}} f^N_{x^\sharp}(\bot)$. De plus, cette valeur peut être simplement calculée en itérant des calculs de $f_{x^\sharp}$ à partir de $\bot$ jusqu'à ce qu'un point fixe soit atteint.

Malheureusement, de nombreux domaines abstraits intéressants ne satisfont pas cette condition auquel cas la suite $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ peut très bien ne pas converger en un nombre fini d'étapes. C'est par exemple le cas du domaine abstrait des intervalles défini dans l'Exemple 5 (la suite $([0,n])_{n\in\mathbb{N}}$ n'est pas stationnaire par exemple). Pour pallier à ce problème, la convergence de la suite $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ va être « accélérée » par un *opérateur d'élargissement.*

**Definition 12** (Opérateur d'élargissement)**.** *Une fonction $\nabla_{\mathcal{D}} : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ est un élargissement pour le domaine abstrait $\mathcal{D}^\sharp$ s'il satisfait les deux conditions suivantes :*

– $\nabla_{\mathcal{D}}$ *est une surapproximation de la borne supérieure* $\sqcup^\sharp_{\mathcal{D}}$ *:*

$$\forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp, x^\sharp \sqcup^\sharp_{\mathcal{D}} y^\sharp \sqsubseteq^\sharp_{\mathcal{D}} x^\sharp \nabla_{\mathcal{D}} y^\sharp;$$

– *pour toute suite* $\left(x^\sharp_n\right)_{n\in\mathbb{N}}$*, la suite* $\left(y^\sharp_n\right)_{n\in\mathbb{N}}$ *définie comme*

$$\begin{cases} y^\sharp_0 := x^\sharp_0 \\ y^\sharp_{n+1} := y^\sharp_n \nabla_{\mathcal{D}} x^\sharp_{n+1} \end{cases}$$

*est stationnaire.*

**Example 7** (Élargissement pour le domaine des intervalles)**.** *En supposant* $\nabla_I : I \times I \to I$ *défini comme :*

$$x \nabla_I y = \begin{cases} y & \text{si } x = \bot_I \\ x & \text{si } y \sqsubseteq^\sharp_{\mathcal{I}} x \\ [a, +\infty] & \text{sinon, si } x = [a, b],\ y = [c, d] \text{ et } a \leqslant c \\ [-\infty, b] & \text{sinon, si } x = [a, b],\ y = [c, d] \text{ et } d \leqslant b \\ [-\infty, +\infty] & \text{sinon} \end{cases}$$

*l'opérateur suivant est un élargissement pour le domaine des intervalles (l'ensemble des variables* $\mathbb{V}$ *étant fini) :*

$$x^\sharp \nabla_{\mathcal{I}} y^\sharp = \left(v \mapsto x^\sharp(v) \nabla_I y^\sharp(v)\right).$$

En utilisant un élargissement, la suite $\left(y^\sharp_n\right)_{n\in\mathbb{N}}$ définie comme $y^\sharp_0 := \bot$ et $y^\sharp_{n+1} := y^\sharp_n \nabla_{\mathcal{D}} f_{x^\sharp}(y^\sharp_n)$ est stationnaire ce qui permet de calculer une surapproximation de la sémantique abstraite (4).

**Definition 13** (Sémantique abstraite)**.** *La sémantique abstraite des énoncés est définie par induction structurelle sur les énoncés :*

$$\begin{array}{l} [\![s_1; s_2]\!]^\sharp(x^\sharp) := [\![s_2]\!]^\sharp\left([\![s_1]\!]^\sharp(x^\sharp)\right) \\ [\![\textbf{if } \mathrm{e} \leqslant \mathrm{r} \textbf{ then } \mathrm{s}_1 \textbf{ else } \mathrm{s}_2 \textbf{ fi}]\!]^\sharp(x^\sharp) := \\ \qquad\qquad [\![s_1]\!]^\sharp\left([\![e \leqslant r]\!]^\sharp(x^\sharp)\right) \sqcup^\sharp [\![s_2]\!]^\sharp\left([\![-e \leqslant -r]\!]^\sharp(x^\sharp)\right) \\ [\![\textbf{while } \mathrm{e} \leqslant \mathrm{r} \textbf{ do } \mathrm{s} \textbf{ od}]\!]^\sharp(x^\sharp) := [\![-e \leqslant -r]\!]^\sharp(y^\sharp_\infty(x^\sharp)) \end{array} \tag{5}$$

*où* $y^\sharp_\infty(x^\sharp)$ *est la limite de la suite stationnaire définie par* $y^\sharp_0(x^\sharp) := \bot$ *et* $y^\sharp_{n+1}(x^\sharp) := y^\sharp_n(x^\sharp) \nabla \left(x^\sharp \sqcup^\sharp [\![s]\!]^\sharp\left([\![e \leqslant r]\!]^\sharp(y^\sharp_n(x^\sharp))\right)\right)$*.*

**Property 3** (Correction de la sémantique abstraite)**.** *En supposant que les opérateurs abstraits sont corrects, cette sémantique est alors correcte par rapport à la sémantique concrète :*

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![s]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![s]\!]^\sharp(x^\sharp)\right).$$

*Démonstration.* Par induction structurelle sur les énoncés $s$. □

**Remark 4** (Inutilité de $x^\sharp \sqcup^\sharp .$)**.** *Bien que rien ne l'exige dans la définition, il est très courant que* $\bot \nabla x^\sharp = x^\sharp$ *et que la sémantique abstraite satisfasse* $[\![.]\!]^\sharp(\bot) = \bot$*. Dans ce cas, les premier termes de la suite* $y^\sharp(x^\sharp)$ *dans la Définition 13 sont* $y^\sharp_0(x^\sharp) = \bot$ *et* $y^\sharp_1(x^\sharp) = x^\sharp$*.*

*De plus, l'élargissement est souvent tel que pour tout $x^\sharp$, $y^\sharp$ et $z^\sharp \in \mathcal{D}^\sharp$, si $z^\sharp \sqsubseteq^\sharp x^\sharp$ alors $x^\sharp \nabla \left(z^\sharp \sqcup^\sharp y^\sharp\right) = x^\sharp \nabla y^\sharp$. Dans ce cas, la définition des termes suivants de la suite se ramène à $y^\sharp_{n+1}(x^\sharp) = y^\sharp_n(x^\sharp) \nabla \left(\llbracket s \rrbracket^\sharp \left(\llbracket e \leqslant r \rrbracket^\sharp \left(y^\sharp_n(x^\sharp)\right)\right)\right)$. Cette remarque peut être utilisé pour simplifier le calcul des sémantiques abstraites des boucles.*

**Example 8** (Une analyse avec le domaine des intervalles)**.** *Cet exemple va détailler l'analyse, i.e., le calcul de la sémantique abstraite, du programme suivant avec le domaine des intervalles introduit dans les Exemples 5, 6 et 7:*

```
x := 0; y := ?(0, 12);
while x ⩽ 42 do
  x := x+1;
  y := y−x
od.
```

*L'analyse commence de* $(a) := \top_{\mathcal{I}}$ *et calcule tout d'abord*

$$(b) := \llbracket x := 0 \rrbracket^\sharp(a) = \{\, x \mapsto [0,0], y \mapsto [-\infty, +\infty] \,\}$$

*puis*

$$(c) := \llbracket y := ?(0,12) \rrbracket^\sharp(b) = \{\, x \mapsto [0,0], y \mapsto [0,12] \,\}.$$

*Puisque l'élargissement des intervalles défini dans l'Exemple 7 remplit les exigences de la Remarque 4, l'analyse continue en calculant*

$$\begin{aligned}(d) &:= (c) \nabla \left(\llbracket x := x+1; y := y-x \rrbracket^\sharp \left(\llbracket x \leqslant 42 \rrbracket^\sharp(c)\right)\right)\\ (e) &:= (d) \nabla \left(\llbracket x := x+1; y := y-x \rrbracket^\sharp \left(\llbracket x \leqslant 42 \rrbracket^\sharp(d)\right)\right)\\ &\vdots\end{aligned}$$

*jusqu'à ce qu'un point fixe soit atteint. D'où :*

$$\begin{aligned}(d) &= \{\, x \mapsto [0,0], y \mapsto [0,12] \,\} \nabla \{\, x \mapsto [1,1], y \mapsto [-1,11] \,\}\\ &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}\\ (e) &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\} \nabla \{\, x \mapsto [1,+\infty], y \mapsto [-\infty,11] \,\}\\ &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}.\end{aligned}$$

*Un point fixe est atteint et la sémantique abstraite après la boucle est finalement*

$$(f) := \llbracket -x \leqslant -42 \rrbracket^\sharp(e) = \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}.$$

*$(e)$ montre que $x$ reste toujours positif en tête de boucle et que $y$ reste toujours inférieur à 12.*

Ce chapitre a donné un bref aperçu du cadre de l'interprétation abstraite. De nouvelles analyses peuvent être définies dans ce cadre de façon agréable, « simplement » en définissant un nouveau domaine abstrait, c'est à dire en donnant : (1) un treillis complet définissant le domaine, avec une fonction de concrétisation donnant un sens à ses valeurs abstraites ; (2) des opérateurs abstraits, simulant leurs alter egos concrets ; (3) un opérateur d'élargissement, pour garantir la terminaison des analyses. Prouver la monotonicité de la fonction de concrétisation, la correction des opérateurs abstraits et la correction ainsi que la propriété de terminaison de l'élargissement garanti alors la correction et la terminaison de l'analyse. Ce sera l'objet principal des Parties III et IV de ce document.

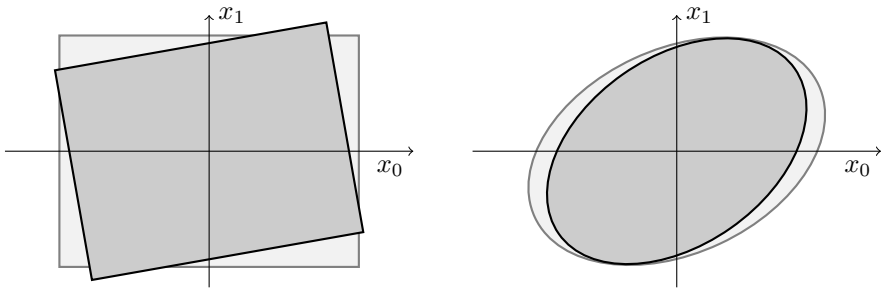
# État de l'art

Ce chapitre vise à passer brièvement en revue quelques méthodes d'analyse statique actuellement disponibles pour borner le coeur numérique de systèmes de contrôle. Les méthodes sont d'abord introduites avec leurs forces et faiblesses connues. Puis quelques indications sont données sur la façon dont ce travail va essayer d'aborder certaines de ces difficultés.

## Domaines linéaires

La plupart des systèmes de contrôle sont basés sur un coeur linéaire. C'est par exemple le cas du régulateur gaussien quadratique linéaire donné dans l'Exemple 4, page xxiii. Malheureusement, ceux cis sont difficiles à analyser en utilisant des domaines abstraits linéaires simples, tels le domaine des intervalles vu précédemment dan l'Exemple 5, page xxvi (par exemple, le régulateur susmentionné n'admet pas d'invariant non trivial dans ce domaine). La Figure 5a donne une intuition de ce problème. Une propriété d'intervalles sur deux variables subit une petite rotation suivie d'une homothétie de facteur $0.92 < 1$ (i.e., une transformation linéaire strictement contractante), montrant que la propriété n'est pas inductive.

Néanmoins, comme le savent les automaticiens depuis longtemps, les systèmes linéaires stables admettent un invariant quadratique (appelé fonction de Lyapunov [BEEFB94, Lya47]). De tels invariants peuvent être dessinés comme des ellipsoïdes. Sur la Figure 5b, un ellipsoïde est dessiné avec son image par la même transformation linéaire que précédemment. Cette fois, la propriété apparaît inductive.




(a) intervalles : non inductifs (b) une propriété quadratique : inductive

FIGURE 5 – Intervalles et propriété quadratique pour une transformation linéaire.

En pratique, quand un invariant quadratique existe, l'approximer avec assez de facettes peut donner un invariant dans les domaines abstraits linéaires classiques comme les polyèdres [CH78] ou les zonotopes [GGP09]. Ainsi, on pourrait penser que les invariants quadratiques sont inutiles et que les même résultats peuvent être obtenus en utilisant seulement des invariants linéaires. Toutefois, cette approche souffre de deux inconvénients, souvent rédhibitoires, qui en font un argument essentiellement théorique :

**Grand nombre de facettes** « assez de facettes » peut être bien trop pour être effectivement réalisable, en particulier quand le nombre de variables augmente. Ce problème devient encore plus sérieux en présence de transformations faiblement contractantes, qui nécessitent intuitivement que l'invariant linéaire soit assez « doux » pour être inductif, donc composé d'un grand nombre de facettes. Plus que l'espace mémoire nécessaire pour stocker ces objets, le coût de leur manipulation peut rendre l'approche totalement irréaliste. Comparé aux invariants linéaires, les invariants quadratiques ont une complexité spatiale quadratique en le nombre de variables et sont intrinsèquement « doux ».

**Inefficacité des itérations de Kleene** L'existence d'un invariant ne signifie pas pour autant l'existence d'un moyen pratique de le calculer. En particulier, les itérations de Kleene avec des polyèdres sont connues pour mal se comporter quand on essaye de générer de tels invariants linéaires[SB13]. De plus, aucune des stratégies classiques d'élargissement ne permet de générer de tels résultats sans réaliser un très grand nombre d'itérations.

## Déroulage

Le déroulage constitue une alternative pratique à la recherche d'invariants linéaires avec une myriade de facettes. Pour des systèmes purement linéaires, dérouler jusqu'à une profondeur $k$ allant de quelques centaines à quelque milliers permet de calculer des invariants $k$-inductifs précis tout en conservant leur nombre de facettes raisonnablement bas [Fer04, GGP09]. Des travaux récents [SB13] démontrent même que des bornes précises (i.e., les valeurs maximales atteintes) peuvent être calculées avec de simples fonction support en déroulant totalement le système. Intuitivement, dérouler transforme un système contractant en un autre plus contractant. Ainsi des propriétés qui ne sont pas inductives peuvent apparaître $k$-inductives. Ceci est illustré sur la Figure 6.

Ces résultats sont certainement intéressants mais ne produisent que des invariants $k$-inductifs pour de grandes valeurs de $k$, qui présentent les défauts suivants :

**Vérifier les résultats** Si l'utilisateur ne fait pas confiance à l'analyseur et souhaite vérifier ses résultats à posteriori, ne pas avoir un invariant inductif peut grandement lui compliquer la tâche. Dans le cas extrême où le système est entièrement déroulé, cela peut même revenir à rejouer l'analyse en entier.

**Difficulté à dérouler en présence de gardes** Dérouler des systèmes purement linéaires fonctionne bien car la taille du système déroulé est linéaire en la profondeur de déroulage $k$. Toutefois, quand le système contient des gardes, les choses peuvent devenir plus complexes. En considérant

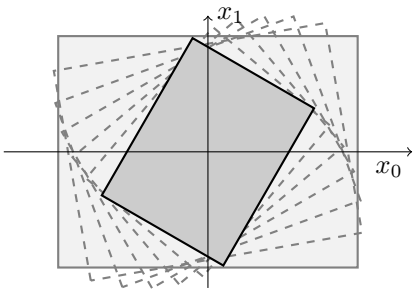



FIGURE 6 – Les intervalles de la Figure 5 sont 6-inductifs.

tous les chemins à travers $k$ itérations peut mener à un système de taille exponentielle $2^k$ ce qui devient rapidement irréaliste pour de grands $k$.

## Invariants quadratiques

Bien que les invariants quadratiques sont connus depuis longtemps en tant que *fonctions de Lyapunov* quadratiques par les automaticiens [BEEFB94, Lya47], leur utilisation en analyse statique ne date pas de plus de dix ans.

Le premier usage célèbre d'invariants quadratiques pour l'analyse statique fut l'usage d'ellipses (ellipsoïdes de dimension deux) pour borner des filtres du second ordre [Fer04, Fer05, Mon05].

**Definition 14** (Filtres d'ordre $n$)**.** *Un filtre d'ordre $n$ (ou filtre du $n$-ième ordre) est un cas particulier de système linéaire dans lequel une variable $x$ reçoit une combinaison linéaire de ses $n$ valeurs précédentes et d'une entrée $y$ avec ses $n$ valeurs précédentes. Écrit comme un programme, la boucle prend la forme suivante :*

**while** $-1 \leqslant 0$ **do**
  y := ?(−1, 1);
  x := $a_1$x1+ $\cdots$ + $a_n$xn+$b_0$y+$b_1$y1+ $\cdots$ + $b_n$yn;
  xn := x(n−1);. . . ; x2 := x1; x1 := x;
  yn := y(n−1);. . . ; y2 := y1; y1 := y;
**od**.

**Property 4** (Borner les filtres du second ordre)**.** *En supposant $a_1^2 + 4a_2 < 0$ et $|x - (a_1x_1 + a_2x_2)| \leqslant b$, si $x_1^2 - a_1x_1x_2 - a_2x_2^2 \leqslant r$ alors $x^2 - a_1xx_1 - a_2x_1^2 \leqslant (\sqrt{-a_2r} + b)^2$. Pour un filtre du second ordre, si $a_1^2 + 4a_2 < 0$ et $\sqrt{-a_2} < 1$, cela permet d'inférer l'invariant inductif $x_1^2 - a_1x_1x_2 - a_2x_2^2 \leqslant r$ où $r$ vérifie $(\sqrt{-a_2r} + b)^2 \leqslant r$ and $b = |b_0| + |b_1| + |b_2|$.*

**Example 9.** *En considérant le filtre du second ordre suivant*

x1 := 0; x2 := 0;
y1 := 0; y2 := 0;
**while** $-1 \leqslant 0$ **do**
  y := ?(−1, 1);
  x := 1.5x1−0.7x2+0.5y−0.7y1+0.4y2;
  x2 := x1; x1 := x;
  y2 := y1; y1 := y;
**od**,

*$a_1$ et $a_2$ vérifient $a_1^2 + 4a_2 = 1.5^2 - 4 * 0.7 < 0$ et $\sqrt{-a_2} = \sqrt{0.7} < 1$. De plus, $|x - (a_1x_1 + a_2x_2)| \leqslant b$ avec $b = |b_0| + |b_1| + |b_2| = 1.6$ puisque $y$, $y_1$ et $y_2$ se trouvent tous dans l'intervalle $[-1, 1]$. L'invariant suivant est alors inféré*

$$x_1^2 - 1.5x_1x_2 + 0.7x_2^2 \leqslant 95.96$$

*signifiant que $|x| \leqslant 22.11$.*

*Il est intéressant de noter que cette borne est assez large puisque $|x|$ ne peut pas dépasser 1.42. D'autres invariants quadratiques sur $x_1$ et $x_2$ (i.e., des ellipsoïdes de dimension 2) pourraient donner des bornes plus fines ($|x| \leqslant 16.14$) mais l'essentiel du conservatisme vient du fait que $0.5y - 0.7y_1 + 0.4y_2$ est simplement abstrait par $[-1.6, 1.6]$, oubliant totalement le fait que $y_1$ est la valeur précédente de $y$ et $y_2$ la valeur précédente de $y_1$ (en fait, une fois cette abstraction faite, il devient impossible de prouver une meilleure borne que $|x| \leqslant 14.84$).*

Des bornes sur les filtre d'ordre $n$ peuvent alors être calculées en les décomposant en filtres d'ordre 1 (bornés avec des intervalles) et 2 (bornés par la méthode précédente) et en raffinant les bornes obtenues au moyen d'une sorte de déroulage [Fer04, Fer05, Mon05]. La méthode présente donc les même avantages (précision des invariants calculés) et inconvénients (aucun invariant inductif n'est produit) que les autres méthodes de déroulage.

D'autres travaux proposent de calculer des invariants quadratiques inductifs de dimension supérieure sur des classes plus larges de systèmes linéaires [AGG10, AFP09, GS10, RFM05]. De tels invariants sont calculés grâce à des solveurs numériques nommés solveurs de programmation semi-définie.

**Example 10.** *Sur l'exemple précédent, un ellipsoïde de dimension 4 (sur les variables $x_1$, $x_2$, $y_1$ et $y_2$) constitue un invariant inductif donnant la borne $|x| \leqslant 1.64$.*

**Remark 5** (L'espace d'états accessibles exact n'est pas un ellipsoïde)**.** *Malgré la capacité des invariants quadratiques à borner n'importe quel système linéaire stable, il faut noter que l'espace d'états accessibles de tels systèmes n'est habituellement pas un ellipsoïde. Ainsi, bien que les ellipsoïdes sont de bons invariants, ils ne produiront pas toujours les bornes les plus fines possibles.*

**Example 11** (L'espace d'états accessibles exact n'est pas un ellipsoïde)**.** *En considérant la suite définie par $x_0 := 0$ et $x_{k+1} := Ax_k + Bu_k$ où*

$$A := \begin{pmatrix} 0.92565 & -0.0935 \\ 0.00935 & 0.935 \end{pmatrix} \qquad B := \begin{pmatrix} 1 \\ 0 \end{pmatrix}$$

*et pour tout $k \in \mathbb{N}$, $\|u_k\|_\infty \leqslant 1$. L'espace d'états accessibles de ce système, dessiné sur la Figure 7, n'est pas un ellipsoïde.*

*Démonstration.* Supposons que l'espace d'états accessibles $R$ est un ellipsoïde. Prenons $x' \in R$ tel que $\pi_1(x')$ est maximal (avec $\pi_1(x')$ la projection sur la seconde composante de $x'$). Alors il existe $x \in R$ et $u \in \mathbb{R}$ tels que $x' = Ax + Bu$. Supposons (sans perte de généralité grâce à la symétrie) $u \geqslant 0$, alors $x'' := Ax + B(u - 1) \in R$ et $\pi_1(x'') = \pi_1(x')$. Finalement, par convexité stricte de l'ellipsoïde $R$, il existe $\lambda \in \mathbb{R}$ tel que $\lambda > 1$ et $\lambda\left(\frac{x'+x''}{2}\right) \in R$. Mais $\pi_1\left(\lambda\left(\frac{x'+x''}{2}\right)\right) = \lambda\pi_1(x') > \pi_1(x')$ ce qui contredit sa maximalité. □

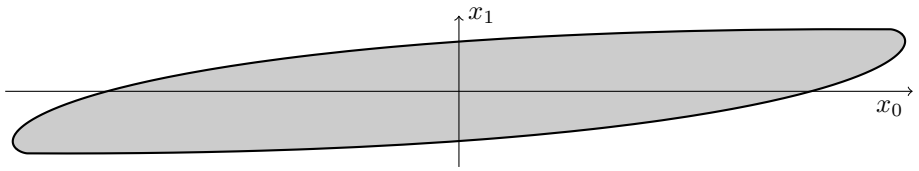



FIGURE 7 – Espace d'états accessibles de l'Exemple 11.

## Itérations sur les politiques sur des domaines à gabarits

Cette section présente les qualités des ellipsoïdes pour borner des systèmes linéaires. Pourtant, ils souffrent d'un défaut de taille comparés aux domaines abstraits classiques tels les polyèdres, les octogones, les zonotopes,...: l'ensemble des ellipsoïdes muni de l'ordre inclusion $\subseteq$ n'est pas un treillis. En effet, il n'existe généralement pas de plus petit ellipsoïde contenant deux autres ellipsoïdes donnés (i.e., pas de borne supérieure). Ceci est illustré sur la Figure 8 et empêche le calcul effectif par des itérations de Kleene sur l'ensemble des ellipsoïdes.

Une solution habituelle est de choisir — avant toute analyse — une *forme* d'ellipsoïde et de ne calculer que son *rayon*, comme illustré sur la Figure 9. L'ensemble des rayons $(\mathbb{R} \cap [0, +\infty[) \cup +\infty$ constitue alors un treillis complet. Cette idée est généralisée à travers la notion de *domaines à gabarits*.

**Definition 15** (Domaines à gabarits)**.** *Étant donné un ensemble fini* $\{ t_1, \ldots, t_n \}$ *d'expressions sur les variables* $\mathbb{V}$*, le domaine à gabarit* $\mathcal{T}$ *est définie comme* $\overline{\mathbb{R}}^n = (\mathbb{R} \cup \{ -\infty, +\infty \})^n$ *et une valeur abstraite* $(b_1, \ldots, b_n) \in \mathcal{T}$ *représente tous les environnements* $\gamma_{\mathcal{T}}(b_1, \ldots, b_n) = \{\rho \in (\mathbb{V} \to \mathbb{R}) \mid [\![t_1]\!](\rho) \leqslant b_1, \ldots, [\![t_n]\!](\rho) \leqslant b_n\}$*. En d'autres termes, la valeur abstraite* $(b_1, \ldots, b_n)$ *représente tous les environnements vérifiant les contraintes* $t_i \leqslant b_i$*.*

En fait, de nombreux domaines abstraits courants sont des domaines à gabarits. Par exemple le domaine des intervalles de l'Exemple 5 est obtenu avec les gabarits $x_i$ et $-x_i$ pour chaque variable $x_i \in \mathbb{V}$ et le domaine des octogones [Min01] en ajoutant tous les $\pm x_i \pm x_j$.

Néanmoins, le calcul d'invariants précis sur des programmes numériques peut être dur à réaliser en utilisant des itérations de Kleene classiques avec élargissement. L'itération sur les politiques [AGG10, CGG+05, GS07a, GS10, GSA+12] est une des alternatives au simple élargissement développée durant la dernière décennie [BSC12, FG10, GR06, HH12, SJ11]. Cette technique permet de calculer des post point fixes précis, habituellement en se basant sur des solveurs d'optimisation mathématique comme les outils de programmation linéaire ou semi-définie. De telles techniques ont été récemment développées pour le calcul d'invariants quadratiques pour les systèmes linéaires [AGG10, GS10, GSA+12].

La forme des gabarits à considérer pour l'itération sur les politiques dépend de l'outil d'optimisation utilisé. Par exemple, la programmation linéaire [GGTZ07, GS07b] permet n'importe quel gabarit linéaire tandis que les gabarits quadratiques peuvent être pris en charge grâce à la programmation semi-définie et une relaxation appropriée [AGG10, GS10, GSA+12].

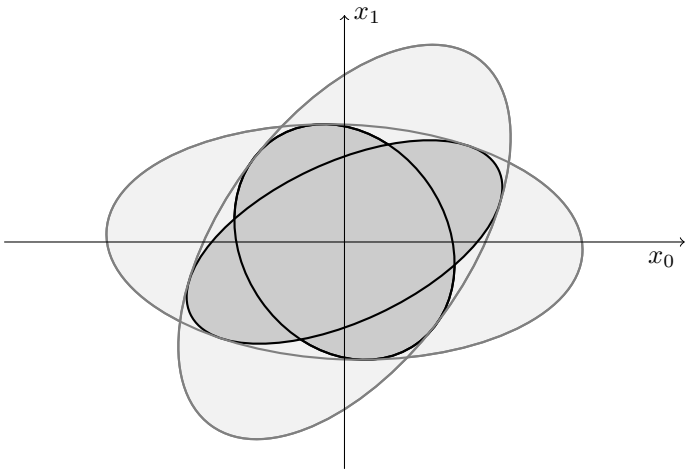



FIGURE 8 – Il n'existe en toute généralité pas de plus petit ellipsoïde contenant deux ellipsoïdes donnés : les deux ellipsoïdes gris clair contiennent tous deux les deux ellipsoïdes gris foncé mais aucun des deux n'est inclus dans l'autre.

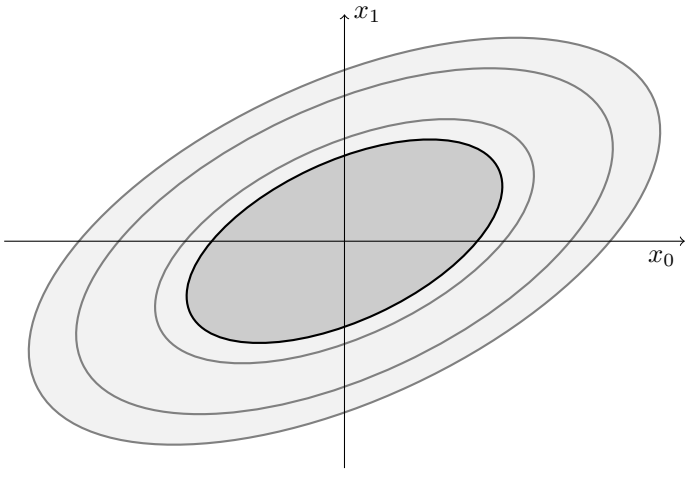



FIGURE 9 – Ellipsoïdes de même forme et de rayons différents.

**Example 12.** *Pour borner les variables du programme de la Figure 4, page xxiv, le gabarit quadratique* $t_1 := 6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_0x_2 + 2.4182x_1x_2$ *est utilisé. Les gabarits* $t_2 := x_0^2$, $t_3 := x_1^2$ *et* $t_4 := x_2^2$ *sont ajoutés de façon à obtenir des bornes sur chaque variable. En utilisant ces gabarits, l'itération sur les politiques calcule l'invariant* $(1.0029, 0.1795, 0.1136, 0.2757) \in \mathcal{T}$, *signifiant :* $t_1 \leqslant 1.0029 \wedge x_0^2 \leqslant 0.1795 \wedge x_1^2 \leqslant 0.1136 \wedge x_2^2 \leqslant 0.2757$ *ou de manière équivalente :* $t_1 \leqslant 1.0029 \wedge |x_0| \leqslant 0.4236 \wedge |x_1| \leqslant 0.3371 \wedge |x_2| \leqslant 0.5251$. *C'est à dire un ellipsoïde rogné comme affiché sur la Figure 10, page xl.*

Une introduction plus approfondie à l'itération sur les politiques peut être trouvée au Chapitre 9. Bien que très attrayant, l'utilisation de ces techniques dans des analyseurs statiques basés sur l'interprétation abstraite se heurte à deux obstacles majeurs :

**Besoin de gabarits** Des gabarits adaptés doivent être fournis par l'utilisateur. Typiquement, le gabarit $t_1$ dans l'exemple précédent est non trivial.

**Manque d'intégration avec l'interprétation abstraite** L'approche semble assez orthogonale aux itérations de Kleene avec élargissement classiquement utilisées dans les outils d'interprétation abstraite [Jea10]. Cela gêne la coopération avec d'autres domaines abstraits au travers de produits réduits alors même qu'il a été démontré que cette fonctionnalité joue un rôle clef dans l'un des plus brillant succès de l'interprétation abstraite : dans l'analyseur statique Astrée, les domaines relationnels coûteux (comme les octogones) ne sont appliqués que localement sur quelques variables du programme analysé, communiquant leurs résultats avec des domaines non relationnels moins lourd (comme le domaine des intervalles) ou d'autres instances de domaines relationnels [CCF+06].

Cette thèse tente principalement de répondre à ces deux problèmes en proposant des heuristiques pour générer automatiquement de bon gabarits avant de réaliser l'itération sur les politiques et en intégrant tout cela dans un domaine abstrait classique. Puisque ce domaine abstrait présente la même interface que d'autres domaines relationnels (comme les polyèdres ou le domaine des octogones par exemple), son intégration dans un analyseur statique basé sur l'interprétation abstraite est facilitée, tout comme la communication à travers des produits réduits avec d'autres domaines déjà présents dans l'analyseur.

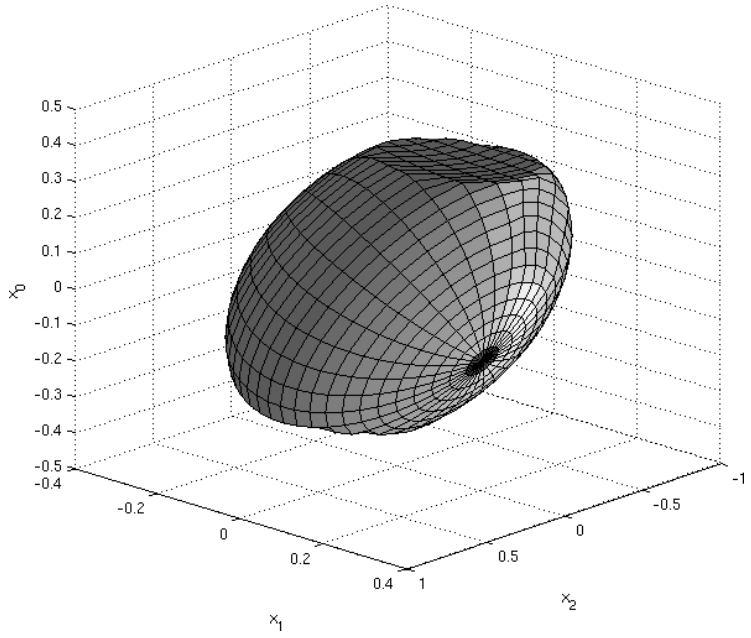



FIGURE 10 – Invariant pour notre exemple filé.

# Résumé des chapitres en anglais

Les Chapitres 1, 2, 3 et 4 sont entièrement traduits en français ci dessus.

Le Chapitre 5 propose des heuristiques basées sur les fonction de Lyapunov quadratiques pour générer automatiquement de bons gabarits quadratiques.

Le Chapitre 6 traite des questions d'implémentation liées aux calculs en virgule flottante. À la fois les erreurs d'arrondi dans le programme analysé et ceux apparaissant dans l'analyseur lui même sont pris en compte.

Le Chapitre 7 présente des résultats expérimentaux pour les idées discutées dans les deux précédents chapitres.

Le Chapitre 8 conclut cette première partie sur les systèmes purement linéaires.

Le Chapitre 9 est une introduction à l'itération sur les politiques. Plus particulièrement l'itération sur les politiques avec des gabarits quadratiques, par les deux méthodes majeures : itérations sur les min et sur les max politiques. Cette technique étant utilisée par la suite pour générer des invariants quadratiques sur des programmes linéaires avec des gardes.

Le Chapitre 10 présente un domaine abstrait symbolique permettant de reconstruire des graphes de flot de contrôle de façon à intégrer les itérations sur les politiques dans un domaine abstrait classique.

Le Chapitre 11 présente l'utilisation du domaine précédent pour construire un domaine abstrait d'itération sur les politiques.

Le Chapitre 12 traite des questions de calcul en virgule flottante dans ce nouveau cadre.

Le Chapitre 13 présente quelques points d'implémentation plus ou moins important pour obtenir des résultats valables ainsi que des résultats expérimentaux.

Le Chapitre 14 conclut cette deuxième partie sur les programmes linéaires avec gardes.

Enfin, le Chapitre 15 présente un domaine abstrait à gabarit polynomiaux pour l'analyse de programmes polynomiaux. Ce travail, beaucoup plus prospectif que le reste de ce document est basé sur des outils d'optimisation globale polynomiale basés sur les polynômes de Bernstein.

Enfin, un dernier chapitre de conclusion est entièrement traduit ci dessous.

# Conclusion et perspectives

Le but de cette thèse était d'analyser des systèmes de contrôle commande. La plupart d'entre eux, y compris ceux dans les commandes de vol des avions, sont linéaires ou linéaires par morceaux. Il est bien connu des automaticiens qui développent ces contrôleurs qu'ils sont stables si et seulement si ils admettent un invariant quadratique, c'est à dire un ellipsoïde. Toutefois, la plupart des techniques d'analyse statique actuellement disponibles se basent sur des invariants linéaires. Pour donner des résultats dignes d'intérêt sur ces systèmes, elles nécessitent généralement de lourds déroulages, ce qui peut devenir problématique en présence de gardes qui apparaissent souvent du fait de l'ajout de choses comme des saturations ou des antiwindups aux contrôleurs purement linéaires. Quelques méthodes d'inférence d'invariants quadratiques existent mais sont soit restreintes à une classe réduite de systèmes ou ne proposent qu'un niveau limité d'automaticité, nécessitant que des gabarits appropriés soient fournis avant toute analyse.

Les deux premières parties de cette thèse (Parties II et III) proposent une méthode entièrement automatique d'inférence d'invariants quadratiques pour les systèmes purement linéaires et certains systèmes linéaires par morceaux avec gardes. Ceci est réalisé grâce à la technique d'*itération sur les politiques* utilisant des solveurs numériques de programmation semi-définie pour calculer de *propriétés quadratiques*. Cette technique fait face à deux inconvénients majeurs. Tout d'abord, les itérations sur les politiques semblent assez orthogonales à l'interprétation abstraite classique. Ensuite, elle nécessitent que des gabarits appropriés soient fournis avant toute analyse. La Partie III tente de répondre au premier problème en intégrant l'itération sur les stratégies dans un domaine abstrait classique tandis que la Partie II propose une heuristique de génération de gabarits pour répondre au second. Cette heuristique est basée sur des idées d'automatique, à savoir les *fonctions de Lyapunov quadratiques*.

Ainsi, en partant d'un code source, le domaine abstrait implémenté extrait d'abord un graphe de flot de contrôle de la partie du code ciblée par l'analyse (le reste étant abstrait à travers des produits réduits avec d'autres domaines abstraits comme le domaine des intervalles). À partir de ce graphe, un gabarit quadratique approprié peut être automatiquement calculé ce qui permet à l'itération sur les stratégies de calculer un invariant. Cet invariant peut finalement être projeté sur des intervalles par exemple et partagé avec d'autres domaines par produit réduit.

Pour autant que les auteurs sachent, notre analyseur statique, utilisant le domaine abstrait en résultant, est le seul capable d'inférer entièrement automatiquement des invariants sur certains de nos jeux de test qui sont de plus

souvent assez serrés[33]. De plus, l'interface de domaine abstrait classique devrait permettre une intégration facile dans tout analyseur statique basé sur le cadre de l'interprétation abstraite et une coopération facile avec des domaines abstraits déjà existants, à travers des produits réduits. Les temps d'analyse se sont avérés rester raisonnables sur de petits exemples. Bien sûr, cela ne passerait pas à l'échelle sur des systèmes énormes mais il pourrait être possible de gérer de véritables systèmes critiques en se concentrant sur de tels petit systèmes qui les composent.

Les solveurs numériques utilisés étant implémentés — pour des raisons d'efficacité — avec des calculs en virgule flottante, ils n'offrent aucune garantie de correction de leur résultat. Une vérification à posteriori a donc été mise au point qui s'avère efficace sur nos cas de test. Une méthode pour prendre en compte les erreurs d'arrondis dans le programme analysé a également été schématisée. Ces deux aspects sont parfois passés sous silence dans certain travaux liés mais sont indispensables pour obtenir une pleine confiance dans la correction des invariants synthétisés.

En ce qui concerne les perspectives, l'heuristique de génération de gabarits proposée considère différents comportements du contrôleur analysé (saturé, non saturé, remise à zéro,...) indépendamment. Comme vu dans nos cas de test, cela peut parfois échouer. Des modèles de systèmes en boucle fermée (i.e., incluant à la fois le contrôleur et le système physique); constituent des cas intéressants pour lesquels cela apparaît inapproprié, puisque les systèmes physiques sont souvent instables, menant à des comportements problématiques dès qu'une saturation est introduite. Des travaux futures devraient s'intéresser à ce problème. Une meilleure heuristique de génération de gabarits pourrait permettre de réaliser des preuves de stabilité en boucle fermée. Toutefois, il y existe un autre obstacle à de telles preuves puisque les automaticiens ne possèdent qu'un modèle continu du système physique qui doit être discrétisé, bien que cette discrétisation puisse être effectuée à la main dans un premier temps.

De plus, il pourrait être intéressant d'étudier des extensions de la méthode à d'autres propriétés que la stabilité comme la performance et la robustesse. Bien que les propriétés de performance puissent sembler assez différentes des propriétés de stabilité, c'est bien moins le cas pour la robustesse. En effet, une preuve de stabilité peut être réalisée avec des incertitudes sur certains paramètres, devenant de fait une preuve de robustesse.

Il serait également intéressant de prouver la partie la plus critique de notre domaine abstrait en utilisant un assistant de preuve comme Coq. Comme déjà noté, pour établir la correction de l'analyse, seule l'étape finale de vérification devrait être prouvée. En particulier, il n'y a rien à prouver au sujet des itérations sur les politiques ou de la programmation semi-définie. Ainsi, bien qu'important, le volume de travail nécessaire pourrait rester dans un périmètre raisonnable.

D'un point de vue pratique, le domaine abstrait pourrait être intégré dans un interprète abstrait plus mature comme IKOS[34]. Il pourrait aussi être plus finement intégré dans le model checker basé SMT développé à l'ONERA comme

[33] Il serait toutefois intéressant de comparer ces résultats avec ceux donnés par d'autres analyseur statiques à l'état de l'art. Par exemple, l'interprète abstrait Fluctuat (http://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html) pourrait retourner de meilleurs résultats sur certains exemples grâce à ses mécanismes de déroulage.

[34] http://ti.arc.nasa.gov/opensource/ikos/

un oracle de synthèse d'invariants qui pourrait grandement l'aider en fournissant des invariants numériques difficiles à obtenir par des méthodes symboliques.

À plus long terme et d'un point de vue plus théorique, la collaboration fructueuse avec les automaticiens pourrait être poursuivie, en essayant d'importer en analyse statique des notions comme les fonctions de Lyapunov communes ou les fonctions de Lyapunov quadratique par morceaux linéaires [JR98] qui pourraient permettre d'analyser des systèmes plus complexes.

Une autre direction serait de cibler des systèmes plus que linéaires (ou linéaires par morceaux). Dans ce but, la dernière partie du document (Partie IV) propose un domaine abstrait à gabarits pour inférer des invariants polynomiaux sur des programmes polynomiaux. C'est certainement un travail très prospectif. Tout d'abord, la méthode doit être mieux testée sur plus d'exemples. Ensuite, la question cruciale du choix des gabarits doit encore être traitée, peut être en regardant du côté de méthodes connues des automaticiens, basées sur des techniques de sommes de carrés [KHJ13]. Finalement, des fonctions autres que les fonctions polynomiales pourraient être étudiées au travers d'expansion de Taylor ou de Poisson par exemple.

# Part I

# Context

# Chapter 1

# Control-Command Critical Systems

Digital flight commands are now widespread in large commercial aircrafts. These control command systems are said critical in the sense that a failure could have serious impacts, particularly in terms of human lives, hence their stringent validation demands and a strong interest in formal proofs of correctness. This chapter will first give a brief historical reminder of the introduction of control command systems in aircraft designs. After what, the computer scientist reader can find a very succinct overview of what control command systems look like, a few words about their critical aspect, hence the need for proofs, and eventually the kind of properties that have to be proven. Finally, the contributions that are to be found in the remaining of the document are introduced.

**Historically,** from the first flight of the Wright Flyer in 1903, pilot controlled their aircraft through cables mechanically linking control commands (stick and rudder pedals) to the flight control surfaces (ailerons for the longitudinal axis (roll)[1], elevators for the lateral axis (pitch) and rudder for the vertical axis (yaw)) which actually move the aircraft by disturbing the airflow around it. With this system, the larger the aircraft, the larger the control surfaces and the stronger the pilot has to act on her or his stick to move them, although well-balanced surfaces (with little masses) and trim tabs (little control surfaces on the edge of the main control surfaces moving in reverse direction to help moving them with less manual force)[2] allow to save pilot forces. Little general aviation aircraft still use such simple mechanisms[3] but this became a major obstacle to the development of larger commercial aircrafts in the second half of the twentieth century. Hydraulic actuators were then installed to help pilots move the control surfaces, just like power steering later introduced in cars.

[1] Although, to be precise, the Wright Flyer was not equipped with ailerons (hinged control surfaces) but used a wing warping system for longitudinal control. This system was still used on a few early wood and fabric aircrafts (such as the Blériot XI which first crossed the English Channel) on which wings were flexible enough but was soon abandoned with the use of stiffer structures. Paradoxically enough, a similar technique is now again studied by NASA teams.

[2] They were developed by the German engineer Anton Flettner, hence their name in french.

[3] This is even the case of the Italo-French twin turboprop aircrafts ATR, the most fuel efficient aircrafts in their range (short haul regional flights) thanks to their relative rusticity.

This then enabled to introduce a calculator between the pilot controls and the actuators. The first equipped commercial aircraft was the Anglo-French supersonic Concorde, which made is maiden flight in 1969, whereas the first mass product military aircrafts with electrical flight commands were the F15, F16 and Mirage 2000 in the seventies. Such electrical flight commands allow inherently unstable aircraft designs which a human being would otherwise be unable to pilot. In the field of combat aircrafts, this gave birth to planes with incredible maneuverability, while relaxing the stability of commercial aircrafts allows to reduce the size of the control surfaces, decreasing both their weight and drag which leads to improved fuel efficiency. The flight command calculators embedded in these aircrafts were nonetheless analog[4]. In 1987, less than twenty years after Concorde, the Airbus A320 performed its first flight, being the first commercial aircraft with digital flight commands[5]. Boeing followed the same path less than ten years later with the maiden flight of its B777 in 1994 and nowadays the design of a large commercial aircraft cannot be envisioned anymore without digital flight commands. They even begin to be introduced in business jets with the Dassault Falcon 7X, flying for the first time in 2005, and in helicopters with the NH90 in 2003[6].

Digital flight commands can allow better numerical accuracy and above all increase the flexibility of the system by replacing the specially designed hardware of analog calculators with software running on off-the-shelf processors. This allows to add new features among which the most iconic is probably the flight envelope protection. This system prevents the aircraft from being put in dangerous postures whatever the pilot do with its stick. In particular, this allows them to react quickly in an emergency situation without having to take care not to exceed the structural or aerodynamic limits of their aircraft. Thanks to their flight envelope protection, it is sometimes said from modern Airbuses that they cannot stall[7]. This is sometime criticized for the following perverse effect: since the plane cannot stall, crews are instructed to push the throttle and pull the stick when reaching stall conditions, in order to avoid the ground. However, in rare occurrences, this can be the last advisable thing to do. The lost of the AF447 flight and its 228 occupants is imputed to this effect. After the loss of all velocity sensors (pitot probes had frozen), it became impossible to maintain the flight envelope protection and flight commands switched to an alternate law in which pilot had a more direct control on moving surfaces. Flying at high altitude where the margin between normal flight conditions, overspeed (which could eventually break the aircraft) and stall becomes very narrow[8], the crew, unprepared to such a situation, rapidly made the aircraft stall and eventually perish in the ocean. Nevertheless, besides such exceptional events, flight envelope protection certainly already saved numerous human lives.

[4] Although the calculator controlling air intakes of Concorde already used digital technology.

[5] The space shuttle Entreprise first flew (as a glider) with a digital control system ten years earlier in 1977 but cannot be actually considered as a commercial aircraft.

[6] Although the helicopter first flew in 1995 with classic mechanical commands due to severe delays in the development of the digital flight commands.

[7] When the angle of attack of a wing increases, the airflow on its upper surface can begin to separate from it, inducing a sudden loss of lift. This can result in dramatic losses of altitude for the aircrafts which can then hit the ground when they are too close from it.

[8] Modern jets cruise at high altitudes (more than 10,000m) where the density of the atmosphere is lower in order to reduce the drag hence the fuel consumption. There are also less turbulences at such altitudes which vastly increases passengers comfort.

Digital calculators are now also widespread in other aircraft systems such as the brakes[9] or engines controls[10]. A last example of behavior unexpected by the crew happened during acceptance testing of an Airbus A340-600. During engine tests on ground, the aircraft began moving, the crew attempted both to brake and steer to avoid a concrete deflection wall surrounding the test facility. Unfortunately, steering deactivated brakes of the central part of the main landing gear in order to save tires, the plane was not able to stop before the wall and was destroyed. Paradoxically enough, had the people on board gone straight toward the wall, the plane might have managed to stop before hitting it or would at least have hit it with reduced velocity. Again, in normal conditions, this system probably perfectly does the job of saving tyres.

**Control Command Systems** are designed by control theorists. Before designing a controller or regulator, they first build a model of the physical system they want to regulate. They call it the *plant* and it can either be an inherently stable system whose stabilization is not deemed fast or smooth enough[11] or an actual unstable system they want to stabilize[12]. This can be done by applying the fundamental laws of mechanic to the system and potentially adjusting the result by comparison with the behavior of the actual plant. The resulting model is then usually linearized along an operating point of interest[13]. In practice, this gives matrices $A_p \in \mathbb{R}^{n\times n}$ and $B_p \in \mathbb{R}^{n\times p}$ such that the internal state $x_p \in \mathbb{R}^n$ of the plant[14] follows the differential equation

$$\dot{x}_p = A_p x_p + B_p u_p$$

where $u_p \in \mathbb{R}^p$ is the input of the plant[15]. From this internal state, an output $y_p$ is computed:

$$y_p = C_p x_p + D_p u_p.$$

From this model of the plant, control theorists use various methods to design a controller. In the common case of linear controllers, they would define matrices $A_c$ and $B_c$ such that the internal state $x_c$ of the controller follows the equation

$$\dot{x}_c = A_c x_c + B_c u_c$$

where $u_c$ is the input of the controller, typically coming from input sensors and commands[16] $y_d$ and equal to the output of the plant minus the command

[9] The very last feature, called brake to vacate even allows the pilot to select a runway exit on a map of the airport before landing, leaving the system control the brake process in order to optimize passenger comfort and minimize disc brake attrition.

[10] Usually called FADEC for Full Authority Digital Engine Control, it is also a key factor in fuel efficiency of modern engines by performing a fine control of their numerous parameters that would be difficult to achieve even for the best flight engineer.

[11] Some commercial planes or the hook of a crane suspended under a cable which automatically stabilizes with a vertical cable under its own weight for instance.

[12] For instance fighter aircrafts or an inverted pendulum which can be kept vertical upside down only by carefully moving its base (this find applications for instance in the Segway).

[13] Typically a choice of altitude, velocity, weight and mass distribution for a plane, or the vertical position for the inverted pendulum.

[14] A vector with the position, velocity, angles and angular velocities for example for a plane, or the horizontal position and speed of the base of the inverted pendulum along with its angle and angular velocity for the inverted pendulum.

[15] The position of the control surfaces for a plane or the input voltage of the engine moving the base of the inverted pendulum to stay with our examples.

[16] Stick and rudder pedals of the pilot on a plane, or desired position of the inverted pendulum.

$(y_p - y_d)$. An output is also computed

$$y_c = C_c x_c + D_c u_c.$$

typically going to actuators and equal to the input $u_p$ of the plant. Historically, when the only computing tools available were slide rules, abacus and human assistants[17], controller design were made using graphical tools such as the Bode diagram for instance. This set of tools is referred to as *classic control*. With the advent of electronic computers, it became possible to use optimization procedures which gave birth to techniques such as LQR or $H_\infty$ regulators which can allow to reach better performances[18]. These techniques being referred to as *modern control*.

Eventually, the controller has to be implemented. This can be done mechanically[19], with an analog electronic circuit like in Concorde or with software for digital controllers like in modern airliners. From now on, we focus exclusively on the last case. To be implemented in software, the controller first need to be discretized

$$x_{c_{k+1}} = A'_c x_{c_k} + B'_c u_{c_k}.$$

The value of the internal state $x_c$ is then recomputed at some fixed frequency[20]. This finally gives a code which could looks like the following C code snippet:

```
double x[3] = {0, 0, 0};
double nx[3];
double in;
while (1) {
  in = acquire_input();
  nx[0] = 0.9379*x[0] - 0.0381*x[1] - 0.0414*x[2] + 0.0237*in;
  nx[1] = -0.0404*x[0] + 0.968*x[1] - 0.0179*x[2] + 0.0143*in;
  nx[2] = 0.0142*x[0] - 0.0197*x[1] + 0.9823*x[2] + 0.0077*in;
  x[0] = nx[0]; x[1] = nx[1]; x[2] = nx[2];
  wait_next_clock_tick();  // a tick every 10 ms for instance
}
```

that is a highly numerical code, quite different for instance from the source code of an operating system or a compiler. Of course, the actual flight command controller of an aircraft is something vastly more complicated. Saturations are added on outputs in order to avoid sending to actuators orders out of their range, multiple controllers are designed for a number of flight points (defined by an altitude, a speed, a weight and a mass distribution of the aircraft) and smoothly combined altogether and correctors are added on inputs or outputs to take into account various phenomena. Eventually, the software can require a few hundreds of thousands lines of C code.

Finally, it is worth noting for computer scientists that control theorists would qualify the above controller as having *three states*, since the vector $x_c$ has three components. Obviously, it has an infinite number of states and is not a three state finite automaton. *Three state variables* might be a preferable phrasing.

[17] Before computer era, design offices and engineering research centers were equipped with rooms full of people spending the entire day performing computation for the engineers or researchers. At the time, gender parity in the field was not better than today and most of this people were women while most engineers were men and there were numerous cases of engineers marrying their computer.

[18] For instance, they allowed late versions of the Ariane launcher to put heavier satellites into orbit without requiring more fuel than previous versions.

[19] A good example of which is constituted by the centrifugal governor invented by the English engineer James Watt to regulate the speed of steam engines during the industrial revolution.

[20] Typical frequencies for flight commands are about 100 Hz.

**Critical Systems** such as flight commands may have disastrous results in case of failure. Two different types of failures must be distinguished: hardware and software failures. To prevent a hardware failure to bring disastrous effects, multiple redundant copies of devices more or less likely to fail are embedded. For instance, a common scheme, is to triplicate input sensors[21] and use a voter choosing the median of the three values so that a dysfunctional sensor has no impact. Diversification can also be introduced among the different copies in order to avoid a design failure to affect all the copies at the same time[22]. Such use of duplication and diversification to circumvent hardware failures is older than digital flight commands. For instance, hydraulic commands already used multiple hydraulic circuits, each feeding a complementary set of control surfaces and designers take care to make hydraulic pipes of different circuits traveling as far as possible of each other in the plane so as to limit the risk that all circuits could be damaged at the same time. All these duplication and diversification strategies and voting algorithms raise interesting verification questions. This is however not the subject of this thesis.

Software also constitute a potential source of failures. Therefore, they have to meet stringent certification requirements [RTC92]. This is currently enforced by means of extensive testing. This can be considered successful in the sense that after decades of daily operations of thousands of aircrafts no loss of life can be imputed to a software failure in a numerical flight command system. Such process are nonetheless terribly heavy and expensive[23]. Furthermore, the number of cases to test being way too large, exhaustive testing is intractable and there is no guarantee that the chosen test cases do not fail to expose a bug[24]. All this explains the interest of both the industrial and the academic communities in formal methods able to more or less automatically deliver mathematical proof of correctness. Among them, this thesis will particularly focus on abstract interpretation, an efficient method to automatically generate proofs of numerical properties which are essential in our context.

Two types of properties are of major interest:[25] *closed loop stability* and *open loop stability*. Closed loop stability, illustrated on Figure 1.1, asserts that the controller manages to stabilize the plant (assuming that its model accurately simulate the physical system).[26] Open loop stability is illustrated on Figure 1.2. This properties states that all variables of the controller are bounded (assuming that its input is bounded). This can at first looks like a more artificial property since a system can be closed loop stable without being open loop stable. Indeed, when the controller is not (open loop) stable, as long as the plant does not feed it with inputs that would make it diverge, the whole system remains (closed loop) stable. However, open loop stability is still a nice property. In particular,

[21] Such as the pitot tube measuring airplane velocity for instance.

[22] For instance on the A320, flight command computers are of two different types. Due to an air conditioning failure in the avionic bay where they are located, all computers of one type burned during a test flight. Fortunately, the other type still allowed the plane to fly safely.

[23] For instance, on the B777, verification accounts for 70% of software development costs (which represent themselves a third of total development costs of the airplane) [FBG+].

[24] According to Edsger W. Dijkstra: "Program testing can be a very effective way to show the presence of bugs, but it is hopelessly inadequate for showing their absence." [Dij72].

[25] At least when focusing on stability, other properties such as performance being outside the scope of this thesis.

[26] In the case of a plane, this would mean that its attitude remains between some bounds (no excessive vertical speed for instance), for the inverted pendulum that it remains close from the vertical position.

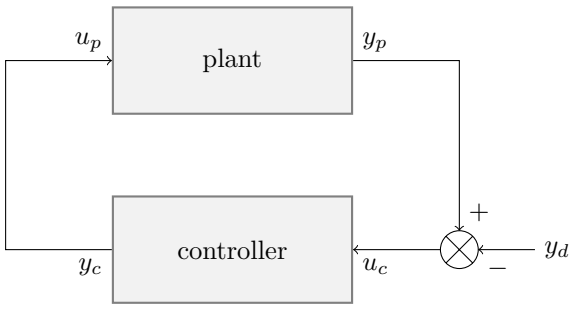



Figure 1.1 – Closed loop stability: the command $y_d$ being bounded, internal states $x_p$ and $x_c$ and outputs $y_p$ and $y_c$ of the plant and controller remain bounded.


Figure 1.2 – Open loop stability: the input $u_c$ of the controller being bounded, its internal state $x_c$ and output $y_c$ remain bounded.

an open loop stable controller will not dramatically diverge when experiencing a temporary failure of its input sensors (i.e., cutting the loop on Figure 1.1) then making its return to regular operation a lot less unlikely when sensors are back. That is why open loop stability is often required from critical systems. It is worth noting that the current notion of open loop stability slightly differs from the one common to control theorist. The latter include the plant which may be interesting for controller design but is completely useless when verifying an already designed controller. It is even a major asset of open loop property not to make any assumption on the plant. This document mainly deals with open loop stability, although closed loop stability would definitely constitute a very interesting extension.

## Contributions

The remaining of this first part of the document does not contain any contribution but rather introduces a few notions that will then be used throughout the remaining of the thesis. More precisely, the next chapter (Chapter 2) introduces the notion of inductive invariant and the need for invariant inference tools. A brief introduction to such an invariant inference technique, namely abstract interpretation, follows (Chapter 3). Finally, a last chapter (Chapter 4) presents a short state of the art of invariant inference techniques for open loop stability of linear control command systems.

The document is then organized in three parts, each corresponding to a distinct contribution. It is well known from control theorists that linear controllers are stable if and only if they admit a quadratic invariant (geometrically speaking, an ellipsoid). They call these invariants *quadratic Lyapunov functions* and a first part (Part II) offers to automatically compute such invariants for controllers given

as a pair of matrices $A'_c$ and $B'_c$. This is done using semi-definite programming optimization tools. It is worth noting that floating point aspects are taken care of, whether they affect computations performed by the analyzed program or by the tools used for the analysis.

However, the actual goal is to analyze programs implementing controllers (and not pairs of matrices), potentially including resets or saturations, hence not purely linears. The *policy iteration* technique is a recently developed static analysis techniques well suited to that purpose. However, it does not marry very easily with the classic abstract interpretation paradigm. The next part (Part III) tries to offer a nice interface between the two worlds.

Finally, the last part (Part IV) is a more prospective work on the use of polynomial global optimization based on Bernstein polynomials to compute polynomial invariants on polynomials systems.

# Chapter 2

# Inductive Invariants

This chapter introduces the notion of $k$-inductive invariant and the $k$-induction verification method and shows how it could benefit from an automatics invariant inferring method. The remaining of the document will then focus on such invariant generation in the case of numerical control command cores.

A very simple model of transition system is first introduced. Despite its simplicity, it allows to model the numerical control command systems of this thesis while enabling the discussion about invariants conducted in this chapter.

**Definition 16** (Transition System)**.** *Given a set $S$, a pair $(I, T)$ is called a* transition system *when $I \subseteq S$ and $T \subseteq S \times S$. The subset $I$ of $S$ is called set of* initial states *whereas the relation $T$ on $S$ is called* transition relation*.*

*In the following, we will denote $T(X)$ the set of states reachable by one transition $T$ from a state in $X$:*

$$T(X) := \{s' \in S \mid \exists\, s \in X, (s, s') \in T\}.$$

The following definition gives a semantic to transition systems.

**Definition 17** (Reachable State Space)**.** *A set $R \subseteq S$ is called* reachable state space *of a transition system $(I, T)$ on $S$ when it is the smallest set (for the order $\subseteq$) satisfying:*

$$\begin{cases} I \subseteq R \\ T(R) \subseteq R. \end{cases} \tag{2.1}$$

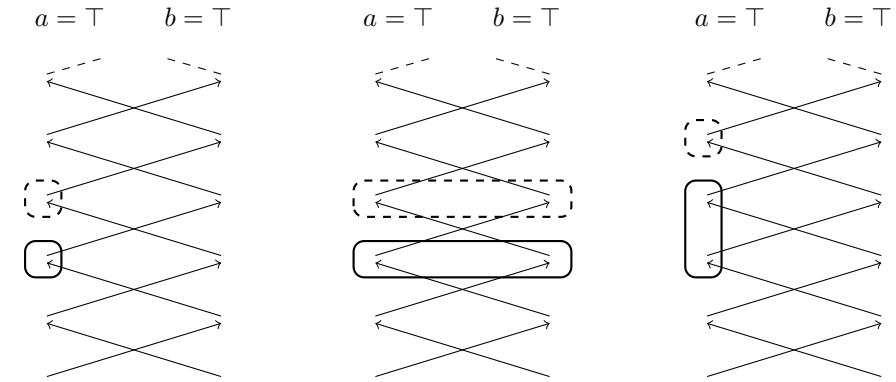
**Property 5.** *Any transition system has a unique reachable state space.*

*Proof.* We need to prove the existence and uniqueness of such an $R$. As any set of subsets, $(2^S, \subseteq)$ is a complete lattice. Moreover, the function $f : X \in 2^S \mapsto I \cup T(X)$ is monotone on this lattice. Then, by the Knaster-Tarski theorem [Kna28, Tar55], $R$ is the unique least fixpoint of function $f$. □

**Example 13.** *Given $S := \mathbb{Z}^2$:*

$$I := \{\,(0,0)\,\}$$
$$T := \left\{ ((x,y),(x',y')) \;\middle|\; \begin{array}{l} (x \leq 0 \wedge x' = x \wedge y' = y) \\ \vee\, (x \geq 1 \wedge x' = x \wedge y' = y + 1) \end{array} \right\}$$

*define a transition system. Its reachable state space is the singleton $R := \{\,(0,0)\,\}$.*




(a) $P := \{(a, b) \mid a = \top\}$ is not inductive.

(b) $P' := \{ (\top, \top) \}$ is inductive.

(c) $P$ is 2-inductive.

Figure 2.1 – Illustration of Example 15.

**Definition 18** (Invariant)**.** *$P \subseteq S$ is said to be an* invariant *property of a transition system when $R \subseteq P$, i.e., all reachable state satisfy the property $P$.*

**Definition 19** (Inductive Invariant)**.** *$P \subseteq S$ is said to be an* inductive invariant *of a transition system when both $I \subseteq P$ and $T(P) \subseteq P$.*

**Property 6.** *All inductive invariants are invariants.*

It is worth noting that the converse usually does not hold, as demonstrated in following example.

**Example 14.** *On the transition system of Example 13, $P := \{(x, y) \mid y = 0\}$ is an invariant. However it is not inductive since $T(P) = \{(x, y) \mid 0 \leq y \leq 1\}$. $P' := \{ (0, 0) \}$ is an inductive invariant.*

This illustrates a commonly occurring fact when trying to prove a mathematical result by induction. It is often needed to consider a stronger version of the result for the induction proof to work. A usual trick is to assume the property holds a few consecutive times (instead of just one) before proving it holds the next time.

**Definition 20** ($k$-inductive Invariant)**.** *$P \subseteq S$ is said to be $k$-*inductive *when:*

$$\bigcup_{0 \leq i \leq k-1} T^i(I) \subseteq P \quad \textit{and} \quad T^k\left( \bigcap_{0 \leq i \leq k-1} \left\{ s \in S \mid T^i(\{ s \}) \subseteq P \right\} \right) \subseteq P.$$

Basically, a property $P$ is $k$-inductive if up to $k-1$ transition from an initial state always lead to a state in $P$ and when an additional transition after $k-1$ transitions in $P$ always remains in $P$. It is interesting to notice that the particular case of 1-induction corresponds to the notion of induction as defined in Definition 19.

Example 14 demonstrated that not any invariant is inductive. There are even invariants which are not $k$-inductive for any $k$. However, there exist $k$-inductive invariants which are not $(k-1)$-inductive.

**Example 15** (Shoelaces)**.** *Given* $S := \mathbb{B}^2 = \{\bot, \top\}^2$*, we define* $(I, T)$ *with* $I := \{(\top, \top)\}$ *and* $T := \{((a, b), (a', b')) \mid (a = \bot \vee b' = \top) \wedge (b = \bot \vee a' = \top)\}$*. The reachable state space of this transition system is* $R := \{(\top, \top)\}$*.*

*Although* $P := \{(a, b) \mid a = \top\}$ *is an invariant of this transition system, it is not inductive. The stronger property* $P' := \{(a, b) \mid a = \top \wedge b = \top\}$ *is inductive but* $P$ *can also be proven* 2*-inductive. All this is illustrated on Figure 2.1 which justifies the name of* shoelaces *often given to this common example.*

This is used by the $k$-induction procedure to attempt to prove that a property $P$ is invariant on a transition system $(I, T)$.

**Definition 21** ($k$-induction Procedure)**.**

1. *Set* $k$ *to 1.*
2. *Test whether there exist* $s_0, \ldots, s_{k-1}$ *satisfying the following formula:*

$$s_0 \in I \wedge \left( \bigwedge_{0 \leq i \leq k-2} (s_i, s_{i+1}) \in T \right) \wedge \neg (s_{k-1} \in P)\,. \tag{2.2}$$

3. *If they exist, then return* `False` *(and* $s_0, \ldots, s_{k-1}$ *constitute a counter example).*
4. *Otherwise, test whether there exist* $s_0, \ldots, s_k$ *satisfying:*

$$\left( \bigwedge_{0 \leq i \leq k_1} s_i \in P \right) \wedge \left( \bigwedge_{0 \leq i \leq k_1} (s_i, s_{i+1}) \in T \right) \wedge \neg (s_k \in P)\,. \tag{2.3}$$

5. *If this formula is unsatisfiable, then return* `True` *(we have proved that* $P$ *is* $k$*-inductive).*
6. *Otherwise, increment* $k$ *and go back to 2.*

In practice, satisfiability of both formulas 2.2 and 2.3 is tested using SMT solvers [BSST09]. These formulas are often called respectively *base* and *step*. Thus, a *step counter example* is a model of 2.3.

This procedure does not terminate. However, in cases where it terminates it either proves that $P$ is an invariant or gives a counter example (at least assuming the underlying logic is decidable and the SMT solver always answers satisfiable (with a model of the formula) or unsatisfiable). In this last case, it is equivalent to a — pretty expensive — run of bounded model checking [Bie09]. Thus, this procedure is of actual interest only when $P$ is $k$-inductive (with $k$ small enough for formulas 2.2 and 2.3 to be proved unsatisfiable by a SMT solver).

**Remark 6** (About Invariants and Inductive Invariants)**.** *In the remaining of this document, the term "invariant" will commonly be used instead of "inductive invariant". In the rare occurrence where invariants that are not inductive are considered, this will be clearly stated.*

# Chapter 3

# Abstract Interpretation

Abstract interpretation is a convenient and versatile formal framework to define static analyses of programs [Cou99, CC77, CC79, CC92].

The goal of this chapter is certainly not to give a full overview of abstract interpretation but rather to make this document reasonably self content by introducing a toy language and notations for later use.

## 3.1 A Toy Imperative Language

Throughout this document, a very classic toy imperative language will be used to illustrate our analyses.

Programs in this language manipulate only real numbers. Moreover, to keep things as simple as possible in the following, the language contains as few constructs as possible. It could be extended with functions calls, variables of different types such as integer or boolean, pointers, elaborate data structures,... However, it is already Turing complete and these additional features would be of dubious interest to introduce the numerical analyses performed all along this document.

Finally, it is worth noting that control command software is ideally written in a synchronous language such as Lustre or Scade [BCE+03, HCRP91]. However, abstract interpretation on programs written in these languages can be efficiently performed by first applying a suitable compilation process leading to imperative programs [Jea00, Jea03]. Thus, this thesis will only deal with such imperative programs.

### 3.1.1 Syntax

A program $p$ of the language is a statement $stm$ in the following grammar:

$$\begin{aligned} stm &::= stm;\ stm \mid v := expr \mid v := ?(r, r) \\ &\quad \mid \textbf{if } expr \leq r \textbf{ then } stm \textbf{ else } stm \textbf{ fi} \\ &\quad \mid \textbf{while } expr \leq r \textbf{ do } stm \textbf{ od} \\ expr &::= v \mid r \mid expr + expr \mid expr - expr \mid expr \times expr \end{aligned}$$

with $v \in \mathbb{V}$, a set of variables, and $r \in \mathbb{R}$. $?(r_1, r_2)$ represents a random choice of a real number between $r_1$ and $r_2$ (useful to simulate inputs).

**Example 16.** *Figure 3.1 presents a program in this language.*

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

Figure 3.1 – Example of program.

**Remark 7.** *The program of Figure 3.1 is typical of control command programs: some initialization statements followed by an infinite loop. In this loop, some input variables (the variable in here) are obtained (typically from sensors measuring some value on the physical system to control) and an internal state (the variables xi here) is updated accordingly. This internal state could then be used to compute some output value (typically to send it to an actuator acting on the physical system). All this being repeated at some fixed period of time. Such programs are often called* reactive systems*.*

*All along this document, this kind of programs of the form*

$$s;\ \textbf{while}\ -1 \leq 0\ \textbf{do}\ \mathrm{s}'\ \textbf{od}$$

*where $s$ and $s'$ are statements and $s$ does not contain any while loop, will be very commonly considered.*

### 3.1.2 Collecting Semantics

**Definition 22** (Semantics of Expressions $[\![e]\!]$)**.** *The semantics $[\![e]\!](\rho) \in \mathbb{R}$ of an expression $e$ in an environment $\rho : \mathbb{V} \to \mathbb{R}$, is defined as:*

$$\begin{aligned}
&[\![v]\!](\rho) := \rho(v)\\
&[\![r]\!](\rho) := r\\
&[\![e_1 \diamond e_2]\!](\rho) := [\![e_1]\!](\rho) \diamond [\![e_2]\!](\rho) \qquad \text{for } \diamond \in \{+, -, \times\}\,.
\end{aligned}$$

**Definition 23** (Semantics of Statements $[\![s]\!]$)**.** *This allows to give a semantics $[\![s]\!](R) \subseteq (\mathbb{V} \to \mathbb{R})$ for a statement $s$ and a set of environments $R \subseteq (\mathbb{V} \to \mathbb{R})$:*

$$\begin{aligned}
&[\![s_1; s_2]\!](R) := [\![s_2]\!]([\![s_1]\!](R))\\
&[\![v := e]\!](R) := \{\rho[v \leftarrow [\![e]\!](\rho)] \mid \rho \in R\}\\
&[\![v := ?(\mathrm{r}_1, \mathrm{r}_2)]\!](R) := \{\rho[v \leftarrow r] \mid \rho \in R, r \in \mathbb{R}, r_1 \leq r \leq r_2\}\\
&[\![e \bowtie r]\!](R) := \{\rho \in R \mid [\![e]\!](\rho) \bowtie r\} \text{ for } \bowtie \in \{>, \geq, <, \leq\}\\
&[\![\textbf{if } \mathrm{e} \leq \mathrm{r} \textbf{ then } \mathrm{s}_1 \textbf{ else } \mathrm{s}_2 \textbf{ fi}]\!](R) := [\![s_1]\!]([\![e \leq r]\!](R)) \cup [\![s_2]\!]([\![e > r]\!](R))\\
&[\![\textbf{while } \mathrm{e} \leq \mathrm{r} \textbf{ do } \mathrm{s} \textbf{ od}]\!](R) := [\![e > r]\!](\mu\,(X \mapsto R \cup [\![s]\!]([\![e \leq r]\!](X))))\,.
\end{aligned}$$

*with $\mu\, f$ the least fixpoint of function $f$.*

*Proof.* For $[\![s]\!]$ to be well defined, the existence of the least fixpoint has to be proven. This can be done by proving $[\![s]\!]$ monotone, i.e., $[\![s]\!]$ is well defined and for all $R, R' \subseteq (\mathbb{V} \to \mathbb{R})$, if $R \subseteq R'$ then $[\![s]\!](R) \subseteq [\![s]\!](R')$. The proof being carried by structural induction on statements $s$, the only non trivial case is the last while loop case.

Thus, considering an expression $e$, a real value $r$ and a statement $s$ and assuming that $[\![s]\!]$ is monotone, we have to prove that $[\![\textbf{while e} \leq \text{r } \textbf{do s od}]\!]$ is well defined on $2^{(\mathbb{V}\to\mathbb{R})}$ and that for all $R, R' \subseteq (\mathbb{V} \to \mathbb{R})$ such that $R \subseteq R'$, the following holds:

$$[\![\textbf{while e} \leq \text{r } \textbf{do s od}]\!](R) \subseteq [\![\textbf{while e} \leq \text{r } \textbf{do s od}]\!](R').$$

Let us first notice that $(2^{(\mathbb{V}\to\mathbb{R})}, \subseteq)$ is a complete lattice (since for any set $S$, the set of its subsets $2^S$ equipped with the order $\subseteq$ is a complete lattice). Moreover, by induction hypotheses, the function $f_R : X \mapsto R \cup [\![s]\!]([\![e \leq r]\!](X))$ is monotone on this lattice. Then, according to the Knaster-Tarski theorem [Kna28, Tar55], the least fixpoint $\mu\, f_R$ is uniquely defined as

$$\mu\, f_R = \bigcap \left\{ X \in 2^{(\mathbb{V}\to\mathbb{R})} \;\middle|\; f_R(X) \subseteq X \right\}$$

and the same holds for $R'$. Since $R \subseteq R'$, by definition of $f_R$ and $f_{R'}$, for all $X$, $f_R(X) \subseteq f_{R'}(X)$ which implies that $\{X \mid f_{R'}(X) \subseteq X\} \subseteq \{X \mid f_R(X) \subseteq X\}$. Thus $\mu\, f_R \subseteq \mu\, f_{R'}$, hence finally the result. □

**Definition 24** (Semantics of Programs $[\![p]\!]$)**.** *Finally the semantics of a program $p$ is given by the semantics $[\![p]\!](\mathbb{V} \to \mathbb{R})$ of $p$ starting from an unknown state.*

This denotational semantics is very classic and can be found in textbooks about program semantics [Win93][1].

It is worth noting that this semantics is given with operations over real numbers $\mathbb{R}$ whereas an actual program would compute using floating point values. This point will be briefly discussed later but is currently mostly left as future work.

**Remark 8.** *The common case of core controller*

$$s;\ \textbf{while } -1 \leq 0 \textbf{ do } s' \textbf{ od}$$

*where $s$ is a statement without loops can be seen as a system of transition, as defined in Chapter 2, with*

$$I := [\![s]\!](\mathbb{V} \to \mathbb{R}) \quad \textit{and} \quad T := \left\{ (X, [\![s']\!](X)) \;\middle|\; X \in 2^{(\mathbb{V}\to\mathbb{R})} \right\}.$$

*Indeed, its reachable state space $R$ is given by*

$$R := \mu\, (X \mapsto [\![s]\!](\mathbb{V} \to \mathbb{R}) \cup [\![s']\!](X)).$$

*This gives a meaning to the notions of invariants and inductive invariants on this kind of programs. Thus, the set $R$, also called set of* reachable states *at loop head will be considered instead of the semantics of the full program $[\![s;\ \textbf{while } -1 \leq 0 \textbf{ do } s' \textbf{ od}]\!]$ which is just $\emptyset$ since the program never terminates.*

[1] It should be noted however that not all authors agree on the definition of pre and postfixpoints of a function $f$. Some [Win93] define a prefixpoint as a point $x$ such that $f(x) \leq x$ whereas in the abstract interpretation community [Cou99, CC77, CC79, CC92], it is usually defined as a point $x$ such that $x \leq f(x)$ (and dually for postfixpoints). In the remaining of this document, the second convention is adopted.

## 3.2 Abstract Domains

The previous concrete semantics being non computable[2], the basic idea of abstract interpretation is to compute an abstract semantics. This abstract semantics is designed as a computable overapproximation of the concrete one.

*Abstract domains* are the basic bricks of abstract interpretation. They are based on a *complete lattice* $\left(\mathcal{D}^\sharp, \sqsubseteq^\sharp_{\mathcal{D}}\right)$ (the exponent $.^\sharp$ is commonly used to distinguish abstract objects from their concrete counterpart (here the complete lattice $(\mathcal{D}, \sqsubseteq_{\mathcal{D}}) := \left(2^{(\mathbb{V}\to\mathbb{R})}, \subseteq\right)$)). The least upper bound (also called join) of the lattice, its greatest lower bound (also called meet), its least and greatest elements are respectively denoted $\sqcup^\sharp_{\mathcal{D}}$, $\sqcap^\sharp_{\mathcal{D}}$, $\bot_{\mathcal{D}}$ and $\top_{\mathcal{D}}$. Binary versions of the least upper bound and greatest lower bound will commonly be used (i.e., $\sqcup^\sharp_{\mathcal{D}} \{ x^\sharp, y^\sharp \}$ will be denoted $x^\sharp \sqcup^\sharp_{\mathcal{D}} y^\sharp$). To lighten notations, the subscript $_{\mathcal{D}}$ will often be omitted when it can easily be inferred from the context. Each element in $\mathcal{D}^\sharp$ called an *abstract value* will "abstract" some concrete element in $\mathcal{D}$. What "abstract" means is formally defined by a function from $\mathcal{D}^\sharp$ to $\mathcal{D}$ called a *concretization function*.

**Definition 25** (Concretization Function)**.** *A* concretization function *is a function $\gamma_{\mathcal{D}} : \mathcal{D}^\sharp \to \mathcal{D}$ from the abstract domain $\mathcal{D}^\sharp$ to the concrete $\mathcal{D}$. This function must be monotone:*

$$\forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp,\ x^\sharp \sqsubseteq^\sharp_{\mathcal{D}} y^\sharp \Rightarrow \gamma_{\mathcal{D}}(x^\sharp) \sqsubseteq_{\mathcal{D}} \gamma_{\mathcal{D}}(y^\sharp).$$

Thus, in our case, an abstract value $x^\sharp$ in $\mathcal{D}^\sharp$ is associated to a concrete value $\gamma_{\mathcal{D}}(x^\sharp) \subseteq (\mathbb{V} \to \mathbb{R})$, i.e., a set of environments $\rho : \mathbb{V} \to \mathbb{R}$ associating a value in $\mathbb{R}$ to each variable in $\mathbb{V}$. The monotonicity constraint allows to give a concrete meaning to the abstract order $\sqsubseteq^\sharp_{\mathcal{D}}$: when an abstract value $x^\sharp$ is smaller than another $y^\sharp$ in $\mathcal{D}^\sharp$ (i.e., $x^\sharp \sqsubseteq^\sharp_{\mathcal{D}} y^\sharp$), then $x^\sharp$ represents less environments than $y^\sharp$ (i.e., $\gamma_{\mathcal{D}}(x^\sharp) \subseteq \gamma_{\mathcal{D}}(y^\sharp)$). In other words, $x^\sharp \sqsubseteq^\sharp_{\mathcal{D}} y^\sharp$ means that the abstraction $x^\sharp$ is *more precise* than $y^\sharp$.

**Example 17** (Intervals Domain)**.** *A very common abstract domain is the* intervals domain *defined as: $\mathcal{I}^\sharp := \mathbb{V} \to I$ with $I := \bot_I \cup \left\{[a, b] \;\middle|\; a \in \overline{\mathbb{R}}, b \in \overline{\mathbb{R}}, a \leq b\right\}$ where $\overline{\mathbb{R}} := \mathbb{R} \cup \{ -\infty, +\infty \}$ and $\leq$ is the usual order on $\overline{\mathbb{R}}$. An order $\sqsubseteq^\sharp_{\mathcal{I}}$ is defined on $\mathcal{I}^\sharp$ by $x^\sharp \sqsubseteq^\sharp_{\mathcal{I}} y^\sharp$ when for all $v \in \mathbb{V}$:*

- *either $x^\sharp(v) = \bot_I$;*
- *or $x^\sharp(v) = [a, b]$ and $y^\sharp(v) = [c, d]$ and $c \leq a$ and $b \leq d$.*

*Equipped with this order, $\mathcal{I}^\sharp$ is a complete lattice. Giving a subset $S$ of $I$, $\sqcup^\sharp_I S$ is defined as:*

$$\sqcup^\sharp_I S := \begin{cases} \bot_I & \text{when } S = \emptyset \text{ or } S = \{ \bot_I \} \\ \left[\inf \left\{a \in \overline{\mathbb{R}} \;\middle|\; [a, b] \in S\right\}, \right. \\ \left. \sup \left\{b \in \overline{\mathbb{R}} \;\middle|\; [a, b] \in S\right\}\right] & \text{otherwise} \end{cases}$$

*which allows to define the least upper bound $\sqcup^\sharp_{\mathcal{I}}$ of $\mathcal{I}^\sharp$:*

$$\sqcup^\sharp_{\mathcal{I}} S := \left(v \mapsto \sqcup^\sharp_I \left\{x^\sharp(v) \;\middle|\; x^\sharp \in S\right\}\right).$$

[2] Otherwise, the halting problem (on our Turing complete programming language) would be solved by just checking whether the semantics of programs is $\emptyset$ or not.

*The infimums $\bot_{\mathcal{I}}$ and $\top_{\mathcal{I}}$ are the functions mapping every variable respectively to $\bot_I$ and $[-\infty, +\infty]$.*

*The concretization function $\gamma_{\mathcal{I}} : \mathcal{I}^\sharp \to 2^{(\mathbb{V}\to\mathbb{R})}$ is given by:*

$$\gamma_{\mathcal{I}}(x^\sharp) := \left\{\rho : \mathbb{V} \to \mathbb{R} \;\middle|\; \forall v \in \mathbb{V}, \exists a, b \in \overline{\mathbb{R}}, x^\sharp(v) = [a, b] \wedge a \leq \rho(v) \leq b\right\}.$$

*It is worth noting that $\gamma_{\mathcal{I}}(x^\sharp) = \emptyset$ when $x^\sharp(v) = \bot_I$ for some $v \in \mathbb{V}$.*

## 3.3 Abstract Operators

The previous section just defined an abstraction. The goal being to use this abstraction to compute an overapproximation of the concrete semantics (which is not computable), this requires operations on this abstraction. *Abstract operators* are functions on an abstract domain which mimic actual operations such as assignments and guards in order to enable the computation in the abstract domain of an overapproximation of the concrete semantics defined in Section 3.1.2.

**Definition 26** (Abstract operators)**.** *Given a variable $v \in \mathbb{V}$ and an expression $e$, an abstract operator for the assignment $v{:=}e$ is a function $[\![v{:=}e]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$. Similarly, abstract operators for random assignments $[\![v{:=}?(r_1, r_2)]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ and for guards $[\![e \leq r]\!]^\sharp : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ are defined.*

*These operators are called* sound *with respect to their concrete counterpart when they fulfill the following condition:*

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![v{:=}e]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![v{:=}e]\!]^\sharp(x^\sharp)\right).$$

*This condition is called* soundness condition*. Similar conditions are defined for random assignments and guards.*

Intuitively, the soundness condition expresses the fact that the abstract operator does not forget any behavior of its concrete counterpart. More precisely, if an environment $\rho$ can be obtained through the assignment $v{:=}e$ starting from an environment represented by $x^\sharp$ (i.e., in $\gamma_{\mathcal{D}}(x^\sharp)$), that is $\rho \in [\![v{:=}e]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right)$, then this environment is required to be represented by the result of the abstract operator $[\![v{:=}e]\!]^\sharp$ on $x^\sharp$, that is $\rho \in \gamma_{\mathcal{D}}\left([\![v{:=}e]\!]^\sharp(x^\sharp)\right)$.

Given these abstract operators, an abstract semantics of statements $[\![s]\!]^\sharp$ can be defined which mostly corresponds to the concrete semantics of Section 3.1.2 in which abstract operators replace their concrete counterparts:

$$\begin{aligned}
&[\![s_1; s_2]\!]^\sharp(x^\sharp) := [\![s_2]\!]^\sharp\left([\![s_1]\!]^\sharp(x^\sharp)\right)\\
&[\![\textbf{if } \mathrm{e} \leq \mathrm{r} \textbf{ then } \mathrm{s}_1 \textbf{ else } \mathrm{s}_2 \textbf{ fi}]\!]^\sharp(x^\sharp) :=\\
&\qquad [\![s_1]\!]^\sharp\left([\![e \leq r]\!]^\sharp(x^\sharp)\right) \sqcup^\sharp [\![s_2]\!]^\sharp\left([\![-e \leq -r]\!]^\sharp(x^\sharp)\right) \qquad (3.1)\\
&[\![\textbf{while } \mathrm{e} \leq \mathrm{r} \textbf{ do } \mathrm{s} \textbf{ od}]\!]^\sharp(x^\sharp) :=\\
&\qquad [\![-e \leq -r]\!]^\sharp\left(\mu\left(X \mapsto x^\sharp \sqcup^\sharp [\![s]\!]^\sharp\left([\![e \leq r]\!]^\sharp(X)\right)\right)\right).
\end{aligned}$$

Provided the abstract operators are sound, this semantics is then sound with respect to the concrete one:

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![s]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![s]\!]^\sharp(x^\sharp)\right).$$

The abstract semantics of a program $p$ is finally $[\![p]\!]^\sharp(\top)$.

When the abstract operators, as well as the least fixpoint $\mu$, can be computed efficiently, this abstract semantics gives a practical algorithm to actually compute an overapproximation of the concrete semantics. If the abstract semantics is precise enough so that it remains included in some property $P$, it then gives a method to prove that this property holds on the program. Of course, the method is not complete[3] and the converse is not true, the fact that the abstract semantics is not included in $P$ can either means that property $P$ is false or that the abstract semantics is a too coarse overapproximation of the concrete one. However, practically computing the least fixpoint $\mu$ is far from trivial and will be the subject of next section.

**Example 18** (Abstract Operators for the Intervals Domain)**.** *Abstract operations* $+^\sharp$, $-^\sharp$ *and* $\times^\sharp : I \times I \to I$ *can be defined for arithmetic operations:*

$$+^\sharp : (x,y) \mapsto \begin{cases} \bot_I & \text{when } x = \bot_I \text{ or } y = \bot_I \\ [a+c, b+d] & \text{when } x = [a,b] \text{ and } y = [c,d] \end{cases}$$

$$-^\sharp : (x,y) \mapsto \begin{cases} \bot_I & \text{when } x = \bot_I \text{ or } y = \bot_I \\ [a-d, b-c] & \text{when } x = [a,b] \text{ and } y = [c,d] \end{cases}$$

$$\times^\sharp : (x,y) \mapsto \begin{cases} \bot_I & \text{when } x = \bot_I \text{ or } y = \bot_I \\ [\min(ab,ac,ad,bd), \\ \ max(ab,ac,ad,bd)] & \text{when } x = [a,b] \text{ and } y = [c,d]. \end{cases}$$

*They allow to give an abstract semantics for expressions:*

$$\begin{array}{l} [\![v]\!]^\sharp(x^\sharp) := x^\sharp(v) \\ [\![r]\!]^\sharp(x^\sharp) := [r,r] \\ [\![e_1 \diamond e_2]\!]^\sharp(x^\sharp) := \diamond^\sharp \left( [\![e_1]\!]^\sharp(x^\sharp), [\![e_2]\!]^\sharp(x^\sharp) \right) \qquad \text{for } \diamond \in \{+,-,\times\} \end{array}$$

*which eventually enable to define abstract operators for the intervals domain:*

$$[\![v{:=}e]\!]^\sharp(x^\sharp) := \begin{cases} \bot_{\mathcal{I}} & \text{when } x^\sharp = \bot_{\mathcal{I}} \\ x^\sharp \left[ v \mapsto [\![e]\!]^\sharp(x^\sharp) \right] & \text{otherwise} \end{cases}$$

$$[\![v{:=}?(r_1,r_2)]\!]^\sharp(x^\sharp) := \begin{cases} \bot_{\mathcal{I}} & \text{when } x^\sharp = \bot_{\mathcal{I}} \text{ or } r_1 > r_2 \\ x^\sharp \left[ v \mapsto [r_1,r_2] \right] & \text{otherwise} \end{cases}$$

$$[\![e \leq r]\!]^\sharp(x^\sharp) := x^\sharp.$$

*To prove these abstract operators sound with respect to their concrete counterparts, arithmetic operators are first proven sound (i.e., defining* $\gamma_I : I \to 2^{\mathbb{R}}$ *as the function mapping* $\bot$ *to* $\emptyset$ *and* $[a,b]$ *to* $\{x \in \mathbb{R} \mid a \leq x \leq b\}$*, the following property holds:* $\forall x^\sharp, y^\sharp \in I, \left\{ x \diamond y \;\middle|\; x \in \gamma_I\left(x^\sharp\right), y \in \gamma_I\left(y^\sharp\right) \right\} \subseteq \gamma_I\left(\diamond^\sharp\left(x^\sharp, y^\sharp\right)\right)$*). Then soundness of the abstract semantics of expressions can be established by structural induction on expressions (i.e., with the same* $\gamma_I$*:* $\forall x^\sharp \in \mathcal{I}^\sharp, [\![e]\!]\left(\gamma_{\mathcal{I}}(x^\sharp)\right) \subseteq \gamma_I\left([\![e]\!]^\sharp(x^\sharp)\right)$*) which eventually allows to prove the soundness of the abstract operator for assignments. Soundness of abstract operators for random assignments and guards can be obtained more directly.*

*It is worth noting that, although sound, the abstract operator for guards is not very precise (for instance,* $[\![x \leq 0]\!]^\sharp([-\infty, +\infty])$ *could be defined as* $[-\infty, 0]$

[3] As already stated, this would solve the halting problem.

*instead of* $[-\infty, +\infty]$*). A more precise abstract guard is usually defined using a backward semantics of expressions and iterative reductions [Cou99, Gra92].*

## 3.4 Kleene Iterations and Widening

Given computable abstract operators, the only missing element to be able to actually compute an abstract semantics of programs is an effective way to compute least fixpoint operator $\mu$ appearing in definition of the semantics of while loops. In practice, exactly reaching the least fixpoint being too hard, only an overapproximation thereof is computed. This leads to a further overapproximated abstract semantics but does not break its soundness.

In the following of this section, the function $f_{x^\sharp} : \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is assumed to give the abstract semantics of a loop guard and body starting from some abstract value $x^\sharp$ (i.e., for a loop "**while** e $\leq$ r **do** s **od**", $f_{x^\sharp} := y^\sharp \mapsto x^\sharp \sqcup^\sharp [\![s]\!]^\sharp \left([\![e \leq r]\!]^\sharp(y^\sharp)\right)$). The goal is then to compute an overapproximation of $\mu\, f_{x^\sharp}$, i.e., a $y^\sharp \in \mathcal{D}^\sharp$ such that there exists a fixpoint $z^\sharp$ ($f_{x^\sharp}(z^\sharp) = z^\sharp$) smaller than $y^\sharp$ ($z^\sharp \sqsubseteq^\sharp_{\mathcal{D}} y^\sharp$).

When the lattice $\mathcal{D}^\sharp$ satisfies the ascending chain condition (i.e., any sequence $\left(x^\sharp_n\right)_{n\in\mathbb{N}}$ such that for all $i \in \mathbb{N}$, $x^\sharp_i \sqsubseteq^\sharp_{\mathcal{D}} x^\sharp_{i+1}$, is ultimately stationary), the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ is ultimately stationary (since $f^0_{x^\sharp}(\bot) = \bot \sqsubseteq^\sharp_{\mathcal{D}} f_{x^\sharp}(\bot)$ and by an immediate induction using $f_{x^\sharp}$ monotonicity, the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ is ascending, hence ultimately stationary). This means, there exists a $N \in \mathbb{N}$ such that $f_{x^\sharp}\left(f^N_{x^\sharp}(\bot)\right) = f^N_{x^\sharp}(\bot)$. $f^N_{x^\sharp}(\bot)$ is then a fixpoint of $f_{x^\sharp}$ and by definition an overapproximation of its least fixpoint: $\mu\, f_{x^\sharp} \sqsubseteq^\sharp_{\mathcal{D}} f^N_{x^\sharp}(\bot)$. Moreover, this value can be simply computed by iterating computations of $f_{x^\sharp}$ starting from $\bot$ until a fixpoint is reached.

Unfortunately, lots of interesting abstract domains do not satisfy the ascending chain condition in which case the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ may not converge in finitely many iterations. This is for instance the case of the intervals abstract domain given in Example 17 (the sequence $([0, n])_{n\in\mathbb{N}}$ is not ultimately stationary for instance). To address this issue, the convergence of the sequence $\left(f^n_{x^\sharp}(\bot)\right)_{n\in\mathbb{N}}$ will be "accelerated" by a *widening operator*.

**Definition 27** (Widening Operator)**.** *A function $\nabla_{\mathcal{D}} : \mathcal{D}^\sharp \times \mathcal{D}^\sharp \to \mathcal{D}^\sharp$ is a widening for the abstract domain $\mathcal{D}^\sharp$ when it satisfies the two following conditions:*

- *$\nabla_{\mathcal{D}}$ is an overapproximation of the least upper bound $\sqcup^\sharp_{\mathcal{D}}$:*

$$\forall x^\sharp, y^\sharp \in \mathcal{D}^\sharp, x^\sharp \sqcup^\sharp_{\mathcal{D}} y^\sharp \sqsubseteq^\sharp_{\mathcal{D}} x^\sharp \nabla_{\mathcal{D}} y^\sharp;$$

- *for any sequence $\left(x^\sharp_n\right)_{n\in\mathbb{N}}$, the sequence $\left(y^\sharp_n\right)_{n\in\mathbb{N}}$ defined as*

$$\begin{cases} y^\sharp_0 := x^\sharp_0 \\ y^\sharp_{n+1} := y^\sharp_n \nabla_{\mathcal{D}} x^\sharp_{n+1} \end{cases}$$

*is ultimately stationary.*

**Example 19** (Widening for the Intervals Domain)**.** *Assuming* $\nabla_I : I \times I \to I$ *defined as:*

$$x \,\nabla_I\, y = \begin{cases} y & \text{when } x = \bot_I \\ x & \text{when } y \sqsubseteq^\sharp_{\mathcal{I}} x \\ [a, +\infty] & \text{otherwise, when } x = [a, b],\ y = [c, d] \text{ and } a \leq c \\ [-\infty, b] & \text{otherwise, when } x = [a, b],\ y = [c, d] \text{ and } d \leq b \\ [-\infty, +\infty] & \text{otherwise} \end{cases}$$

*the following operator is a widening for the intervals domain (the set of variables* $\mathbb{V}$ *being finite):*

$$x^\sharp \,\nabla_{\mathcal{I}}\, y^\sharp = \left(v \mapsto x^\sharp(v) \,\nabla_I\, y^\sharp(v)\right).$$

Using a widening, the sequence $\left(y^\sharp_n\right)_{n\in\mathbb{N}}$ defined as $y^\sharp_0 := \bot$ and $y^\sharp_{n+1} := y^\sharp_n \,\nabla_{\mathcal{D}}\, f_{x^\sharp}(y^\sharp_n)$ is ultimately stationary which enables to compute an overapproximation of the abstract semantics (3.1).

**Definition 28** (Abstract Semantics)**.** *Abstract semantics of statements is defined by structural induction on statements:*

$$\begin{array}{l} [\![s_1; s_2]\!]^\sharp(x^\sharp) := [\![s_2]\!]^\sharp\left([\![s_1]\!]^\sharp(x^\sharp)\right) \\ [\![\textbf{if e} \leq \text{r } \textbf{then } \text{s}_1 \textbf{ else } \text{s}_2 \textbf{ fi}]\!]^\sharp(x^\sharp) := \\ \qquad\qquad [\![s_1]\!]^\sharp\left([\![e \leq r]\!]^\sharp(x^\sharp)\right) \sqcup^\sharp [\![s_2]\!]^\sharp\left([\![-e \leq -r]\!]^\sharp(x^\sharp)\right) \\ [\![\textbf{while e} \leq \text{r } \textbf{do s od}]\!]^\sharp(x^\sharp) := [\![-e \leq -r]\!]^\sharp(y^\sharp_\infty(x^\sharp)) \end{array} \tag{3.2}$$

*where* $y^\sharp_\infty(x^\sharp)$ *is the limit of the ultimately stationary sequence defined by* $y^\sharp_0(x^\sharp) := \bot$ *and* $y^\sharp_{n+1}(x^\sharp) := y^\sharp_n(x^\sharp) \nabla \left(x^\sharp \sqcup^\sharp [\![s]\!]^\sharp\left([\![e \leq r]\!]^\sharp(y^\sharp_n(x^\sharp))\right)\right)$.

**Property 7** (Soundness of the Abstract Semantics)**.** *Provided the abstract operators are sound, this semantics is then sound with respect to the concrete one:*

$$\forall x^\sharp \in \mathcal{D}^\sharp, [\![s]\!]\left(\gamma_{\mathcal{D}}(x^\sharp)\right) \subseteq \gamma_{\mathcal{D}}\left([\![s]\!]^\sharp(x^\sharp)\right).$$

*Proof.* By structural induction on statement $s$. □

**Remark 9** (Uselessness of $x^\sharp \sqcup^\sharp .$)**.** *Although nothing requires it in the definitions, it is very common that* $\bot \nabla x^\sharp = x^\sharp$ *and the abstract semantics satisfies* $[\![.]\!]^\sharp(\bot) = \bot$*. In this case, the first terms of the sequence* $y^\sharp(x^\sharp)$ *in Definition 28 are* $y^\sharp_0(x^\sharp) = \bot$ *and* $y^\sharp_1(x^\sharp) = x^\sharp$*.*

*Moreover, the widening is often such that for all* $x^\sharp$*,* $y^\sharp$ *and* $z^\sharp \in \mathcal{D}^\sharp$ *if* $z^\sharp \sqsubseteq^\sharp x^\sharp$ *then* $x^\sharp \nabla \left(z^\sharp \sqcup^\sharp y^\sharp\right) = x^\sharp \nabla y^\sharp$*. In this case, the definition of the following terms of the sequence amounts to* $y^\sharp_{n+1}(x^\sharp) = y^\sharp_n(x^\sharp) \nabla \left([\![s]\!]^\sharp\left([\![e \leq r]\!]^\sharp\left(y^\sharp_n(x^\sharp)\right)\right)\right)$*. This can be used to simplify the computation of abstract semantics of loops.*

**Example 20** (An Analysis with the Intervals Domain)**.** *This example will detail the analysis, i.e., the computation of the abstract semantics, of the following program with the intervals domain introduced in Examples 17, 18 and 19:*

```
x := 0; y := ?(0, 12);
while x ≤ 42 do
  x := x+1;
  y := y−x
od.
```

*The analysis starts from* $(a) := \top_{\mathcal{I}}$ *and first computes*

$$(b) := [\![x := 0]\!]^\sharp(a) = \{\, x \mapsto [0,0], y \mapsto [-\infty, +\infty] \,\}$$

*then*

$$(c) := [\![y :=?(0,12)]\!]^\sharp(b) = \{\, x \mapsto [0,0], y \mapsto [0,12] \,\}.$$

*Since the widening for intervals defined in Example 19 fulfills the requirements of Remark 9, the analysis now proceeds computing*

$$\begin{aligned}(d) &:= (c) \nabla \left([\![x := x+1; y := y-x]\!]^\sharp \left([\![x \leq 42]\!]^\sharp(c)\right)\right)\\ (e) &:= (d) \nabla \left([\![x := x+1; y := y-x]\!]^\sharp \left([\![x \leq 42]\!]^\sharp(d)\right)\right)\\ &\vdots\end{aligned}$$

*until a fixpoint is reached. Hence:*

$$\begin{aligned}(d) &= \{\, x \mapsto [0,0], y \mapsto [0,12] \,\} \nabla \{\, x \mapsto [1,1], y \mapsto [-1,11] \,\}\\ &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}\\ (e) &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\} \nabla \{\, x \mapsto [1,+\infty], y \mapsto [-\infty,11] \,\}\\ &= \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}.\end{aligned}$$

*A fixpoint is reached and the abstract semantics after the loop is eventually*

$$(f) := [\![-x \leq -42]\!]^\sharp(e) = \{\, x \mapsto [0,+\infty], y \mapsto [-\infty,12] \,\}.$$

$(e)$ *proves that* $x$ *always remains non negative at head of the loop and that* $y$ *always remains below* 12.

This chapter gave a brief overview of the abstract interpretation framework. New analyses can be defined in this framework in a nice way, by "just" defining a new abstract domain, that is by giving: (1) a complete lattice defining the domain, along with a concretization function giving a meaning to its abstract values; (2) abstract operators, simulating their concrete counterparts; (3) a widening operator, to enforce the termination of the analyses. Proving the monotonicity of the concretization function, the soundness of the abstract operators and the soundness and termination properties of the widening then guarantees a sound and terminating analysis. This will be the main focus of Parts III and IV of this document.

# Chapter 4

# State of the Art

This chapter intends to give a brief review of some static analysis methods currently available to bound the numerical core of control systems. Methods are first introduced along with their perceived strengths and weaknesses. Then some hints are given on how this work plans to address some of these issues.

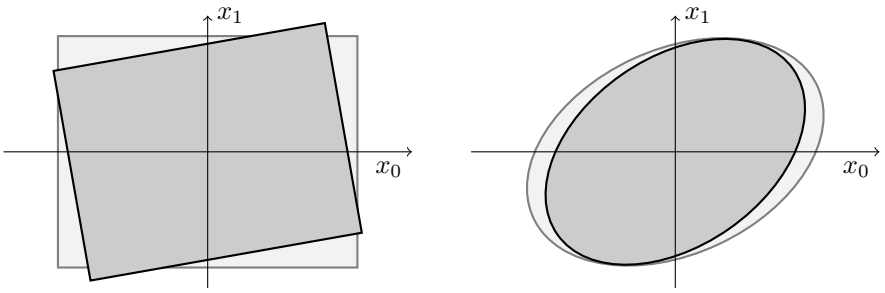
## 4.1 Linear Domains

Most control systems are based on a linear core. This is for instance the case of the Linear Quadratic Gaussian regulator given in Example 16, page 15. Unfortunately, these are hard to analyze using simple linear abstract domains, such as the intervals domain previously seen in Example 17, page 18 (for instance, the previous regulator does not admit any non trivial invariant in this domain). Figure 4.1a gives an intuition of this point. An interval property on two variables undergoes a small rotation composed with an homothety of factor $0.92 < 1$ (i.e., a strictly contracting linear transformation), showing that the property is not inductive.

Nevertheless, as control theorists know for long, stable linear systems admit quadratic invariants (called Lyapunov functions [BEEFB94, Lya47]). Such invariants can be depicted as ellipsoids. On Figure 4.1b, an ellipsoid is depicted along with its image by the same linear transformation as previously. This time, the property appears to be inductive.

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

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




(a) an intervals property: not inductive (b) a quadratic property: inductive

Figure 4.1 – Intervals vs quadratic properties for a linear transformation.

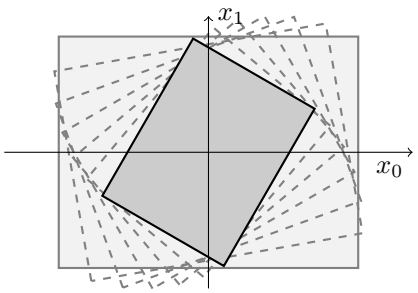



Figure 4.2 – The intervals property of Figure 4.1 is 6-inductive.

quadratic invariants have a space complexity quadratic in the number of variables and are intrinsically "smooth".

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

## 4.2 Unrolling

Unrolling constitutes a practical alternative to the search for linear inductive invariants with myriads of faces. For purely linear systems, unrolling to depths $k$ ranging from a few hundreds to a few thousands allows to compute precise $k$-inductive invariants while keeping the number of faces reasonably small [Fer04, GGP09], recent work [SB13] even demonstrates that precise bounds (i.e., the maximum reachable values) can be computed with simple support functions by fully unrolling the system. Intuitively, unrolling turns a contracting transformation into a more contracting one. Thus, properties which are not inductive may appear $k$-inductive. This is illustrated on Figure 4.2.

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

**Checking Results** When the user does not trust the analyzer and wants

to check its results a posteriori, not having an inductive invariant can dramatically complicate the task. In the extreme case where the system is fully unrolled, this even means that a posteriori checking amounts to replaying the whole analysis.

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

## 4.3 Quadratic Invariants

Although quadratic invariants are known for long, as quadratic *Lyapunov functions*, from control theorists [BEEFB94, Lya47], their use in static analysis dates back from less than a decade.

The first famous use of quadratic invariants for static analysis was two dimensional ellipsoids to bound second order filters [Fer04, Fer05, Mon05].

**Definition 29** (Filter of Order $n$)**.** *A filter of order $n$ (or $n$-th order filter) is a particular case of linear system in which a variable $x$ is assigned a linear combination of its $n$ previous values and an input $y$ along with its $n$ previous values. Written as a program, the loop has the following form:*

**while** $-1 \leq 0$ **do**
  y := **?**$(-1, 1)$;
  x := $a_1$x1+$\cdots$+$a_n$xn+$b_0$y+$b_1$y1+$\cdots$+$b_n$yn;
  xn := x(n−1);...; x2 := x1; x1 := x;
  yn := y(n−1);...; y2 := y1; y1 := y;
**od**.

**Property 8** (Bounding Second Order Filters)**.** *Assuming $a_1^2 + 4a_2 < 0$ and $|x - (a_1x_1 + a_2x_2)| \leq b$, if $x_1^2 - a_1x_1x_2 - a_2x_2^2 \leq r$ then $x^2 - a_1xx_1 - a_2x_1^2 \leq (\sqrt{-a_2r} + b)^2$. For a second order filter, when $a_1^2 + 4a_2 < 0$ and $\sqrt{-a_2} < 1$, this allows to infer the inductive invariant $x_1^2 - a_1x_1x_2 - a_2x_2^2 \leq r$ where $r$ satisfies $(\sqrt{-a_2r} + b)^2 \leq r$ and $b = |b_0| + |b_1| + |b_2|$.*

**Example 21.** *Considering the following second order filter*

x1 := 0; x2 := 0;
y1 := 0; y2 := 0;
**while** $-1 \leq 0$ **do**
  y := **?**$(-1, 1)$;
  x := 1.5x1−0.7x2+0.5y−0.7y1+0.4y2;
  x2 := x1; x1 := x;
  y2 := y1; y1 := y;
**od**,

*$a_1$ and $a_2$ satisfy $a_1^2 + 4a_2 = 1.5^2 - 4 * 0.7 < 0$ and $\sqrt{-a_2} = \sqrt{0.7} < 1$. Moreover, $|x - (a_1x_1 + a_2x_2)| \leq b$ with $b = |b_0| + |b_1| + |b_2| = 1.6$ since $y$, $y_1$ and $y_2$ all lie in the interval $[-1, 1]$. The following invariant is then inferred*

$$x_1^2 - 1.5x_1x_2 + 0.7x_2^2 \leq 95.96$$

*which means that* $|x| \leq 22.11$.

*It is worth noting that this bound is rather conservative since* $|x|$ *cannot exceed* 1.42. *Other quadratic invariants on* $x_1$ *and* $x_2$ *(i.e., ellipsoids of dimension* 2*) could yield tighter bounds (*$|x| \leq 16.14$*) but most of the conservatism comes from the fact that* $0.5y - 0.7y_1 + 0.4y_2$ *is simply abstracted as* $[-1.6, 1.6]$*, completely forgetting the fact that* $y_1$ *is the previous value of* $y$ *and* $y_2$ *the previous value of* $y_1$ *(indeed, once this abstraction is made, it becomes impossible to prove a better bound than* $|x| \leq 14.84$*).*

Bounds on filters of order $n$ can then be computed by decomposing them in filters of order 1 (bounded with intervals) and 2 (bounded as above) and refining the obtained bounds by means of some kind of unrolling [Fer04, Fer05, Mon05]. The method then presents the same advantages (precision of the computed bounds) and drawbacks (no inductive invariant is produced) as other unrolling methods.

Other work offer to compute quadratic inductive invariants of higher dimensions on larger classes of linear systems [AGG10, AFP09, GS10, RFM05]. Such invariants are computed thanks to the use of some numerical solvers, namely semi-definite programming solvers.

**Example 22.** *On the previous example, an ellipsoid of dimension* 4 *(on variables* $x_1$*,* $x_2$*,* $y_1$ *and* $y_2$*) constitutes an inductive invariant giving the bound* $|x| \leq 1.64$.

**Remark 10** (Exact Reachable State Space is not an Ellipsoid)**.** *Despite the ability of quadratic invariants to bound any stable linear system, it should be noted that the reachable state space of such systems is usually not an ellipsoid. Thus, although ellipsoids are good invariants, they will not always yield the tightest possible bounds.*

**Example 23** (Exact Reachable State Space is not an Ellipsoid)**.** *Consider the sequence defined by* $x_0 := 0$ *and* $x_{k+1} := Ax_k + Bu_k$ *where*

$$A := \begin{bmatrix} 0.92565 & -0.0935 \\ 0.00935 & 0.935 \end{bmatrix} \qquad B := \begin{bmatrix} 1 \\ 0 \end{bmatrix}$$

*and for all* $k \in \mathbb{N}$, $\|u_k\|_\infty \leq 1$. *The reachable state space of this system, depicted in Figure 4.3, is not an ellipsoid.*

*Proof.* Assume that the reachable state space $R$ is an ellipsoid. Pick some $x' \in R$ such that $\pi_1(x')$ is maximal (where $\pi_1(x')$ is the projection on the second component of $x'$). Then there exist $x \in R$ and $u \in \mathbb{R}$ such that $x' = Ax + Bu$. Assume (without loss of generality thanks to symmetry) $u \geq 0$, then $x'' := Ax + B(u-1) \in R$ and $\pi_1(x'') = \pi_1(x')$. Finally, by strict convexity of the ellipsoid $R$, there exists $\lambda \in \mathbb{R}$ such that $\lambda > 1$ and $\lambda\left(\frac{x'+x''}{2}\right) \in R$. But $\pi_1\left(\lambda\left(\frac{x'+x''}{2}\right)\right) = \lambda\pi_1(x') > \pi_1(x')$ which contradicts its maximality. □

## 4.4 Policy Iterations on Template Domains

This section advocates how nice ellipsoids are to bound linear systems. Yet, they suffer a significant disadvantage compared to classic abstract domains such

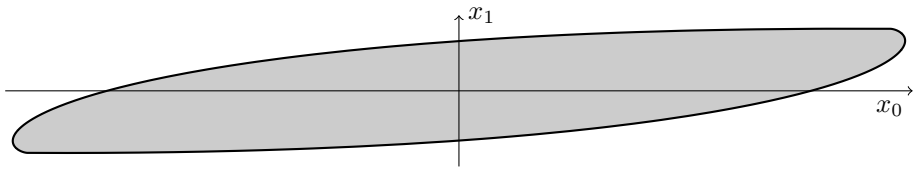



Figure 4.3 – Reachable state space for Example 23.

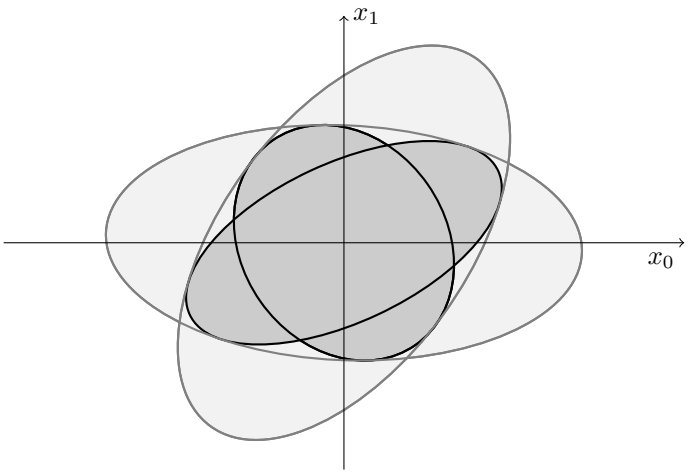



Figure 4.4 – There is usually no smallest ellipsoid containing two other given ellipsoids: the two light gray ellipsoids both contain the two darker ellipsoids but none of them is included in the other.

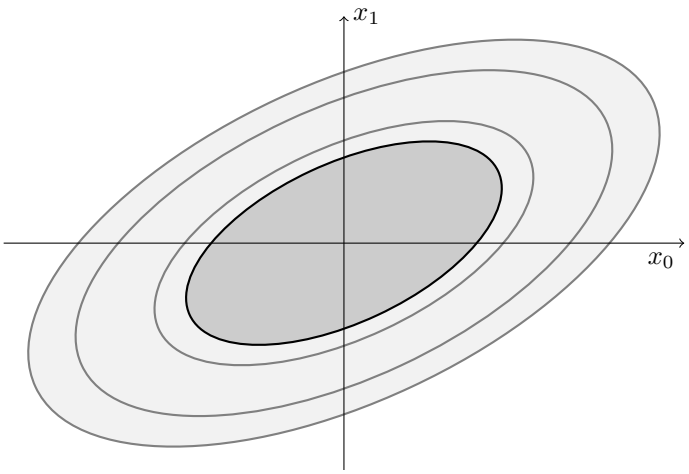



Figure 4.5 – Ellipsoids with the same shape and different radii.

as polyhedra, octagons, zonotopes,...: the set of ellipsoids equipped with the inclusion order $\subseteq$ is not a lattice. Indeed, there usually does not exist a smallest ellipsoid containing two other given ellipsoids (i.e., no least upper bound). This fact is illustrated on Figure 4.4 and prevents practical computations through Kleene iterations on the whole set of ellipsoids.

A common solution is to choose — prior to the analysis — an ellipsoid *shape* and only compute its *radius*, as illustrated on Figure 4.5. The set of radii $(\mathbb{R} \cap [0, +\infty)) \cup +\infty$ then constitutes a complete lattice. This idea is generalized through the notion of *template domains*.

**Definition 30** (Template Domains)**.** *Given a finite set* $\{ t_1, \ldots, t_n \}$ *of expressions on variables* $\mathbb{V}$*, the template domain* $\mathcal{T}$ *is defined as* $\overline{\mathbb{R}}^n = (\mathbb{R} \cup \{ -\infty, +\infty \})^n$ *and the meaning of an abstract value* $(b_1, \ldots, b_n) \in \mathcal{T}$ *is the set of environments* $\gamma_{\mathcal{T}}(b_1, \ldots, b_n) = \{\rho \in (\mathbb{V} \to \mathbb{R}) \mid [\![t_1]\!](\rho) \leq b_1, \ldots, [\![t_n]\!](\rho) \leq b_n\}$*. In other words, the abstract value* $(b_1, \ldots, b_n)$ *represents all the environments satisfying all the constraints* $t_i \leq b_i$*.*

Indeed, many common abstract domains are template domains. For instance the intervals domain of Example 17 is obtained with templates $x_i$ and $-x_i$ for all variables $x_i \in \mathbb{V}$ and the octagon domain [Min01] by adding all the $\pm x_i \pm x_j$.

Nonetheless, computation of precise invariants on numerical programs can be hard to achieve using classic Kleene iterations with widening. Policy iterations [AGG10, CGG+05, GS07a, GS10, GSA+12] is one of the alternatives to simple widening developed during the last decade [BSC12, FG10, GR06, HH12, SJ11, and references therein]. This technique allows to compute precise postfixpoints, usually by relying on mathematical optimization solvers such as linear or semi-definite programming tools. Such techniques have been recently developed for the computation of quadratic invariants for linear systems [AGG10, GS10, GSA+12].

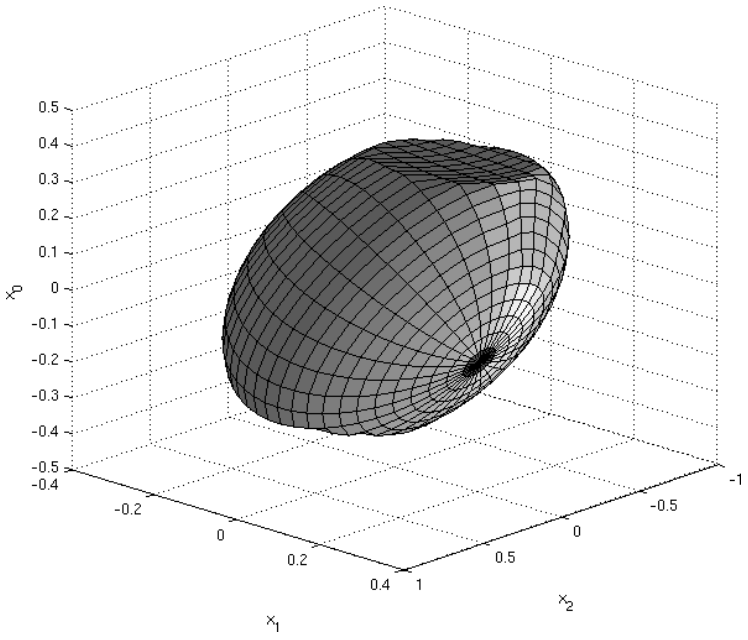



Figure 4.6 – Invariant for our running example.

The shape of the templates to be considered for policy iteration depends on the optimization tools used. For instance, linear programming [GGTZ07, GS07b] allows any linear templates whereas quadratic templates can be handled thanks to semi-definite programming and an appropriate relaxation [AGG10, GS10, GSA+12].

**Example 24.** *To bound the variables of the program of Figure 3.1, page 16, the quadratic template* $t_1 := 6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_0x_2 + 2.4182x_1x_2$ *is used. Templates* $t_2 := x_0^2$, $t_3 := x_1^2$ *and* $t_4 := x_2^2$ *are added in order to get bounds on each variable. Using these templates, policy iterations compute the invariant* $(1.0029, 0.1795, 0.1136, 0.2757) \in \mathcal{T}$, *meaning:* $t_1 \leq 1.0029 \wedge x_0^2 \leq 0.1795 \wedge x_1^2 \leq 0.1136 \wedge x_2^2 \leq 0.2757$ *or equivalently:* $t_1 \leq 1.0029 \wedge |x_0| \leq 0.4236 \wedge |x_1| \leq 0.3371 \wedge |x_2| \leq 0.5251$. *This is a cropped ellipsoid as displayed on Figure 4.6.*

An in depth introduction to policy iterations can be found in Chapter 9. Although very appealing, usage of these techniques in abstract interpretation based static analyzers is hampered by two main issues:

**Need for Templates** Suitable templates have to be provided by the user. Typically, the template $t_1$ in previous example is non trivial.

**Lack of Integration in the Abstract Interpretation Framework** The approach looks rather orthogonal to Kleene iterations with widening classically used in abstract interpretation tools [Jea10]. This prevents cooperation with other abstract domains through reduced products whereas this was demonstrated to be a key feature of abstract interpretation in one of it most brilliant successes: in the Astrée static analyzer, expensive relational domains (such as the octagons) are applied locally on a few variables of the analyzed program, communicating their results back and forth to less expensive non relational domains (such as the intervals domain) or other instances of relational domains [CCF+06].

This thesis mostly tries to address these issues by offering heuristics to automatically generate good templates before performing policy iterations and by integrating all this in a classic abstract domain. Since this abstract domain exhibits the same interface than other relational domains (such as the polyhedra or octagons domains for instance), its integration in an abstract interpretation based static analyzer is easier, as well as communication through reduced products with other domains already present in this analyzer.

# Part II

# Linear Systems

# Chapter 5

# Finding Good Ellipsoids

This chapter will present various heuristics to generate quadratic invariants, i.e., *ellipsoids* for pure linear systems. More precisely, systems considered in this part of this document start from some state $x_0$ and are updated according to the transition relation $x_{k+1} = Ax_k + Bu_k$ where $A$ is a matrix in $\mathbb{R}^{n\times n}$, $B$ is a matrix in $\mathbb{R}^{n\times p}$ $(n, p \in \mathbb{N}, n \neq 0)$ and $u$ is a bounded sequence: $\forall k \in \mathbb{N}, \|u_k\|_\infty \leq 1$. Matrices $A$ and $B$ are assumed to be given as input of the analyses conducted here, no actual code will be dealt with in the present part of this document. Analyzing actual program code will be the subject of Part III. This chapter is mostly an extended version of a conference paper [RJGF12].

Section 5.1 introduces Lyapunov theory as used throughout the remaining of the chapter. Section 5.2 then presents the overall approach before Section 5.3 offers various heuristics to find appropriate ellipsoidal invariants.

Chapter 6 will then deal with the floating point issues arising with the analyses presented in the current chapter before Chapter 7 details experimental results and Chapter 8 finally concludes.

## 5.1 Introduction to Lyapunov Stability Theory

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

$$V(0) = 0 \wedge \forall x \in \mathbb{R}^n \backslash \{0\}, V(x) > 0 \wedge \lim_{\|x\|\to\infty} V(x) = \infty \tag{5.1}$$

$$\forall x \in \mathbb{R}^n, V(f(x)) - V(x) \leq 0. \tag{5.2}$$

It is shown for example in [HC08] that exhibiting such a function proves the Lyapunov stability of the system, meaning that its state variables will remain bounded through time. Equation (5.2) expresses the fact that the function $k \mapsto V(x_k)$ decreases, which, combined with (5.1), shows that the state variables remain in the bounded sublevel-set $\{x \in \mathbb{R}^n | V(x) \leq V(x_0)\}$ at all instants $k \in \mathbb{N}$.

In the case of Linear Time Invariant systems (of the form $x_{k+1} = Ax_k$, with $A \in \mathbb{R}^{n\times n}$), one can always look for $V$ as a quadratic form in the state variables

of the system: $V(x) = x^T P x$ with $P \in \mathbb{R}^{n \times n}$ a symmetric matrix such that

$$P \succ 0 \tag{5.3}$$
$$A^T P A - P \preceq 0 \tag{5.4}$$

where "$P \succ 0$" means that the matrix $P$ is positive definite, i.e., for all non-zero vector $x, x^T P x > 0$.

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

$$x_{k+1} = Ax_k + Bu_k, \|u_k\|_\infty \leq 1. \tag{5.5}$$

The condition $\|u_k\|_\infty \leq 1$ reflects the fact that values coming from input sensors usually lies in a given range. The bound 1 is chosen without loss of generality since one can always alter the matrix $B$ to account for different bounds. To study this equation as precisely as possible, another model, expressing the behavior of the controlled system (the plant), is usually introduced. The two systems taken together form a closed system with no inputs which can be analyzed by looking for a $P$ matrix matching the criteria mentioned before. Such an analysis is referred to as 'closed loop stability analysis'. Here we seek not to model the plant, instead we only require for $\|u\|_\infty$ to remain bounded. Then, through a slight reinforcement of Equation (5.4) into

$$A^T P A - P \prec 0 \tag{5.6}$$

we can still guarantee that the state variables of (5.5) will remain in a sublevel set $\left\{x \in \mathbb{R}^n \;\middle|\; x^T P x \leq \lambda\right\}$ (for some $\lambda > 0$), which is an ellipsoid in this case. This approach only enables us to study control laws that are inherently stable, i.e., stable when taken separately from the plant they control. Nevertheless a wide range of controllers remains that can be analyzed. In addition, inherent stability is required in a context of critical applications.

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

## 5.2 Overall Method

### 5.2.1 Separate Shape and Ratio

We represent an ellipsoid by a pair $(P, \lambda)$ where $P \in \mathbb{R}^{n \times n}$ is a symmetric positive definite matrix giving the *shape* of the ellipsoid and $\lambda \in \mathbb{R}$ a scalar giving its *ratio*. The represented ellipsoid is then the set of all $x \in \mathbb{R}^n$ such that $x^T P x \leq \lambda$, i.e., the concretization function $\gamma$ is given by $\gamma : (P, \lambda) \mapsto \left\{x \in \mathbb{R}^n \;\middle|\; x^T P x \leq \lambda\right\}$. To avoid having multiple representations for the same ellipsoid[1] we can normalize $P$

[1] For instance $(P, 2\lambda)$ and $\left(\frac{P}{2}, \lambda\right)$ represent exactly the same ellipsoid.

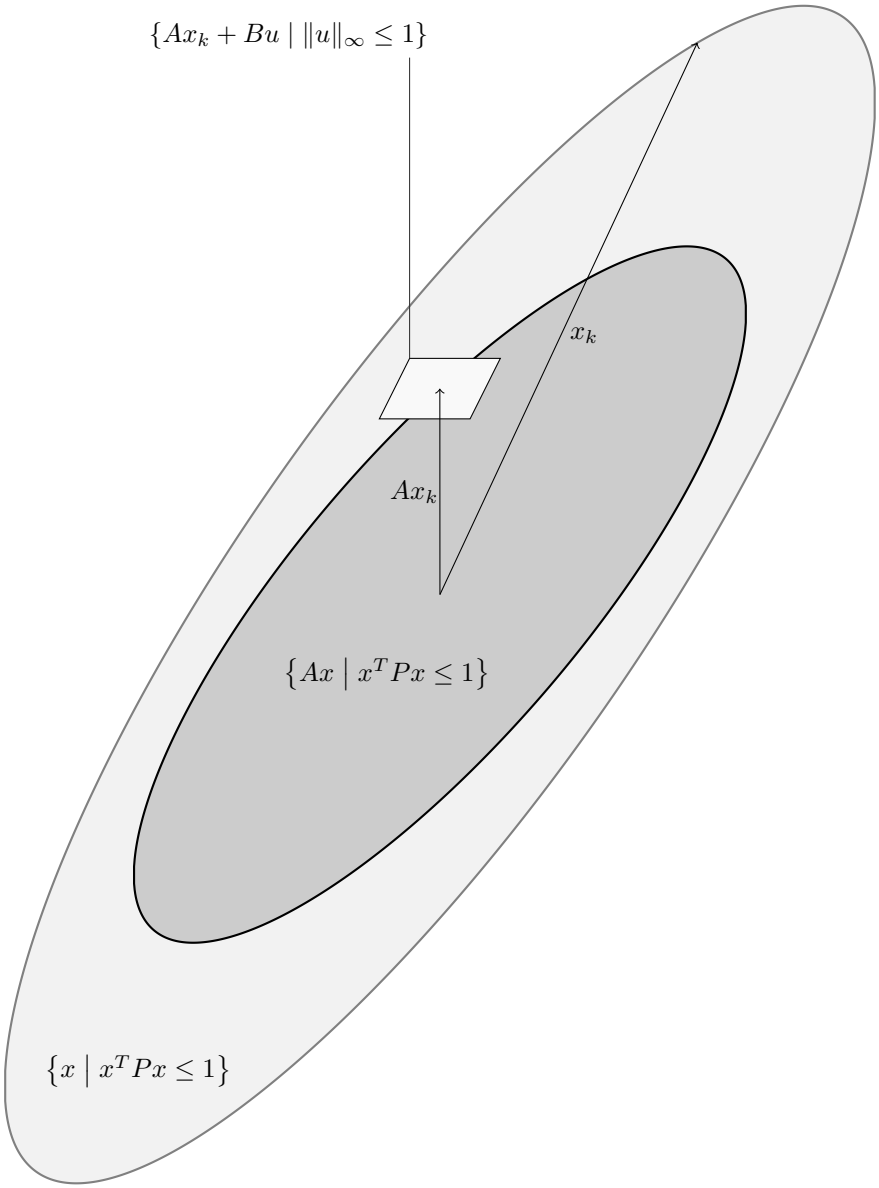



Figure 5.1 – Illustration of the stability concepts: if $x_k$ is in the light gray ellipse, then, after a time step, $Ax_k$ is in the dark gray ellipse, which is exactly what is expressed by Equation (5.4). The white box represents the potential values of $x_{k+1}$ after adding the effect of the bounded input $u_k$. We see here the necessity that the dark gray ellipse be strictly included in the light gray one, which is the stronger condition expressed by Equation (5.6).

for instance by requiring its largest coefficient to be 1. This is a particular case of template domain where the template is the degree two polynomial $x^T Px$.

This seemingly strange choice at first sight allows us to decompose the computation in two successive steps: (1) first determine the shape of the ellipsoid by choosing a well suited matrix $P$; (2) then find the smallest possible ratio $\lambda$ such that $x \in \gamma(P, \lambda)$ is an invariant. Various methods for both steps are detailed and compared in Sections 5.3 and 5.4.

### 5.2.2 Instrumentation: Use of Semi-definite Programming

To perform the aforementioned computations we heavily rely on semi-definite programming [BV04, VB96]. These tools allow to compute in polynomial time a solution to a linear matrix inequality (LMI) while minimizing a linear objective function[2]. A LMI is an inequality of the form

$$A_0 + \sum_{i=1}^{k} y_i A_i \succeq 0$$

where the $A_i$ are known matrices, the $y_i$ are the unknowns and "$P \succeq 0$" means that the matrix $P$ is positive semi-definite, i.e., $x^T Px \geq 0$ for all vector $x$. Indeed we can easily have unknown matrices since a matrix $A \in \mathbb{R}^{n\times n}$ can be expressed as $\sum_{i=0,j=0}^{n-1,n-1} A_{i,j} E^{i,j}$, where $E^{i,j}$ is the matrix with zeros everywhere except a one at line $i$ and column $j$. Likewise, multiple LMIs can be grouped into one since $A \succeq 0 \wedge B \succeq 0$ is equivalent to $\begin{bmatrix} A & 0 \\ 0 & B \end{bmatrix} \succeq 0$.

We will also have to deal with some implications which will be achieved by transforming them into a LMI by performing a relaxation thanks to the following theorem.

**Theorem 1** (S-Procedure)**.** *For any $P, P_1, \ldots, P_k \in \mathbb{R}^{n\times n}$ and $b, b_1, \ldots, b_k \in \mathbb{R}$ and $b, b' \in \mathbb{R}$, the following*

$$\exists \tau_1, \ldots, \tau_k \in \mathbb{R}, \left( \bigwedge_{i=1}^{k} \tau_i \geq 0 \right) \wedge \begin{bmatrix} -P & 0 \\ 0 & b \end{bmatrix} - \sum_{i=1}^{k} \tau_i \begin{bmatrix} -P_i & 0 \\ 0 & b_i \end{bmatrix} \succeq 0 \tag{5.7}$$

*is a sufficient condition for*

$$\forall x \in \mathbb{R}^n, \left( \bigwedge_{i=1}^{k} x^T P_i\, x \leq b_i \right) \Rightarrow x^T P\, x \leq b. \tag{5.8}$$

*Proof.* Assuming (5.7) holds, we have to prove (5.8). For any $x \in \mathbb{R}^n$, assuming the hypotheses of (5.8)

$$\bigwedge_{i=1}^{k} x^T P_i\, x \leq b_i$$

means that the sum

$$\begin{bmatrix} x \\ 1 \end{bmatrix}^T \left( \sum_{i=1}^{k} \tau_i \begin{bmatrix} -P_i & 0 \\ 0 & b_i \end{bmatrix} \right) \begin{bmatrix} x \\ 1 \end{bmatrix}$$

[2] Maximization of linear objectives is also possible by just minimizing the opposite function.

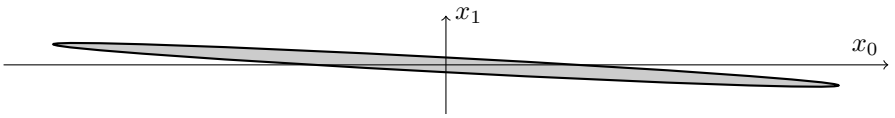



Figure 5.2 – 'flat' ellipsoids can yield very large bounds on some variables.

is non negative (since the $\tau_i$ are non negative). Then, according to (5.7)

$$\begin{bmatrix} x \\ 1 \end{bmatrix}^T \begin{bmatrix} -P & 0 \\ 0 & b \end{bmatrix} \begin{bmatrix} x \\ 1 \end{bmatrix}$$

is non negative, i.e., $x^T P\, x \leq b$. □

## 5.3 Shape of the Ellipsoid

As was presented in Section 5.1, any positive definite matrix $P$ satisfying the Lyapunov equation

$$A^T P\, A - P \prec 0 \tag{5.9}$$

will yield a proof of stability and provide *some* bound on the variables. However, additional constraints on $P$ can be introduced that make it possible to obtain better results than others.

While in Control Theory the existence of such ellipsoids is sufficient to prove stability of the system, we are here interested in characterizing it concretely. Investigating heuristically multiple possible shapes allows us to find one which is more adequate, i.e., more precise, with respect to the analyzed system.

The following subsections describe three different types of additional constraints on $P$ and their respective advantages.

### 5.3.1 Minimizing Condition Number

Graphically, the condition number of a positive definite matrix expresses a notion similar to that addressed by excentricity for ellipses in dimension 2. It measures how 'close' to a circle (or its higher dimension equivalent) the resulting ellipsoid will be. Multiples of the identity matrix, which all represent a circle, have a condition number of 1. Thus one idea of constraint we can impose on $P$ is to have its condition number as close to 1 as possible. A rationale for this is that 'flat' ellipsoids, i.e., having a large condition number, can yield a very bad bound on one of the variables, as illustrated on Figure 5.2.

This is done [BEEFB94] by minimizing a new variable, $r$, in the following matrix inequality

$$I \preceq P \preceq rI.$$

Indeed, if a point $x$ is in the ellipsoid $P$, then $x^T Px \leq 1$ which implies $x^T Ix \leq 1$, i.e., $x$ is in the sphere of radius 1. Thus, the ellipsoid $P$ is included in the sphere of radius 1. Similarly, $P$ contains the sphere of radius $r^{-\frac{1}{2}}$. This way, $P$ is sandwiched between these two spheres and making their radius as close as possible will make $P$ as 'round' as possible, as depicted on Figure 5.3. This constraint, along with the others (Lyapunov equation, symmetry and positive

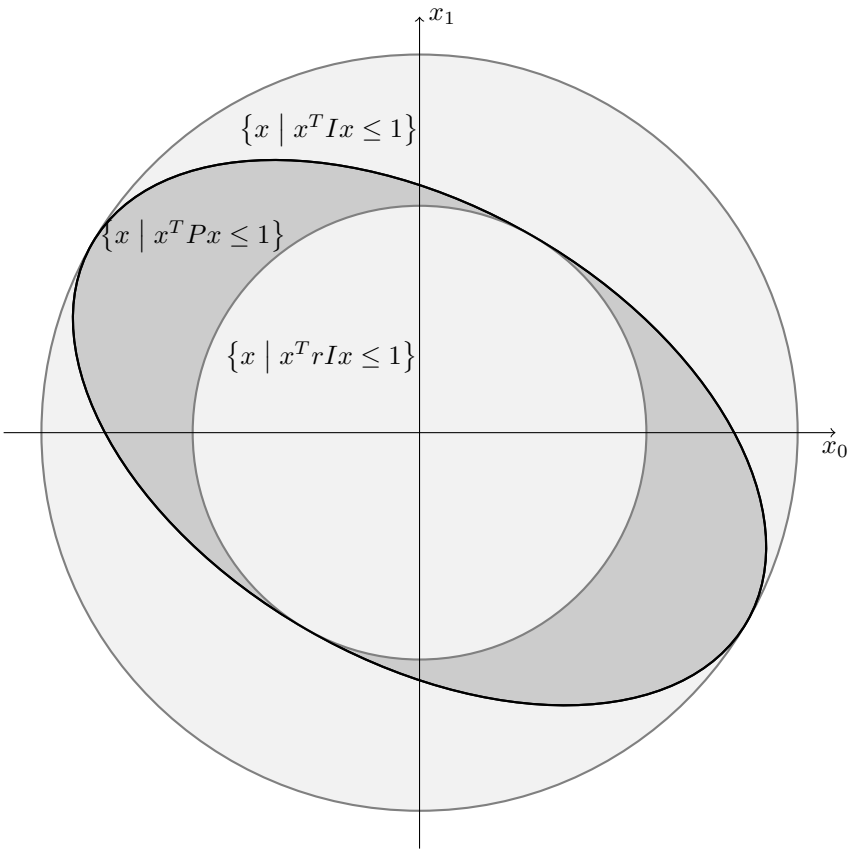



Figure 5.3 – Making the ellipsoid $P$ as 'round' as possible by sandwiching it between spheres of radius $r^{-\frac{1}{2}}$ and 1: $I \preceq P \preceq rI$ and minimizing $r$.

definiteness of $P$), can be expressed as a LMI, which is solved using the semi-definite programming techniques mentioned in Section 5.2.2:

$$\begin{array}{ll} \text{minimize} & r \\ \text{subject to} & A^T P A - P \prec 0 \\ & I \preceq P \preceq rI \\ & P^T = P. \end{array}$$

**Example 25.** *With the following matrix $A$ of the running example*

$$A := \begin{bmatrix} 0.9379 & -0.0381 & -0.0414 \\ -0.0404 & 0.968 & -0.0179 \\ 0.0142 & -0.0197 & 0.9823 \end{bmatrix}$$

*a semi-definite solver simply returns $r = 1$ and the identity matrix*

$$P = \begin{bmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{bmatrix}.$$

**Example 26.** *With the following matrix $A$*

$$A := \begin{bmatrix} 0.6227 & 0.3871 & -0.113 & 0.0102 \\ -0.3407 & 0.9103 & -0.3388 & 0.0649 \\ 0.0918 & -0.0265 & -0.7319 & 0.2669 \\ 0.2643 & -0.1298 & -0.9903 & 0.3331 \end{bmatrix}$$

*a semi-definite solver returns $r = 1.9645$ and the matrix*

$$P = \begin{bmatrix} 1.5277 & -0.2140 & -0.1352 & 0.0204 \\ -0.2140 & 1.4854 & -0.1529 & -0.0301 \\ -0.1352 & -0.1529 & 1.7485 & -0.3441 \\ 0.0204 & -0.0301 & -0.3441 & 1.2045 \end{bmatrix}$$

*(all figures being rounded to four digits).*

### 5.3.2 Preserving the Shape

Another approach [Yan92] is to minimize $r \in (0, 1)$ in the following inequality

$$A^T P A - rP \preceq 0. \tag{5.10}$$

Intuitively, this corresponds to finding the shape of ellipsoid that gets 'preserved' the best when the update $x_{k+1} = Ax_k$ is applied, as depicted on Figure 5.4. $r$ can be seen as the minimum contraction achieved by this update in the norm defined by $P$, hence the name *decay rate* given to this value by control theorists. This is the choice implicitly made in [Fer04] for a particular case of matrices $A$ of order 2.

With this technique however, the presence of a quadratic term $rP$ in the equation prevents the use of usual LMI solving tools 'as is'. To overcome this, the following property enables the choice of an approach where the value for $r$ is refined by dichotomy. Only a few steps are then required to obtain a good approximation of the optimal value.

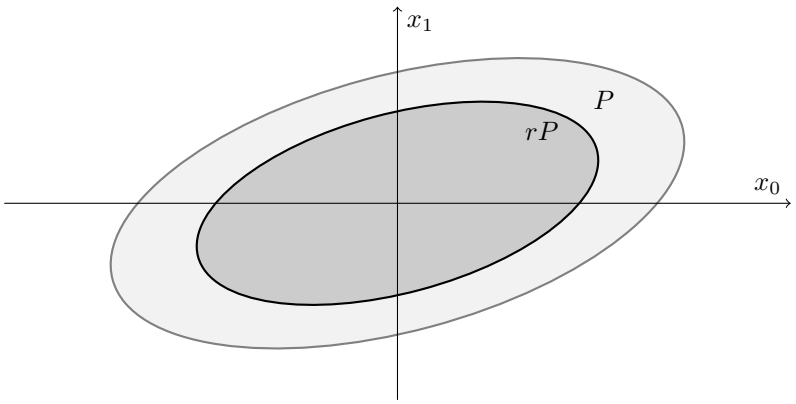



Figure 5.4 – Choice of the ellipsoid whose shape is the best preserved.

**Property 9.** *If Equation 5.10 admits as solution a positive definite matrix $P$ for a given $r$, then it is also the case for any $r' \geq r$.*

*Proof.* If $P$ is a solution for $r$ (i.e., $A^T P\, a - rP \preceq 0$ with $P$ positive definite), for any $r' \geq r$, it can be shown that $P$ is also a solution. Indeed

$$A^T P\, A - r'P = \left(A^T P\, a - rP\right) - (r' - r)\, P,$$

$A^T P\, A - rP \preceq 0$ by hypothesis and $-(r' - r)\, P \preceq 0$ since $P$ is positive definite and $r' - r \geq 0$, which concludes. □

**Example 27.** *With the following matrix $A$ of the running example:*

$$A := \begin{bmatrix} 0.9379 & -0.0381 & -0.0414 \\ -0.0404 & 0.968 & -0.0179 \\ 0.0142 & -0.0197 & 0.9823 \end{bmatrix},$$

*looking for a small $r \in (0, 1)$, the first value tested is $r = 0.5$, i.e., a solution to the following semi-definite program is looked for*

$$\begin{array}{ll} \textit{minimize} & 0 \\ \textit{subject to} & A^T P\, A - 0.5P \preceq 0 \\ & P \succ 0 \\ & P^T = P. \end{array}$$

*Since there is no solution, $r$ is now looked for in interval $(0.5, 1)$. $r = 0.75$ is tested, without more success, then $r = 0.875$, $r = 0.9375$, $r = 0.98675$ and $r = 0.984375$ are still unsuccessful. Finally $r = 0.9921875$ yields the following solution (all figures being rounded to four digits):*

$$P = \begin{bmatrix} 239.1338 & 37.5557 & 77.9203 \\ 37.5557 & 226.3640 & 65.8287 \\ 77.9203 & 65.8287 & 325.1628 \end{bmatrix}.$$

*Stopping here leaves $r \in (0.984375, 0.9921875)$ and the above matrix $P$ as solution for $r = 0.9921875$.*

**Example 28.** *With the following matrix $A$:*

$$A := \begin{bmatrix} 0.6227 & 0.3871 & -0.113 & 0.0102 \\ -0.3407 & 0.9103 & -0.3388 & 0.0649 \\ 0.0918 & -0.0265 & -0.7319 & 0.2669 \\ 0.2643 & -0.1298 & -0.9903 & 0.3331 \end{bmatrix}$$

*the following matrix $P$ can be found for $r = 0.7109375$:*

$$P = \begin{bmatrix} 1104.5465 & -468.5191 & 54.9427 & -102.8232 \\ -468.5191 & 1194.1353 & -247.1750 & 116.2385 \\ 54.9427 & -247.1750 & 1385.3882 & -442.9282 \\ -102.8232 & 116.2385 & -442.9282 & 550.1209 \end{bmatrix}.$$

### 5.3.3 All in One

The two previous methods were based only on $A$, completely abstracting $B$ away, which could lead to rather coarse abstractions. We try here to take both $A$ and $B$ into account by finding the ellipsoid $P$ included in the smallest possible sphere which is stable, i.e., such that

$$\forall x, \forall u, \|u\|_\infty \leq 1 \Rightarrow x^T P\, x \leq 1 \Rightarrow (Ax + Bu)^T P\, (Ax + Bu) \leq 1.$$

This is illustrated in Figure 5.5.

The previous condition can be rewritten

$$\forall x, \forall u, \left( \bigwedge_{i=0}^{p-1} \left(e_i^T u\right)^2 \leq 1 \right) \wedge x^T P\, x \leq 1 \Rightarrow (Ax + Bu)^T P\, (Ax + Bu) \leq 1$$

where $e_i$ is the $i$-th vector of the canonical basis (i.e., with all coefficients equal to 0 except the $i$-th one which is 1). This amounts to

$$\forall x, \forall u, \left( \bigwedge_{i=0}^{p-1} \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} 0 & 0 \\ 0 & E^{i,i} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1 \right) \wedge \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} P & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1$$
$$\Rightarrow \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} A^T P A & A^T P B \\ B^T P A & B^T P B \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1$$

where $E^{i,j}$ is the matrix with 0 everywhere except the coefficient at line $i$, column $j$ which is 1. Using the S-procedure (Theorem 1, page 38), this holds when there are $\tau$ and $\lambda_0, \ldots, \lambda_{p-1}$ all non negatives such that

$$\begin{bmatrix} -A^T P A & -A^T P B & 0 \\ -B^T P A & -B^T P B & 0 \\ 0 & 0 & 1 \end{bmatrix} - \tau \begin{bmatrix} -P & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & 1 \end{bmatrix} - \sum_{i=0}^{p-1} \lambda_i \begin{bmatrix} 0 & 0 & 0 \\ 0 & -E^{i,i} & 0 \\ 0 & 0 & 1 \end{bmatrix} \succeq 0 \tag{5.11}$$

As in Section 5.3.2, this is not an LMI since $\tau$ and $P$ are both variables. And again, there is a $\tau_{min} \in (0, 1)$ such that this inequality admits as solution a positive definite matrix $P$ if and only if $\tau \in (\tau_{min}, 1)$. This value $\tau_{min}$ can by the way be approximated thanks to the exact same procedure. Similarly to what was done in Section 5.3.1, $P$ is forced to be contained in the smallest possible sphere by maximizing $r$ in the additional constraint

$$P \succeq rI. \tag{5.12}$$

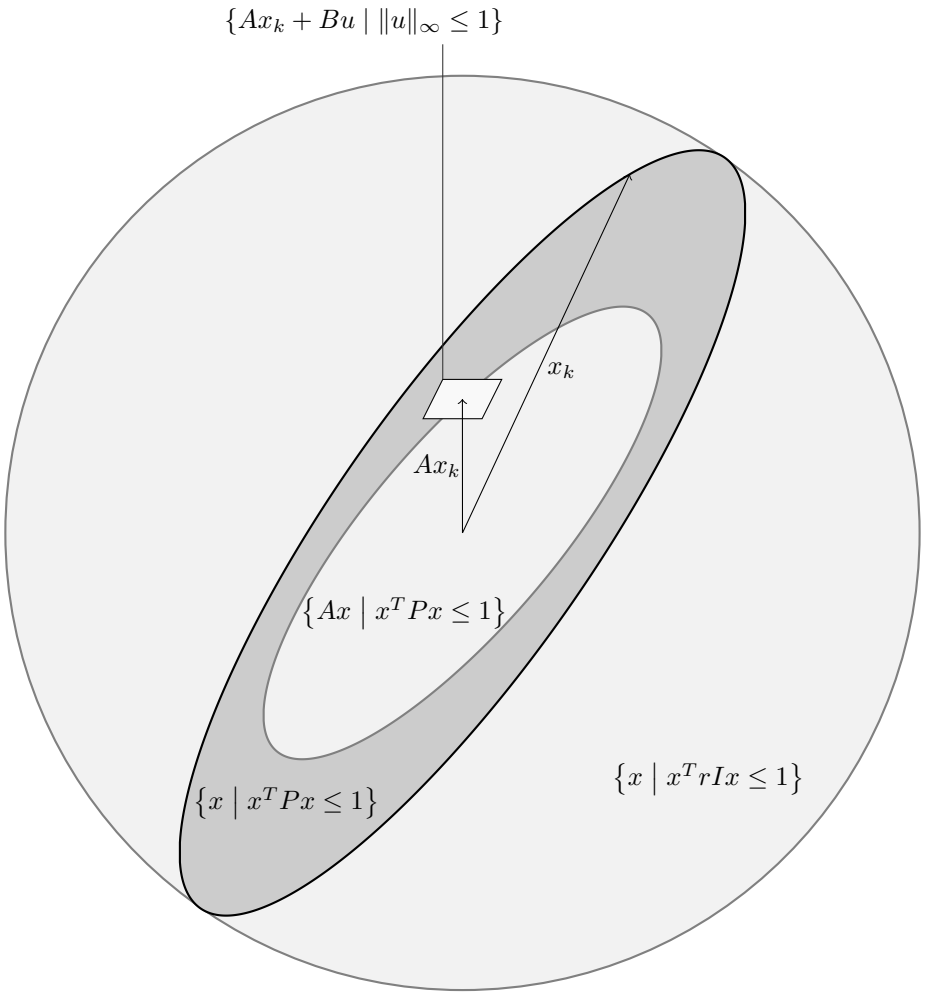



Figure 5.5 – Looking for an invariant ellipsoid included in the smallest possible sphere by maximizing $r$.

The function $f$ is then defined as the function mapping $\tau \in (\tau_{min}, 1)$ to the optimal value of the following semi-definite program

$$\begin{aligned} \text{maximize} \quad & r \\ \text{subject to} \quad & (5.11) \\ & (5.12) \\ & P^T = P \\ & \bigwedge_{i=0}^{p-1} \lambda_i \geq 0 \end{aligned}$$

This function $f$ can then be evaluated for a given input $\tau$ simply by solving the above semi-definite program. $f$ seems concave which could enable a smart optimization procedure. However, in practice, it is enough to just sample $f$ for some equally spaced values in the interval $(\tau_{min}, 1)$ and just keep the matrix $P$ obtained for the value enabling the greatest $r$.

**Example 29.** *With the following matrices $A$ and $B$ of the running example:*

$$A := \begin{bmatrix} 0.9379 & -0.0381 & -0.0414 \\ -0.0404 & 0.968 & -0.0179 \\ 0.0142 & -0.0197 & 0.9823 \end{bmatrix} \qquad B := \begin{bmatrix} 0.0237 \\ 0.0143 \\ 0.0077 \end{bmatrix},$$

*according to Example 27, $\tau_{min} = 0.9921875$.*

*Then $f$ is evaluated on a few points between $\tau_{min}$ and 1 (rounded figures):*

| $\tau$ | $f(\tau)$ |
|---|---|
| 0.9928 | 1.6064 |
| 0.9935 | 1.4653 |
| 0.9941 | 1.3231 |
| 0.9948 | 1.1798 |
| 0.9954 | 1.0355 |
| 0.9961 | 0.8902 |
| 0.9967 | 0.7440 |
| 0.9974 | 0.5970 |
| 0.9980 | 0.4490 |
| 0.9987 | 0.3002 |
| 0.9993 | 0.1505 |

*and the one giving the best value ($\tau = 0.9928$) is kept with the corresponding*

$$P = \begin{bmatrix} 12.6465 & -14.1109 & -10.5402 \\ -14.1109 & 25.6819 & 3.06577 \\ -10.5402 & 3.06577 & 29.5981 \end{bmatrix}.$$

**Example 30.** *With the following matrices $A$ and $B$:*

$$A := \begin{bmatrix} 0.6227 & 0.3871 & -0.113 & 0.0102 \\ -0.3407 & 0.9103 & -0.3388 & 0.0649 \\ 0.0918 & -0.0265 & -0.7319 & 0.2669 \\ 0.2643 & -0.1298 & -0.9903 & 0.3331 \end{bmatrix} \qquad B := \begin{bmatrix} 0.3064 & 0.1826 \\ -0.0054 & 0.6731 \\ 0.0494 & 1.6138 \\ -0.0531 & 0.4012 \end{bmatrix},$$

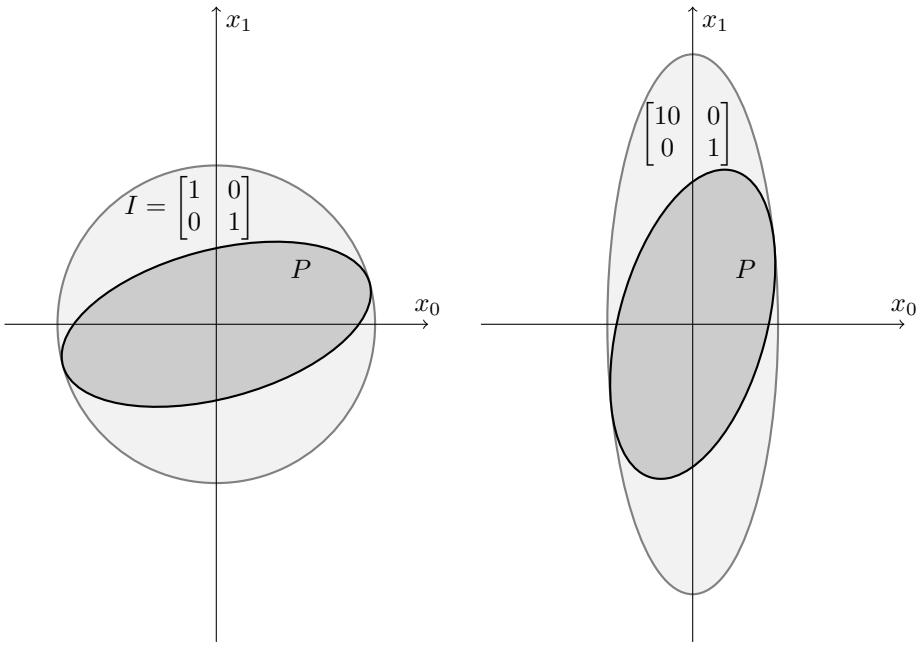



Figure 5.6 – Constraining ellipsoid $P$ to lie in a sphere (left) or in an ellipsoid flatter in a given direction (right).

*the following matrix $P$ can be found:*

$$P = \begin{bmatrix} 0.0484 & -0.0204 & 0.0108 & -0.0149 \\ -0.0204 & 0.0594 & -0.0381 & 0.0357 \\ 0.0108 & -0.0381 & 0.1344 & -0.1101 \\ -0.0149 & 0.0357 & -0.1101 & 0.1383 \end{bmatrix}.$$

### 5.3.4 Directed All in One

If a tighter bound is required on one of the variables, the identity matrix $I$ in inequality (5.12) can be replaced by a diagonal matrix with larger coefficients for variables of interest. For instance, to get a smaller bound on the first variable $x_0$, the matrix $I$ can be replace by $\begin{bmatrix} 10 & 0 \\ 0 & I \end{bmatrix}$.

This intuitively corresponds to minimize the radius of an ellipsoid containing $P$ flatter on the dimension of interest instead of a sphere. This is illustrated on Figure 5.6.

**Example 31.** *With the following matrices $A$ and $B$ of the running example:*

$$A := \begin{bmatrix} 0.9379 & -0.0381 & -0.0414 \\ -0.0404 & 0.968 & -0.0179 \\ 0.0142 & -0.0197 & 0.9823 \end{bmatrix} \qquad B := \begin{bmatrix} 0.0237 \\ 0.0143 \\ 0.0077 \end{bmatrix},$$

*expressing a higher interest in the first variable as exposed above gives*

$$P = \begin{bmatrix} 12.6465 & -14.1109 & -10.5402 \\ -14.1109 & 25.6819 & 3.06577 \\ -10.5402 & 3.06577 & 29.5981 \end{bmatrix}.$$

**Example 32.** *With the following matrices $A$ and $B$:*

$$A := \begin{bmatrix} 0.6227 & 0.3871 & -0.113 & 0.0102 \\ -0.3407 & 0.9103 & -0.3388 & 0.0649 \\ 0.0918 & -0.0265 & -0.7319 & 0.2669 \\ 0.2643 & -0.1298 & -0.9903 & 0.3331 \end{bmatrix} \quad B := \begin{bmatrix} 0.3064 & 0.1826 \\ -0.0054 & 0.6731 \\ 0.0494 & 1.6138 \\ -0.0531 & 0.4012 \end{bmatrix},$$

*expressing a higher interest in the first variable as exposed above gives*

$$P = \begin{bmatrix} 0.1276 & -0.0508 & 0.0087 & -0.0013 \\ -0.0508 & 0.1742 & -0.0829 & 0.0545 \\ 0.0087 & -0.0829 & 0.0952 & -0.0735 \\ -0.0013 & 0.0545 & -0.0735 & 0.0836 \end{bmatrix}.$$

## 5.4 Finding a Stable Ratio

Once a *shape* of ellipsoid $P$ has been chosen, a *radius* $\lambda$ has to be determined in order to get an actual invariant $\{x \mid x^T P x \leq \lambda\}$. This can be done through Kleene iterations, as presented in Section 3.4 using a widening with threshold or, better, some acceleration method [BSC12]. However, this issue will not be more deeply addressed here, since Part III will offer a better solution: policy iterations.

Finally, once an invariant is obtained, it can easily be projected into any linear or quadratic template domain by solving some semi-definite programming problems. This includes, but is not limited to, the intervals domain, the octagons [Min01] or the linear templates domain [SSM05].

# Chapter 6

# Floating Point Issues

Two fundamentally different issues must be considered:

**the analyzed system** contains floating point computations with rounding errors making it behave differently from the way it would if the same computations were done with real numbers, this is discussed in Section 6.1;

**the implementation of the abstract domain** is also carried out with floating point computations for the sake of efficiency, this usually works well in practice but can give erroneous results, hence the need for some a posteriori validation, see Section 6.2 for further details.

## 6.1 Taking Rounding Errors Into Account

The sum or product of two floating point values is, in all generality, not representable as a floating point value and must consequently be rounded. The accumulation of rounding errors can potentially lead to far different results from the ones expected with real numbers [Mon08], thus floating point computations must be taken into account in our analysis.

The rounding errors can be of two different types :

- for normalized numbers represented with a fixed number of bits, we get a relative error: $\text{round}(a \times b) = (1 + \delta)(a \times b)$ where $|\delta| \leq \epsilon$;
- for denormalized numbers (i.e ones very close to 0), we get an absolute error: $\text{round}(a \times b) = (a \times b) + \eta$ where $|\eta| \leq \omega$.

The actual values of the constants $\epsilon$ and $\omega$ depend on the characteristics of the considered floating point system. For instance we will take $\epsilon = 2^{-23}$ and $\omega = 2^{-149}$ for single precision[1]. A common and easy solution to take both possible errors into account is to sum them which in practice leads only to a very slight overapproximation: $\text{round}(a \times b) = (1 + \delta)(a \times b) + \eta$. Although only multiplication is illustrated here, the method works exactly the same way for any other floating point operation[2].

[1] Type `float` in C.

[2] For addition and substraction it can even be noticed that, when denormalized numbers are implemented properly, there is no absolute error: $\text{round}(a \pm b) = (1 + \delta)(a \pm b), |\delta| \leq \epsilon$.

Combining these elementary errors we will get a simple postprocessing for each Kleene iteration of Section 5.4 to soundly overapproximate rounding errors.

**Definition 31.** $\mathbb{F} \subset \mathbb{R}$ *denotes the set of floating point values and* $\mathrm{fl}(e) \in \mathbb{F}$ *represents the floating point evaluation of expression* $e$ *with any rounding mode and any order of evaluation*[3].

**Example 33.** *The value* $\mathrm{fl}(1+2+3)$ *can be either* $\mathrm{round}(1+\mathrm{round}(2+3))$ *or* $\mathrm{round}(\mathrm{round}(1+2)+3)$.

**Lemma 1.** *Assuming*[4] $2n\epsilon < 1$, *we have for all* $a_i, x_i \in \mathbb{F}$

$$\left|\mathrm{fl}\left(\sum_{i=1}^{n} a_i x_i\right) - \sum_{i=1}^{n} a_i x_i\right| \leq \gamma_n \left(\sum_{i=1}^{n} |a_i x_i|\right) + \gamma_{2n}\frac{\omega}{\epsilon} \tag{6.1}$$

*where* $\gamma_n := \frac{n\epsilon}{1-n\epsilon}$.

**Remark 11.** *These are fairly classic notations and results [Hig96, Rum10].*

*Proof.* The proof is carried out by induction on $n$.

- For $n=1$: $|\mathrm{fl}(a_1x_1) - a_1x_1| \leq \epsilon\,|a_1x_1| + \omega \leq \gamma_1\,|a_1x_1| + \gamma_2\frac{\omega}{\epsilon}$.
- Assuming (6.1) for $n$, we can replace the absolute value and the inequality with a more convenient equality by introducing $u$ such that $|u| \leq 1$ and

  $$\mathrm{fl}\left(\sum_{i=1}^{n} a_i x_i\right) - \sum_{i=1}^{n} a_i x_i = u\left(\gamma_n \left(\sum_{i=1}^{n} |a_i x_i|\right) + \gamma_{2n}\frac{\omega}{\epsilon}\right)$$

  then $\delta_0 := u\epsilon$ and $\eta_0 := u\omega$ satisfy $|\delta_0| \leq \epsilon$, $|\eta_0| \leq \omega$ and

  $$\mathrm{fl}\left(\sum_{i=1}^{n} a_i x_i\right) - \sum_{i=1}^{n} a_i x_i = \frac{n\delta_0}{1-n\epsilon}\left(\sum_{i=1}^{n} |a_i x_i|\right) + \frac{2n\eta_0}{1-2n\epsilon}.$$

  Thus, there are some $\delta_1, \delta_2, \delta_3$ and $\eta_1, \eta_2$ such that $|\delta_j| \leq \epsilon$, $|\eta_j| \leq \omega$ and (up to some reindexing)

$$\begin{aligned}
\mathrm{fl}\left(\sum_{i=1}^{n+1} a_i x_i\right) &= \mathrm{fl}\left(\sum_{i=1}^{n} a_i x_i + a_{n+1}x_{n+1}\right) \\
&= (1+\delta_1)\left(\mathrm{fl}\left(\sum_{i=1}^{n} a_i x_i\right) + \mathrm{fl}\left(a_{n+1}x_{n+1}\right)\right) + \eta_1 \\
&= (1+\delta_1)\left(\begin{array}{l}\displaystyle\sum_{i=1}^{n} a_i x_i + \frac{n\delta_0}{1-n\epsilon}\left(\sum_{i=1}^{n} |a_i x_i|\right) + \frac{2n\eta_0}{1-2n\epsilon} \\ +(1+\delta_2)a_{n+1}x_{n+1} + \eta_2\end{array}\right) + \eta_1 \\
&= (1+\delta_1)\left(\begin{array}{l}\displaystyle\sum_{i=1}^{n} a_i x_i + a_{n+1}x_{n+1} \\ \displaystyle+\frac{n\delta_0}{1-n\epsilon}\left(\sum_{i=1}^{n} |a_i x_i|\right) + \delta_3\,|a_{n+1}x_{n+1}| \\ +\frac{2n\eta_0}{1-2n\epsilon} + \eta_2\end{array}\right) + \eta_1
\end{aligned}$$

[3] Order of evaluation matters since floating point addition is not associative.
[4] For $\epsilon = 2^{-23}$, this means $n < 2^{22}$ which is a very reasonable assumption.

where $\delta_1$ and $\eta_1$ come from the rounding of the sum of $\sum_{i=1}^{n} a_i x_i$ and $a_{n+1}x_{n+1}$, $\delta_2$ and $\eta_2$ come from the rounding of the product of $a_{n+1}$ and $x_{n+1}$ and $\delta_3$ is just $\pm\delta_2$.

Then, there are some $\delta_4, \delta_5, \delta_6$ and $\eta_3$ such that $|\delta_j| \leq \epsilon$, $|\eta_j| \leq \omega$ and

$$\begin{aligned}
\mathrm{fl}\left(\sum_{i=1}^{n+1} a_i x_i\right) &= (1+\delta_1)\left(\begin{array}{l}\displaystyle\sum_{i=1}^{n+1} a_i x_i + \tfrac{n\delta_4}{1-n\epsilon}\left(\sum_{i=1}^{n+1}|a_i x_i|\right) \\ \quad + \tfrac{2n\eta_0}{1-2n\epsilon} + \eta_2\end{array}\right) + \eta_1 \\
&= \sum_{i=1}^{n+1} a_i x_i + \delta_1\left(\sum_{i=1}^{n+1} a_i x_i\right) + (1+\delta_1)\frac{n\delta_4}{1-n\epsilon}\left(\sum_{i=1}^{n+1}|a_i x_i|\right) \\
&\qquad + (1+\delta_1)\frac{2n\eta_0}{1-2n\epsilon} + (1+\delta_1)\eta_2 + \eta_1 \\
&= \sum_{i=1}^{n+1} a_i x_i + \left(\delta_5 + (1+\delta_1)\frac{n\delta_4}{1-n\epsilon}\right)\left(\sum_{i=1}^{n+1}|a_i x_i|\right) \\
&\qquad + (1+\delta_1)\frac{2n\eta_0}{1-2n\epsilon} + (1+\delta_1)\eta_2 + \eta_1 \\
&= \sum_{i=1}^{n+1} a_i x_i + \frac{(n+1)\delta_6}{1-(n+1)\epsilon}\left(\sum_{i=1}^{n+1}|a_i x_i|\right) + \frac{2(n+1)\eta_3}{1-2(n+1)\epsilon}
\end{aligned}$$

which proves (6.1) for $n+1$. □

**Lemma 2.** *Given a positive definite matrix $P \in \mathbb{R}^{n\times n}$ and $s \in \mathbb{R}$ such that $I \preceq sP$, for any vector $x \in \mathbb{R}^n$ and $\lambda \in \mathbb{R}$, if $x^T P x \leq \lambda$ then $\|x\|_\infty \leq \sqrt{s\lambda}$.*

*Proof.* Since $I \preceq sP$ and $x^T P x \leq \lambda$, then $x^T I x \leq x^T s P x \leq s\lambda$, i.e., $\|x\|_2^2 \leq s\lambda$. Thus $\|x\|_\infty \leq \|x\|_2 \leq \sqrt{s\lambda}$. □

**Lemma 3.** *Given a positive definite matrix $P \in \mathbb{R}^{n\times n}$, two matrices $A \in \mathbb{F}^{n\times n}$ and $B \in \mathbb{F}^{n\times p}$, two vectors $x, u \in \mathbb{F}^n$ and two scalars $s, \lambda \in \mathbb{R}$, assuming $x^T P x \leq \lambda$, $\|u\|_\infty \leq 1$, $I \preceq sP$ and $2(n+p)\epsilon < 1$, there is a diagonal matrix $D \in \mathbb{R}^{n\times n}$ such that $|D_{i,i}| \leq 1$ and*

$$\mathrm{fl}(Ax+Bu) = Ax + Bu + D\left(\sqrt{s\lambda}\gamma_{n+p}\,|A|\,i + \gamma_{n+p}\,|B|\,i + \gamma_{2(n+p)}\frac{\omega}{\epsilon}i\right) \quad (6.2)$$

*where $i := [1, \cdots, 1]^T \in \mathbb{R}^n$.*

*Proof.* According to Lemma 1, there are $\delta_0$, $\eta_0$ such that $|\delta_0| \leq \epsilon$, $|\eta_0| \leq \omega$ and

$$\mathrm{fl}(Ax+Bu)_i = (Ax+Bu)_i + \gamma_{n+p}\frac{\delta_0}{\epsilon}\left(\sum_{j=1}^{n}|A_{i,j}x_j| + \sum_{j=1}^{p}|B_{i,j}u_j|\right) + \gamma_{2(n+p)}\frac{\eta_0}{\epsilon}.$$

Using Lemma 2, and the hypothesis $\|u\|_\infty \leq 1$, we get

$$\mathrm{fl}(Ax+Bu)_i = (Ax+Bu)_i + \sqrt{s\lambda}\gamma_{n+p}\frac{\delta_1}{\epsilon}\sum_{j=1}^{n}|A_{i,j}| + \gamma_{n+p}\frac{\delta_2}{\epsilon}\sum_{j=1}^{p}|B_{i,j}| + \gamma_{2(n+p)}\frac{\eta_0}{\epsilon}$$

for some $\delta_1, \delta_2$ such that $|\delta_1| \leq \epsilon$, $|\delta_2| \leq \epsilon$. Thus, there is some diagonal matrix $D$ such that $|D_{i,i}| \leq 1$ and (6.2) holds. □

**Lemma 4.** *Given any positive definite matrix $P$, any scalar $s'$ such that $P \preceq s'I$ and any diagonal matrix $D$ such that $|D_{i,i}| \leq 1$, for all vector $x$:*

$$x^T D^T P D x \leq s' \|x\|_2^2.$$

*Proof.* We have $x^T D^T P D x \leq x^T D^T \left(s'I\right) D x \leq s' \|Dx\|_2^2 \leq s' \|x\|_2^2$. □

**Theorem 2.** *Given a positive definite matrix $P \in \mathbb{R}^{n\times n}$, two matrices $A \in \mathbb{F}^{n\times n}$ and $B \in \mathbb{F}^{n\times p}$, two vectors $x, u \in \mathbb{F}^n$ and four scalars $s, s', \lambda, \lambda' \in \mathbb{R}$, assuming $I \preceq sP$, $P \preceq s'I$, $2(n+p)\epsilon < 1$, $x^T P x \leq \lambda$, $\|u\|_\infty \leq 1$ and $(Ax+Bu)^T P (Ax+Bu) \leq \lambda'$, we have*

$$\mathrm{fl}(Ax+Bu)^T P\, \mathrm{fl}(Ax+Bu) \leq \left(\sqrt{\lambda'} + \sqrt{\lambda} a + b + c\right)^2$$

*with $a := \sqrt{ss'}\gamma_{n+p}\sqrt{n}\|A\|_2$, $b := \sqrt{s'}\gamma_{n+p}\sqrt{n}\|B\|_2$ and $c := \sqrt{s'}\gamma_{2(n+p)}\frac{\omega}{\epsilon}\sqrt{n}$.*

**Remark 12.** *The minimal value of $ss'$ with $I \preceq sP$ and $P \preceq s'I$ is the condition number of $P$. Thus, the better $P$ is conditioned, the smaller the error bound.*

*Proof.* For the sake of conciseness, let us first define a few shortcuts: $y := Ax+Bu$, $r := \sqrt{s}\gamma_{n+p}\left|A\right| i$ and $f := \gamma_{n+p}\left|B\right| i + \gamma_{2(n+p)}\frac{\omega}{\epsilon} i$. Thus, according to Lemma 3

$$\mathrm{fl}(y) = y + D\left(\sqrt{\lambda} r + f\right)$$

with $D$ a diagonal matrix with $|D_{i,i}| \leq 1$. Therefore

$$\begin{aligned}
\mathrm{fl}(y)^T P\, \mathrm{fl}(y) &= \left(y + D\left(\sqrt{\lambda} r + f\right)\right)^T P \left(y + D\left(\sqrt{\lambda} r + f\right)\right) \\
&= y^T P y + \left(\sqrt{\lambda} r + f\right)^T D^T P D \left(\sqrt{\lambda} r + f\right) + 2 y^T P D \left(\sqrt{\lambda} r + f\right) \\
&= y^T P y + \lambda r^T D^T P D r + f^T D^T P D f + 2\sqrt{\lambda} r^T D^T P D f \\
&\qquad + 2 y^T P D \sqrt{\lambda} r + 2 y^T P D f
\end{aligned}$$

and by the Cauchy-Schwartz inequality[5] ($P$ being positive definite, the function $x, y \mapsto x^T P y$ is a dot product)

$$\begin{aligned}
\mathrm{fl}(y)^T P\, \mathrm{fl}(y) \leq\ & y^T P y + \lambda r^T D^T P D r + f^T D^T P D f \\
&+ 2\sqrt{\lambda}\sqrt{r^T D^T P D r}\sqrt{f^T D^T P D f} + 2\sqrt{\lambda}\sqrt{y^T P y}\sqrt{r^T D^T P D r} \\
&+ 2\sqrt{y^T P y}\sqrt{f^T D^T P D f}.
\end{aligned}$$

Then, according to Lemma 4

$$\begin{aligned}
\mathrm{fl}(y)^T P\, \mathrm{fl}(y) \leq\ & y^T P y + \lambda s' \|r\|_2^2 + s' \|f\|_2^2 + 2 s' \sqrt{\lambda} \|r\|_2 \|f\|_2 \\
&+ 2\sqrt{s'}\sqrt{\lambda}\sqrt{y^T P y}\|r\|_2 + 2\sqrt{s'}\sqrt{y^T P y}\|f\|_2.
\end{aligned}$$

Thus

$$\begin{aligned}
\mathrm{fl}(y)^T P\, \mathrm{fl}(y) &\leq \lambda' + \lambda s' \|r\|_2^2 + s' \|f\|_2^2 + 2 s' \sqrt{\lambda} \|r\|_2 \|f\|_2 + 2\sqrt{s'}\sqrt{\lambda}\sqrt{\lambda'}\|r\|_2 \\
&\qquad + 2\sqrt{s'}\sqrt{\lambda'}\|f\|_2 \\
&\leq \left(\sqrt{\lambda'} + \sqrt{\lambda}\sqrt{s'}\|r\|_2 + \sqrt{s'}\|f\|_2\right)^2.
\end{aligned}$$

[5] Reminder: with $P$ positive definite, $x^T P\, y \leq \sqrt{x^T P x}\sqrt{y^T P y}$.

Finally

$$\|r\|_2 = \sqrt{s}\gamma_{n+p}\| \, |A| \, i\|_2 \leq \sqrt{s}\gamma_{n+p}\sqrt{n}\|A\|_2$$

and, thanks to the triangular inequality

$$\|f\|_2 \leq \gamma_{n+p}\| \, |B| \, i\|_2 + \gamma_{2(n+p)}\frac{\omega}{\epsilon}\|i\|_2 = \gamma_{n+p}\sqrt{n}\|B\|_2 + \gamma_{2(n+p)}\frac{\omega}{\epsilon}\sqrt{n}.$$

which allows to conclude. □

**Remark 13.** *To keep things somewhat "readable", matrices $A$ and $B$ were assumed to have floating point coefficients ($A \in \mathbb{F}^{n\times n}$ and $B \in \mathbb{F}^{n\times p}$). However, control theorists commonly use decimal values (such as 0.1 or 0.9) which are not floating point values. The previous theorem remains valid in this case ($A \in \mathbb{R}^{n\times n}$ and $B \in \mathbb{R}^{n\times p}$) with adapted values for the parameters*[6] *$a$, $b$ and $c$: $a := \sqrt{ss'}\gamma_{n+p+1}\sqrt{n}\|A\|_2$, $b := \sqrt{s'}\gamma_{n+p+1}\sqrt{n}\|B\|_2$ and $c := \sqrt{s'}\left(\frac{n\sqrt{s}+p}{1-(n+p)\epsilon} + \frac{2(n+p)}{1-2(n+p)\epsilon}\right)\omega\sqrt{n}$.*

**To Sum Up,** provided that the number of variables of the system plus its number of inputs is less than $(2\epsilon)^{-1}$, which is a very reasonable assumption[7], we just have to compute $a$, $b$ and $c$ once as defined above then apply to each Kleene iteration of Section 5.4 the postprocessing $\lambda' \mapsto \left(\sqrt{\lambda'} + \sqrt{\lambda}a + b + c\right)^2$ to take into account computation with floats, whatever the rounding mode and the order of evaluation.

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

## 6.2 Checking Soundness of the Result

Because the LMI solver used is implemented with floating point computations, we have no guarantee on the results it provides[8]. Hence the need to check them. This will amount to checking that a given matrix is actually positive definite which will be done by carefully bounding the rounding error on a floating point Cholesky decomposition.

Given a matrix $P$, i.e., an ellipsoid *shape*, and a *ratio* $\lambda$, the goal is to check that $\left\{x \mid x^T P x \leq \lambda\right\}$ is actually an invariant of the system starting from some initial point $x_0$ and being updated by the transition relation $x_{k+1} = \mathrm{fl}\left(Ax_k + Bu_k\right)$ where $u_k$ remains bounded: $\|u_k\|_\infty \leq 1$.

The first thing to check is that the invariant initially holds, that is $x_0^T P x_0 \leq \lambda$. This can easily be done by computing the left value of this inequality (for instance with rationals or interval arithmetic) and comparing it to $\lambda$.

Thus, most of the work resides in proving that the candidate invariant is inductive. That is

$$\forall x, \forall u, \|u\|_\infty \leq 1 \Rightarrow x^T P x \leq \lambda \Rightarrow \left(\mathrm{fl}(Ax + Bu)\right)^T P \, \mathrm{fl}(Ax + Bu) \leq \lambda.$$

[6] Obtained by replacing $a_i$ with $\mathrm{fl}(a_i) = (1+\delta)a_i + \eta$, $|\delta| \leq \epsilon$, $|\eta| \leq \omega$ in Lemma 1 and carrying on the remaining of the proof in a very similar way.

[7] $(2\epsilon)^{-1} = 2^{22}$ for single precision for instance.

[8] There also exist guaranteed SDP solvers now [JCK07] over and underapproximating the primal and the dual problem to guarantee an error bound on the result. However we only need to check the final result. Thanks to Éric Goubault for pointing that to us.

According to Theorem 2, a sufficient condition is to find a $\lambda'$ such that

$$\forall x, \forall u, \|u\|_\infty \leq 1 \Rightarrow x^T P x \leq \lambda \Rightarrow (Ax+Bu)^T P (Ax+Bu) \leq \lambda'$$

and $\left(\sqrt{\lambda'} + \sqrt{\lambda} a + b + c\right)^2 \leq \lambda$ with $a$, $b$ and $c$ defined as in Theorem 2. The previous condition amounts to

$$\forall x, \forall u, \left(\bigwedge_{i=0}^{p-1} \left(e_i^T u\right)^2 \leq 1\right) \Rightarrow x^T P x \leq \lambda \Rightarrow (Ax+Bu)^T P (Ax+Bu) \leq \lambda'$$

where $e_i$ is the $i$-th vector of the canonical basis (i.e., with all coefficients equal to 0 except the $i$-th one which is 1). This amount to

$$\forall x, \forall u, \left(\bigwedge_{i=0}^{p-1} \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} 0 & 0 \\ 0 & E^{i,i} \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq 1\right) \wedge \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} P & 0 \\ 0 & 0 \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \lambda$$
$$\Rightarrow \begin{bmatrix} x \\ u \end{bmatrix}^T \begin{bmatrix} A^T P A & A^T P B \\ B^T P A & B^T P B \end{bmatrix} \begin{bmatrix} x \\ u \end{bmatrix} \leq \lambda'$$

where $E^{i,j}$ is the matrix with 0 everywhere except the coefficient at line $i$, column $j$ which is 1. According to Theorem 1, page 38, it is enough to prove the existence of $\tau$ and $\lambda_0, \ldots, \lambda_{n-1}$ all non negatives such that

$$\begin{bmatrix} -A^T P A & -A^T P B & 0 \\ -B^T P A & -B^T P B & 0 \\ 0 & 0 & \lambda' \end{bmatrix} - \tau \begin{bmatrix} -P & 0 & 0 \\ 0 & 0 & 0 \\ 0 & 0 & \lambda \end{bmatrix} - \sum_{i=0}^{p-1} \lambda_i \begin{bmatrix} 0 & 0 & 0 \\ 0 & -E^{i,i} & 0 \\ 0 & 0 & 1 \end{bmatrix} \succeq 0.$$

Such $\tau$, $\lambda'$, $\lambda_0, \ldots, \lambda_{n-1}$ can be computed (as floating point values) using a semi-definite programming solver. Then, the matrix in previous inequality can be computed exactly using rational numbers (or it could be approximated using cheaper floating point interval arithmetic) and it remains to check its positive definiteness.

This can be done by carefully bounding the rounding error on a floating point Cholesky decomposition [Rum06][9]. Proof of positive definiteness of an $n \times n$ matrix can then be achieved with $\mathrm{O}(n^3)$ floating point operations, which in practice induces only a very small overhead to the whole analysis.

[9] Thanks to Timothy WANG for pointing this to us.

# Chapter 7

# Experimental Results

All the elements presented in this part have been implemented as an autonomous linear system analysis engine. The tool is composed of three parts:

- The core mathematical computations are done with Scilab [Tea], mainly with the LMI solver [NDEG95] from an OCaml front-end. This part is a set of functions that implement the algorithms presented in Section 5.3, as well as projections of ellipsoids over intervals. Computation in Scilab are done using double precision floats.
- The front-end is an OCaml code using rational numbers (Num library). It loads the $A$ and $B$ matrices and interacts with Scilab to compute the different sequence of calls to Scilab functions.
- A last part, also in OCaml, interfaces the obtained quadratic form with a particular C implementation of a Cholesky decomposition [Rum06] to ensure its stability as explained in Section 6.2.

The code is released under a GPL license and is available at `https://cavale.enseeiht.fr/phd2013/`.

Experiments were conducted on a set of stable linear systems. These systems were extracted from [Fer05], [AGG10] or from basic controllers found in the literature. Table 7.1 details the time spent computing bounds for these systems through the four different methods of Section 5.3 while Table 7.2 compares the bounds obtained with the maximum reachable values of the systems.

From these tables, it can be noticed that the heuristic looking for the roundest possible ellipsoid (Section 5.3.1) yields very poor results. The second heuristic, based on the decay rate (Section 5.3.2) leads to way better results while being a bit more expensive (by a factor about two). The heuristic taking matrix $B$ into account (Section 5.3.3) gives even better results whereas it is again a bit more expensive. Finally, the last heuristic can enable to get slightly better results on a particular variable while sacrificing results on other variables, as expected.

| | M. | $t_P$ (s) | $t_\lambda$ (s) | $t_{valid.}$ (s) |
|---|---|---|---|---|
| Ex. 1 | I | 0.08 | 0.60 | 0.01 |
| From [Fer05, slides] | P | 0.13 | 0.36 | 0.01 |
| n=2, 1 input | U | 0.21 | 0.20 | 0.01 |
| | D | 0.20 | 0.20 | 0.01 |
| Ex. 2 | I | 0.10 | 0.63 | 0.02 |
| From [Fer05, slides] | P | 0.21 | 0.37 | 0.01 |
| n=4, 1 input | U | 0.33 | 0.22 | 0.01 |
| | D | 0.36 | 0.23 | 0.01 |
| Ex. 3 | I | 0.08 | 0.47 | 0.03 |
| Discretized | P | 0.13 | 0.45 | 0.02 |
| lead-lag controller | U | 0.16 | 0.21 | 0.02 |
| n=2, 1 input | D | 0.14 | 0.21 | 0.02 |
| Ex. 4 | I | 0.08 | 0.33 | 0.02 |
| Linear quadratic | P | 0.14 | 0.29 | 0.02 |
| gaussian regulator | U | 0.16 | 0.22 | 0.02 |
| n=3, 1 input | D | 0.17 | 0.22 | 0.03 |
| Ex. 5 Observer | I | 0.09 | 0.76 | 0.03 |
| based controller for | P | 0.17 | 0.43 | 0.03 |
| a coupled mass system | U | 0.27 | 0.23 | 0.03 |
| n=4, 2 inputs | D | 0.38 | 0.24 | 0.03 |
| Ex. 6 | I | 0.11 | 0.65 | 0.03 |
| Butterworth low-pass | P | 0.22 | 0.37 | 0.02 |
| filter | U | 0.56 | 0.25 | 0.02 |
| n=5, 1 input | D | 0.36 | 0.23 | 0.02 |
| Ex. 7 | I | 0.08 | 0.22 | 0.01 |
| Dampened oscillator | P | 0.11 | 0.22 | 0.01 |
| from [AGG10] | U | 0.25 | 0.17 | 0.01 |
| n=2, no input | D | 0.25 | 0.16 | 0.01 |
| Ex. 8 | I | 0.08 | 0.23 | 0.02 ($\perp$) |
| Harmonic oscillator | P | 0.23 | 0.23 | 0.01 |
| from [AGG10] | U | 0.13 | 0.17 | 0.01 |
| n=2, no input | D | 0.13 | 0.17 | 0.01 |

Table 7.1 – Result of the experiments: quadratic invariants computation. Times are expressed in seconds, $t_P$ is the time spent to compute the shape of the ellipsoid, $t_\lambda$ the one spent to find an appropriate ratio $\lambda$ and project the resulting invariant on intervals and $t_{valid.}$ is the time needed to validate the stability of the resulting ellipsoid, as explained in Section 6.2 ($\perp$ means that the check failed). I, P, U and D correspond respectively to the methods of Sections 5.3.1, 5.3.2, 5.3.3 and 5.3.4 (targeting the first variable). All computations were performed on an Intel Core2 @ 2.66GHz.

| | M. | Bounds | Reachable |
|---|---|---|---|
| Ex. 1 | I | $10801, 12633$ | $14.84, 14.84$ |
| | P | $22.18, 26.51$ | |
| | U | $16.20, 17.56$ | |
| | D | $16.14, 17.76$ | |
| Ex. 2 | I | $1661, 1795, 843, 1288$ | $1.42, 1.42, 1.00, 1.00$ |
| | P | $5.50, 6.71, 2.20, 3.44$ | |
| | U | $1.69, 1.92, 2.13, 2.42$ | |
| | D | $1.64, 1.84, 2.25, 2.52$ | |
| Ex. 3 | I | $60.93, 60.52$ | $3.97, 20.00$ |
| | P | $36.55, 35.50$ | |
| | U | $28.83, 25.85$ | |
| | D | $27.58, 32.89$ | |
| Ex. 4 | I | $1.26, 1.26, 1.26$ | $0.32, 0.24, 0.22$ |
| | P | $1.21, 1.23, 1.06$ | |
| | U | $0.68, 0.41, 0.28$ | |
| | D | $0.68, 0.41, 0.28$ | |
| Ex. 5 | I | $4980, 5061, 4768, 5693$ | $2.79, 2.73, 3.50, 3.30$ |
| | P | $6.37, 6.20, 6.07, 9.57$ | |
| | U | $4.97, 4.90, 4.77, 4.63$ | |
| | D | $3.05, 3.45, 6.84, 6.23$ | |
| Ex. 6 | I | $253, 261, 251, 280, 286$ | $1.42, 0.91, 1.44, 1.52, 2.14$ |
| | P | $3.11, 4.30, 4.15, 8.16, 8.81$ | |
| | U | $2.21, 1.10, 1.87, 1.98, 2.83$ | |
| | D | $1.75, 1.07, 1.81, 3.30, 3.94$ | |
| Ex. 7 | I | $1.46, 1.46$ | $1.29, 1.00$ |
| | P | $2.09, 2.14$ | |
| | U | $1.45, 1.45$ | |
| | D | $1.45, 1.45$ | |
| Ex. 8 | I | $\top$ | $1.10, 1.00$ |
| | P | $1.42, 1.42$ | |
| | U | $1.42, 1.42$ | |
| | D | $1.42, 1.42$ | |

Table 7.2 – Result of the experiments: bounds derived from quadratic invariants. The examples are the same and I, P, U and D have the same meaning than in Table 7.1. Column 'Bounds' gives the bounds on absolute values of each variables of the system proved by the tool whereas column 'Reachable' gives the maximum reachable values for comparison purpose.

# Chapter 8

# Conclusion

This part presented a set of analysis allowing us to characterize quadratic invariants, i.e. ellipsoids, for a subset of linear systems: inherently stable linear systems subject to bounded inputs.

Most of the critical embedded control command systems rely on such linear systems. But intervals and simple linear invariants in general will not easily allow to precisely describe their state space.

This analysis is based on ideas from control theory. They are used to prove the stability of the system by exhibiting a proof of existence of an invariant called Lyapunov quadratic form.

This work addresses the explicit computation of such a form by exploring the instantiation of multiple generic templates to find the most appropriate ellipsoids to bound the analyzed system.

Our effort also considers floating point errors and addresses the validity of the computed solution. It has been implemented and applied on several examples.

The approach of this part presents the major drawback of being unable to directly analyze actual systems at code level, in particular because such systems are usually equipped with saturations or resets. This can be addressed by using policy iteration methods. The computation of templates does not play any role in soundness of the analysis and is able to accommodate heuristics extracting potential $A$ and $B$ matrices from the code. All this is addressed in the next part of this document.

# Part III

# Guarded Linear Systems

# Chapter 9

# State of the Art – a Policy Iteration Primer

The previous part of this document focused on generating quadratic templates for pure linear systems with a transition relation of the form $x_{k+1} = Ax_k + Bu_k$. The matrices $A$ and $B$ defining such systems were assumed to be given and such work does not readily enable to define an abstract domain to analyze actual code through Kleene iterations, as introduced in Chapter 3. Moreover, actual controller code, although based on a linear core, tend to include non purely linear features such as resets, saturations, antiwindups,... The current part focuses on the use of previous part's results to design an abstract domain able to infer quadratic invariants on code implementing guarded linear systems.

Section 9.1 first introduces policy iterations. Then Section 9.2 presents the system of equations solved by policy iterations and how it is derived from the analyzed program. Finally, Section 9.3 gives a detailed overview of the two main policy iterations techniques to compute quadratic invariants on linear programs, namely min-policy (Section 9.3.1) and max-policy (Section 9.3.2) iterations.

Chapter 10 will then introduce a control flow graph abstract domain which will be used in Chapter 11 to build a policy iteration abstract domain, inferring quadratic invariants on linear programs. Chapter 12 will then deal with the floating point issues arising with these analyses before Chapter 13 details experimental results and Chapter 14 finally concludes.

## 9.1 Introduction

Classic abstract interpretation based static analysis [CC77] heavily relies on the *widening* operator, as seen in Chapter 3. This operator discards some information in order to enforce termination of the analysis. A *narrowing* can then partly recover this lost information. These heuristics often enable a good trade-off between cost and precision of analyses. However, even if impressive improvements were made in the last decade to widening [BSC12, FG10, GR06, SJ11, and references therein] and narrowing [HH12], they do not always guarantee precise results.

Another approach, that appeared in the last decade in the software verification community [Cou05, for instance], is the use of dedicated mathematical

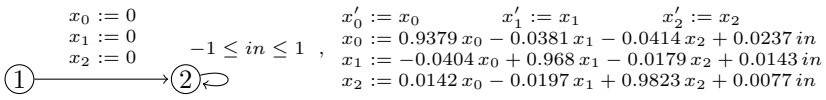



Figure 9.1 – Control flow graph for our running example.

solvers like linear or semi-definite programming as a way to solve some kind of problems in a verification setting. This led to the definition of *policy iterations* [AGG10, CGG+05, GGTZ07, GS07a, GS10, GSA+12], as another way to perform overapproximation but trying to achieve better precision than widening-based analyses.

The basic idea of policy[1] iteration is to use numerical optimization tools to compute those bounds that are hard to guess for the widening or to retrieve via narrowing.

Another advantage of the method is to abstract sequences of program instructions like loop bodies "en bloc", avoiding intermediate abstractions which can cause irreversible losses of precision[2].

This chapter will mostly focus on policy iterations with quadratic templates, using semi-definite programming as underlying mathematical solver [AGG10, GS10, GSA+12].

## 9.2 System of Equations

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

A first step in deriving those equations from the program is to build its control flow graph.

**Example 34.** *Figure 9.1 represents the control flow graph for our running example (Example 16, page 15). Vertex "1" corresponds to the starting point of the program and vertex "2" to the head of the loop. The edge between "1" and "2" reflects the assignments before the loop and the looping edge on vertex "2" represents the loop body.*

**Remark 14.** *It is worth noting that, unlike the usual notion of control flow graph with vertices between each single instruction of the program, sequences of instructions are here considered "en bloc" with graph vertices only for starting point and loop heads of the program. This will both improve the precision of the analysis and decrease its cost by avoiding useless intermediate abstractions.*

A set of expressions $t_j$ defining a template domain (c.f., Definition 30, page 30) are then chosen and a system of equations is defined with a variable $b_{i,j}$ for each vertex $i$ of the graph and each template $t_j$.

[1] The word *strategy* is also used in the literature, with equivalent meaning.
[2] This is not illustrated here, one can refer for instance to [GM11] for more details.

**Example 35.** *To bound the variables of the running example's program, we choose the quadratic template*[3]*:* $t_1 := 6.2547x_0^2 + 12.1868x_1^2 + 3.8775x_2^2 - 10.61x_0x_1 - 2.4306x_0x_2 + 2.4182x_1x_2$*. Templates* $t_2 := x_0^2$*,* $t_3 := x_1^2$ *and* $t_4 := x_2^2$ *are added in order to get bounds on each variable.*

*Here is the system of equations derived from the control flow graph of Figure 9.1 and these templates:*

$$\begin{cases} b_{1,1} = +\infty \quad b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_1) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_2) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_3) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,4} = \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_4) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \end{cases} \tag{9.1}$$

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

Each $b_{i,j}$ bounds the template $t_j$ at program point $i$ and is defined in one equation as a maximum (denoted "$\vee$") over as many terms as incoming edges in vertex $i$. More precisely, each edge between two vertices $v$ and $v'$ translates to a term in each equation $b_{v',j}$ on the pattern: $\max\left\{ r(t_j) \;\middle|\; c \wedge \bigwedge_j t_j \leq b_{v,j} \right\}$ where $c$ and $r$ are respectively the constraints and the assignments associated to this edge. This expresses the maximum value the template $t_j$ can reach in destination vertex $v'$ when applying the assignments $r$ on values satisfying both the constraints $c$ of the edge and the constraints $t_j \leq b_{v,j}$ of the initial vertex $v$. Finally, the program starting point is initialized to $(+\infty, \ldots, +\infty)$, meaning all equations for $b_{i_0,j}$, where $i_0$ is the starting point, become $b_{i_0,j} = +\infty$. Thus, for any solution $(b_{1,1}, \ldots, b_{1,n}, \ldots)$ of the equations, $\gamma_{\mathcal{T}}(b_{i,1}, \ldots, b_{i,n})$ (c.f., Definition 30, page 30) is an overapproximation of reachable states of the program at point $i$.

## 9.3 Policy Iterations

Two different techniques can be found in the literature to compute an overapproximation of the least solution of the previous system of equations (which existence is proved thanks to Knaster-Tarski theorem).

### 9.3.1 Min-Policy Iterations

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

Writing a system of equations $b = F(b)$ with $b = (b_{i,j})_{i \in [\![1,p]\!], j \in [\![1,n]\!]}$ and $F : \overline{\mathbb{R}}^{np} \to \overline{\mathbb{R}}^{np}$ ($n$ being the number of templates and $p$ the number of vertices

[3] This template was computed using one of the methods described in Chapter 5.

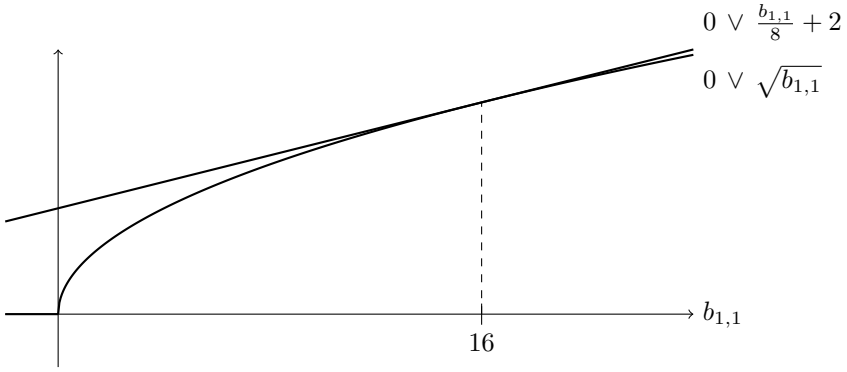



Figure 9.2 – $0 \vee \frac{b_{1,1}}{8} + 2$ is a min-policy for $0 \vee \sqrt{b_{1,1}}$.

in the control flow graph), a min-policy is defined as follows: $\underline{F}$ is a min-policy for $F$ if for every $b \in \overline{\mathbb{R}}^{np}$, $F(b) \leq \underline{F}(b)$ and there exist some $b_0 \in \overline{\mathbb{R}}^{np}$ such that $\underline{F}(b_0) = F(b_0)$.

**Example 36.** *Considering the system of (one) equation*

$$\left\{ \; b_{1,1} = 0 \vee \sqrt{b_{1,1}} \right.$$

*where $\sqrt{x}$ is defined as $-\infty$ for negative numbers $x$, $\underline{F}$ defined as $\underline{F}(b) := 0 \vee \frac{b_{1,1}}{8} + 2$ is a min-policy. Indeed, for all $b_{1,1} \in \overline{\mathbb{R}}$, $0 \leq 0$ and, $\sqrt{b_{1,1}} \leq \frac{b_{1,1}}{8} + 2$, hence $F(b) = 0 \vee \sqrt{b_{1,1}} \leq 0 \vee \frac{b_{1,1}}{8} + 2 = \underline{F}(b)$, and for $b_0 = 16$, $F(b_0) = \sqrt{16} = \frac{16}{8} + 2 = \underline{F}(b_0)$. This is illustrated on Figure 9.2.*

**Example 37.** *A min-policy of the system of equations (9.1):*

$$\left\{ \begin{array}{ll} b_{1,1} = +\infty & b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = f(b_{2,1}, b_{2,2}, b_{2,3}, b_{2,4}) & b_{2,2} = f(b_{2,1}, b_{2,2}, b_{2,3}, b_{2,4}) \\ b_{2,3} = f(b_{2,1}, b_{2,2}, b_{2,3}, b_{2,4}) & b_{2,4} = f(b_{2,1}, b_{2,2}, b_{2,3}, b_{2,4}) \end{array} \right.$$

*where $f$ is the function taking the value $0$ when $b_{2,1} = b_{2,2} = b_{2,3} = b_{2,4} = -\infty$ and the value $+\infty$ elsewhere. This is indeed larger than (9.1) for all $b \in \overline{\mathbb{R}}^8$ and there is equality for instance when $b_{1,1} = b_{1,2} = b_{1,3} = b_{1,4} = 42$ and $b_{2,1} = b_{2,2} = b_{2,3} = b_{2,4} = -\infty$ (i.e., for $b_0 = (42, 42, 42, 42, -\infty, -\infty, -\infty, -\infty)$).*

**Example 38.** *Another min-policy of the system of equations (9.1):*

$$\left\{ \begin{array}{ll} b_{1,1} = +\infty & b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0152 & b_{2,2} = 0 \vee 0.2195\, b_{2,1} + 11.0979 \\ b_{2,3} = 0 \vee 0.1143\, b_{2,1} + 4.8347 & b_{2,4} = 0 \vee 0.2669\, b_{2,1} + 3.9796, \end{array} \right.$$

*the equality being obtained for $b_0 = (+\infty, +\infty, +\infty, +\infty, 1000000, +\infty, +\infty, +\infty)$.*

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

**Theorem 3.** *Given a (potentially infinite) set $\underline{\mathcal{F}}$ of min-policies for $F$. If for all $b \in \overline{\mathbb{R}}^{np}$ there exist a policy $\underline{F} \in \underline{\mathcal{F}}$ interpolating $F$ at point $b$ (i.e. $\underline{F}(b) = F(b)$) and if each $\underline{F} \in \underline{\mathcal{F}}$ has a least fixpoint $\mu\underline{F}$, then the least fixpoint of $F$ satisfies*

$$\mu F = \bigwedge_{\underline{F} \in \underline{\mathcal{F}}} \mu \underline{F}.$$

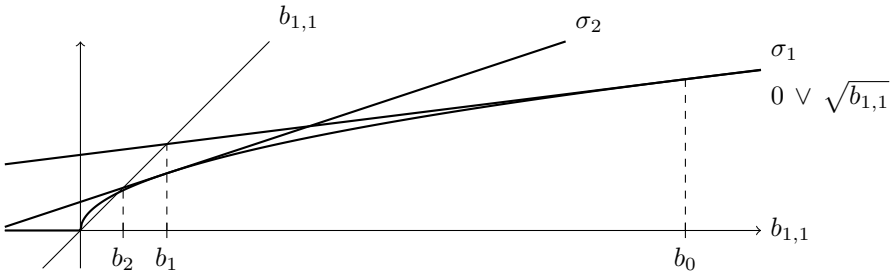



Figure 9.3 – Illustration of Example 39.

*Proof.* See comments on Equation (51) in [GSA+12]. □

**Remark 15.** *This enables to better understand the name* min*-policies since, in the hypotheses of the previous theorem, $F$ is the pointwise minimum of the min-policies $\underline{F} \in \underline{\mathcal{F}}$:*

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

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

- find a linear min-policy $\sigma_{i+1}$ being tangent to $F$ at point $b_i$, this can be done thanks to a semi-definite programming solver and an appropriate relaxation;
- compute the least fixpoint $b_{i+1}$ of policy $\sigma_{i+1}$ thanks to a linear programming solver.

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

**Example 39.** *We perform min-policy iterations on the system of equation of Example 36:*

$$\left\{ \; b_{1,1} = 0 \vee \sqrt{b_{1,1}}. \right.$$

- *We start from the postfixpoint $b_0 = 16$. This postfixpoint could be obtained through Kleene iterations for instance.*
- *For each term of the unique equation, we look for an hyperplane tangent to the term at point $b_0$. $0$ is tangent to $0$ at point $b_0$ and $\frac{b_{1,1}}{8} + 2$ is tangent to $\sqrt{b_{1,1}}$ at point $b_0$ (c.f., Figure 9.2), this gives the following linear min-policy:*
  $\sigma_1 =$

$$\left\{ \; b_{1,1} = 0 \vee \frac{b_{1,1}}{8} + 2 \right.$$

- *The least fixpoint of* $\sigma_1$ *is then:* $b_1 = \frac{16}{7} \simeq 2.2857$.
- *Looking for hyperplanes tangent at point* $b_1$ *gives the min-policy:*
  $\sigma_2 =$
  $$\left\{ \; b_{1,1} = 0 \vee \tfrac{\sqrt{7}}{8} b_{1,1} + \tfrac{2}{\sqrt{7}} \right.$$
- *Hence* $b_2 = \frac{16}{8\sqrt{7}-7} \simeq 1.1295$.

*These two first iterations are illustrated on Figure 9.3. The procedure then rapidly converges to the fixpoint* $b_{1,1} = 1$ *(the next iterates being* $b_3 \simeq 1.0035$ *and* $b_4 \simeq 1.0000$*) and can be stopped as soon as the accuracy is deemed satisfying.*

**Example 40.** *We perform min-policy iterations on the running example.*

- *We start from the postfixpoint* $b_0 = (+\infty, +\infty, +\infty, +\infty, 1000000, +\infty, +\infty, +\infty)$. *This postfixpoint could be obtained through Kleene iterations for instance.*
- *For each term of each equation, we look for an hyperplane tangent to the term at point* $b_0$. *This can be done thanks to a semi-definite programming solver and gives the following linear min-policy:*
  $\sigma_1 =$
  $$\left\{ \begin{array}{ll} b_{1,1} = +\infty & b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0152 & b_{2,2} = 0 \vee 0.2195\, b_{2,1} + 11.0979 \\ b_{2,3} = 0 \vee 0.1143\, b_{2,1} + 4.8347 & b_{2,4} = 0 \vee 0.2669\, b_{2,1} + 3.9796 \end{array} \right.$$
- *A linear programming solver allows to compute the least fixpoint of* $\sigma_1$*:* $b_1 = (+\infty, +\infty, +\infty, +\infty, 1.0664, 11.3324, 4.9568, 4.2644)$.
- $\sigma_2 =$
  $$\left\{ \begin{array}{ll} b_{1,1} = +\infty & b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0143 & b_{2,2} = 0 \vee 0.2302\, b_{2,1} + 0.0120 \\ b_{2,3} = 0 \vee 0.1190\, b_{2,1} + 0.0052 & b_{2,4} = 0 \vee 0.2708\, b_{2,1} + 0.0042 \end{array} \right.$$
- $b_2 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.2429, 0.1245, 0.2757)$.
- $\sigma_3 =$
  $$\left\{ \begin{array}{ll} b_{1,1} = +\infty & b_{1,2} = +\infty \\ b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0143 & \\ b_{2,2} = 0 \vee 0.0390\, b_{2,1} + 0.7426\, b_{2,2} + 0.0114 & \\ b_{2,3} = 0 \vee 0.0340\, b_{2,1} + 0.6635\, b_{2,3} + 0.0050 & \\ b_{2,4} = 0 \vee 0.2709\, b_{2,1} + 0.0040 & \end{array} \right.$$
- $b_3 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.1962, 0.1160, 0.2757)$.
- $\sigma_4 =$
  $$\left\{ \begin{array}{ll} b_{1,1} = +\infty & b_{1,2} = +\infty \\ b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = 0 \vee 0.9857\, b_{2,1} + 0.0143 & \\ b_{2,2} = 0 \vee 0.0194\, b_{2,1} + 0.8340\, b_{2,2} + 0.0104 & \\ b_{2,3} = 0 \vee 0.0214\, b_{2,1} + 0.7688\, b_{2,3} + 0.0049 & \\ b_{2,4} = 0 \vee 0.2709\, b_{2,1} + 0.0040 & \end{array} \right.$$

- $b_4 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.1803, 0.1137, 0.2757)$.

*Two more iterations lead to* $b_6 = (+\infty, +\infty, +\infty, +\infty, 1.0029, 0.1795, 0.1136, 0.2757)$ *which is the invariant given in Example 24 and depicted on Figure 4.6, page 30.*

**Remark 16** (Number and size of semi-definite programs)**.** *At each iteration, one semi-definite program has to be solved for each term of each equation in order to compute a new policy. This leads to many semi-definite programs but each focusing on a single term, hence rather small. The computed policies being linear are then solved through linear programming. This way, at the scale of the whole system, onyl linear programs are solved, which scales better than semi-definite programming.*

### 9.3.2 Max-Policy Iterations

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

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

**Example 41.** *A max-policy of the system of equations from Example 35:*

$$\begin{cases} b_{1,1} = +\infty \quad b_{1,2} = +\infty \quad b_{1,3} = +\infty \quad b_{1,4} = +\infty \\ b_{2,1} = \max\{r(t_1) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,4} = \max\{r(t_4) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \end{cases}$$

*From the system of equations (9.1), page 65, the first (and only) term* $(+\infty)$ *was chosen for* $b_{1,1}$*,* $b_{1,2}$*,* $b_{1,3}$ *and* $b_{1,4}$*. For* $b_{2,1}$ *and* $b_{1,4}$*, the second term is used. Finally, for* $b_{2,2}$ *and* $b_{2,3}$*, the first term was selected.*

Iterations are again done with two main objects: a max-policy $\sigma$ and a tuple $b$ of values for variables $b_{i,j}$ of the system of equations. Considering that computing a fixpoint on a given policy reduces to a mathematical optimization problem and that a fixpoint of the whole equation system is also a fixpoint of some policy, the following policy iteration algorithm aims at finding such a policy by solving optimization problems. To initiate the algorithm, a term $-\infty$ is added to each equation, the initial policy $\sigma_0$ is then $-\infty$ for each equation and the initial value $b_0$ is the tuple $(-\infty, \ldots, -\infty)$. Then policies are iterated:

- find a policy $\sigma_{i+1}$ improving policy $\sigma_i$ at point $b_i$, i.e. that reaches (strictly) greater values evaluated at point $b_i$; if none is found, exit;
- compute the greatest fixpoint $b_{i+1}$ of policy $\sigma_{i+1}$.

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

**Remark 17.** *Although min and max policies are dual concepts, we are in both cases looking for* over*approximations of the least fixpoint of the system of equations, thus the algorithms are* not *dual.*

**Example 42.** *We perform max-policy iterations on the running example. For that, we first add $-\infty$ terms to each equation, leading to the following system of equations:*

$$\begin{cases} b_{1,1} = -\infty \vee +\infty & b_{1,2} = -\infty \vee +\infty \\ b_{1,3} = -\infty \vee +\infty & b_{1,4} = -\infty \vee +\infty \\ b_{2,1} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_1) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,2} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_2) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,3} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_3) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \\ b_{2,4} = -\infty \vee \max\{0 \mid \mathrm{be}(1)\} \vee \max\{r(t_4) \mid -1 \leq in \leq 1 \wedge \mathrm{be}(2)\} \end{cases}.$$

- *We start with initial policy $\sigma_0 =$*

$$\begin{cases} b_{1,1} = -\infty & b_{1,2} = -\infty & b_{1,3} = -\infty & b_{1,4} = -\infty \\ b_{2,1} = -\infty & b_{2,2} = -\infty & b_{2,3} = -\infty & b_{2,4} = -\infty \end{cases}.$$

- *Its greatest fixpoint is $b_0 = (-\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty, -\infty)$.*

- *We now look for a policy $\sigma_1$ improving $\sigma_0$ at point $b_0$. For the first four equations, the term $+\infty$ is definitely greater than $-\infty$. The strategy $\sigma_1 =$*

$$\begin{cases} b_{1,1} = +\infty & b_{1,2} = +\infty & b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = -\infty & b_{2,2} = -\infty & b_{2,3} = -\infty & b_{2,4} = -\infty \end{cases}.$$

  *is then a suitable choice.*

- *Hence $b_1 = (+\infty, +\infty, +\infty, +\infty, -\infty, -\infty, -\infty, -\infty)$.*

- *We again look for a policy $\sigma_2$ improving $\sigma_1$ at point $b_0$. There is nothing strictly greater than $+\infty$ in $\overline{\mathbb{R}}$ and we keep the $+\infty$ terms for the first four equations. In the four remaining equations, replacing the $b_{i,j}$ with values from $b_1$ in* $\mathrm{be}(1)$ *and* $\mathrm{be}(2)$ *respectively gives formula equivalent to true and false. This way, for these four equations, the first term reduces to 0 whereas the second term evaluates to $-\infty$. 0 being greater than the $-\infty$ from $b_1$, we get an improving strategy $\sigma_2 =$*

$$\begin{cases} b_{1,1} = +\infty & b_{1,2} = +\infty \\ b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,2} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,3} = \max\{0 \mid \mathrm{be}(1)\} \\ b_{2,4} = \max\{0 \mid \mathrm{be}(1)\} \end{cases}.$$

- $b_2 = (+\infty, +\infty, +\infty, +\infty, 0, 0, 0, 0)$.

- *Now that the $b_{2,j}$ in $b_2$ are no longer $-\infty$,* $\mathrm{be}(2)$ *is no longer false and it becomes interesting to select the second terms in the four last equations,*

*hence* $\sigma_3 =$

$$\begin{cases} b_{1,1} = +\infty & b_{1,2} = +\infty \\ b_{1,3} = +\infty & b_{1,4} = +\infty \\ b_{2,1} = \max\{r(t_1) \mid -1 \leq in \leq 1 \wedge \text{be}(2)\} \\ b_{2,2} = \max\{r(t_2) \mid -1 \leq in \leq 1 \wedge \text{be}(2)\} \\ b_{2,3} = \max\{r(t_3) \mid -1 \leq in \leq 1 \wedge \text{be}(2)\} \\ b_{2,4} = \max\{r(t_4) \mid -1 \leq in \leq 1 \wedge \text{be}(2)\} \end{cases}.$$

- *The greatest fixpoint* $b_3 = (+\infty, +\infty, +\infty, +\infty,\ 1.0077,\ 0.1801,\ 0.1141,\ 0.2771)$ *of* $\sigma_3$ *can be computed thanks to a semi-definite programming solver and an appropriate relaxation.*
- *No more improving strategy.*

*After four iterations, the algorithm has found the same least fixpoint than min policies in Example 40.*

**Remark 18** (Number and size of semi-definite programs)**.** *In contrary to min-policies (c.f., Remark 16), max-policies are not linear. Solving them then requires semi-definite programs whereas min-policies only solves linear programs at the scale of the whole system.*

The Max-Policy iterations build an ascending chain of abstract elements similarly to Kleene iterations elements. However it is guaranteed to be finite, bounded by the number of policies $\sigma$, while Kleene iterations require the use of widening to ensure termination. Since there are exponentially many max-policies in the number of templates and points of the control flow graph and since each policy can be an improving one only once, we have an exponential bound on the number of iterations. However, in practice, only a small number of policies are usually considered and the number of iterations remains reasonable.

# Chapter 10

# A Control Flow Graph Abstract Domain

The previous chapter demonstrated how policy iterations can be used to compute fixpoints which would be difficult to obtain using classic Kleene iterations with widening. However, even if promising, policy iterations have had very little impact on existing tools yet: their use seems orthogonal to the classic use of abstract domains with Kleene iterations, where reduced products allow domains to exchange knowledge about the system during computation.

An explanation to the lack of integration of policy iterations with Kleene-based abstract domains is that they need to work on a global view of the analyzed system, typically a control flow graph representation of it, while Kleene-based analyzers iterate through program points without providing a global view of the program to the abstract domains. Although simply running both kind of analyses in parallel is easy, that would hinder the chances of a tight cooperation between them. The contribution of our work is to define an abstract domain which will gracefully interface both worlds. Our solution is mainly to design a symbolic abstract domain recording the control flow graph while iterating through the program points with a Kleene-based analysis. Once the graph is obtained it can easily be used by policy iterations to compute numerical properties about the program. Those properties can finally be exported to other domains through a reduced product. Moreover, this new abstract domain can be applied on a strict subset of program variables, abstracting other variables by information obtained through reduced product from other domains during the analysis[1], thus allowing a true interplay between policy iterations and existing abstract domains.

This chapter describes a symbolic abstract domain reconstructing control flow graphs similar to Figure 9.1, page 64. This domain will basically "record" assignments and guards in graph edges thanks to the corresponding abstract operators and close loops during widenings. First, Section 10.1 first gives an intuition of basic ideas of the domain on an example. The following sections develop formal definitions and proof of correctness of the lattice structure of the domain 10.2, its abstract operators 10.3 and widening 10.4. Finally, Section 10.5 contains a few last examples and remarks.

This domain will finally be used in the next chapter to provide an embedding

[1] As done with expensive relational domains in the abstract interpreter Astrée [CCF+09].

```
0 x := 0;1 y := 1;
while2 y ≤ 2 do
  3 x := x+3;
  4 if x ≤ 4 then
    5 y := x+5;
  else
    6 y := x+6;
  fi;
  7 x := x+7;
od8
```

Figure 10.1 – Example of program.

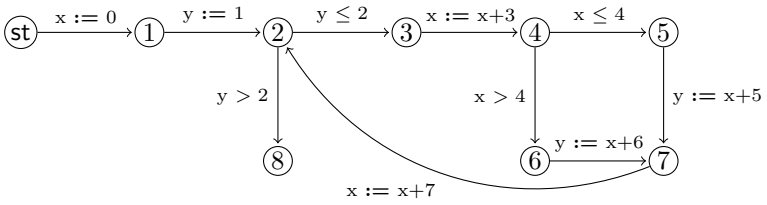



Figure 10.2 – Control flow graph for program of Figure 10.1.

of policy iterations in a classic abstract domain for Kleene iterations. These two chapter are mostly an extended version of a conference paper [RG13a].

## 10.1 Introductory Example

This introductory section aims at giving some intuition of the basic concepts of the abstract domain formally defined in the remaining of the chapter by illustrating them on the example program of Figure 10.1. On this figure, subscripts $0, \ldots, 8$ mark program points which will be referred upon throughout the discussion.

### Recording Classic Control Flow Graphs

We first show how an abstract domain can along Kleene iterations record the control flow graph of Figure 10.2. As usual, Kleene iterations start at program point *0* with the abstract value $\top$. Then the assignment "x := 0" is considered leading, at program point *1*, to an abstract value represented as a graph with two vertices st and 1 and an edge between them labeled with the assignment "x := 0". This is displayed on Figure 10.3. st is a special vertex identifying the starting point of the program. Moreover, we also need to keep track of which vertex corresponds to the program point the abstract value is associated with, in this case the vertex 1. This particular vertex is marked by a double circle on the figures and will be called *current point* in the following. The concretization of an abstract value is then the reachable state space at the doubly circled vertex in the graph[2].

[2]The concretization of the abstract value of Figure 10.3b is then $\{\rho : \mathbb{V} \to \mathbb{R} \mid \rho(x) = 0\}$.

Then, Kleene iterations proceed from program point *1* to point *2* by taking into account the assignment "y := 1". To that end, a new vertex 2 is added in the graph[3] with an edge from the current point to it labeled with the assignment. Finally, the current point is moved to the new vertex 2, as displayed on Figure 10.4.

Since we are at a loop head, a widening is performed. Entering the loop for the first time, the previous iterate at this point was $\bot$ and the widening leaves the abstract value unchanged ($\bot \nabla (c) = (c)$). Going on with Kleene iterations between program points *2* and *3*, the guard "y $\leq$ 2" is considered. As for assignments, a new vertex 3 is added, along with an edge to it, labeled with the guard, and the current point is moved to 3. Then the assignment "x := x+3" is processed, as shown on Figure 10.5.

The guard "x $\leq$ 4" then leads to program point *5* and the guard "x $>$ 4" to program point *6*. This can be seen[4] respectively on Figures 10.6 and 10.7.

Program point *7* is reachable either from program point *5* through assignment "y := x+5" or from program point *6* through assignment "y := x+6". The two assignments are then applied to the abstract values obtained respectively at program points *5* and *6* before merging the results thanks to the join operator $\sqcup^\sharp$. This operator simply returns the union of the vertices (and edges) of its two operands (i.e., vertices with the same name in both operands are merged into a single one in the result), except for the current points which are merged (even if they had different names in the two operands). All this is displayed on Figure 10.8.

Kleene iterations are then back at program point *2* through the assignment "x := x+7". This point being a widening point, a widening is performed with the abstract value (c) obtained at the previous iteration, which basically amounts to a join[5] (c.f., Figure 10.9). The resulting abstract value being different from the one at the previous loop iteration ($(c) \neq (l)$), further iterations are required to reach a fixpoint.

Kleene iterations then proceed to program point *3* through the guard "y $\leq$ 2". This means we should add a new vertex and an edge to it from the current point (vertex 2), labeled with the guard. However, vertex 2 already has an outgoing edge labeled "y $\leq$ 2" so we just move the current point to the destination of this edge (vertex 3, here), as shown on Figure 10.10. Intuitively, the new current point already existed since we already iterated through this code.

Figures 10.11 and 10.12 display the following steps up to program point *7* going on similarly, only moving the current point along already existing edges.

Figure 10.13 shows the widening performed at point *2*. This time, the result is identical to the result (l) of the previous widening, a fixpoint is reached and it is no longer needed to iterate through the loop.

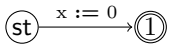
Finally, the result of Kleene iterations on our example is obtained at the last program point (point *8*), by going through the guard "y $>$ 2", and is displayed on Figure 10.14. It is worth noting that this is indeed the graph of Figure 10.2.

[3] For the sake of readability, name of vertices are chosen to match program point numbers. There is otherwise no particular reason to call 2 the vertex corresponding to program point *2*.

[4] We assume a function `new_node_name` giving a different result each time it is called so that vertices 5 and 6 have different names. Giving them the same name (calling both of them 5 for instance) would remain sound but would end up in very strange and unexpected overapproximations by adding spurious paths in the control flow graph.

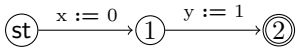
[5] Minor difference: when merging two current points, the name of the first one is kept.

(a) $\top$



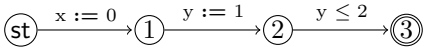

(b) $[\![x := 0]\!]^\sharp$(a)

Figure 10.3 – First step of Kleene iterations on our example.

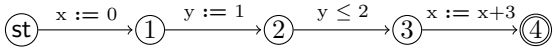



(c) $[\![y := 1]\!]^\sharp$(b)

Figure 10.4 – Second step of Kleene iterations.

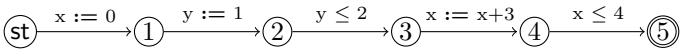



(d) $[\![y \leq 2]\!]^\sharp$(c)



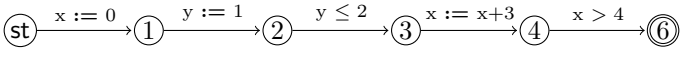

(e) $[\![x := x + 3]\!]^\sharp$(d)

Figure 10.5 – Next steps.



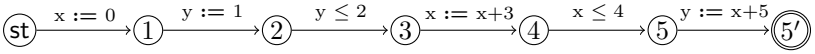

(f) $[\![x \leq 4]\!]^\sharp$(e)

Figure 10.6 – Next step between program points *4* and *5*.

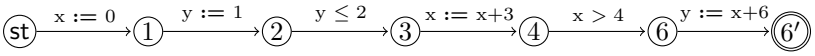



(g) $[\![x > 4]\!]^\sharp$(e)

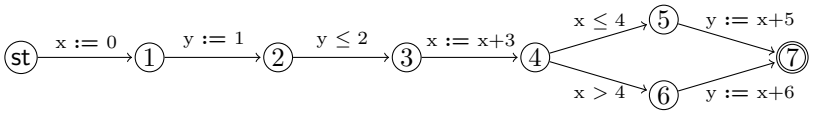
Figure 10.7 – Next step between program points *4* and *6*.




(h) $[\![y := x + 5]\!]^\sharp$(f)

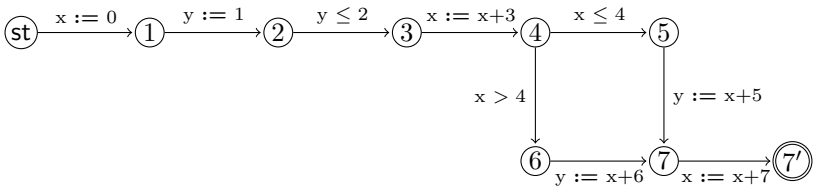



(i) $[\![y := x + 6]\!]^\sharp$(g)

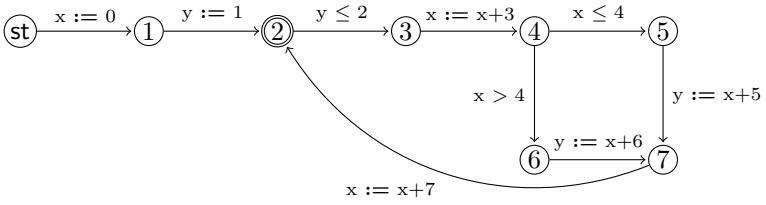



(j) (h) $\sqcup^\sharp$ (i)

Figure 10.8 – Next step, to program point *7*.

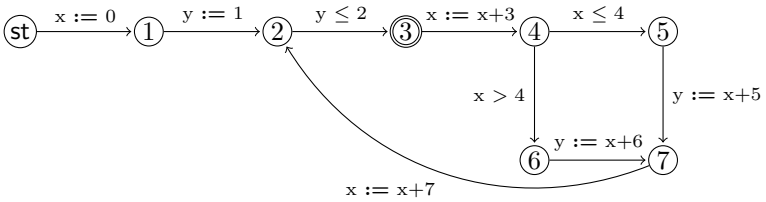



(k) $[\![x := x + 7]\!]^\sharp$(j)



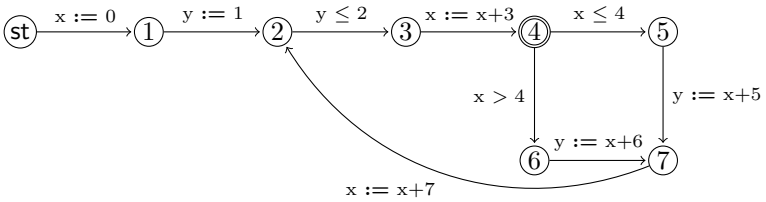

(l) (c) $\nabla$ (k)

Figure 10.9 – Next step, to program point *2*.



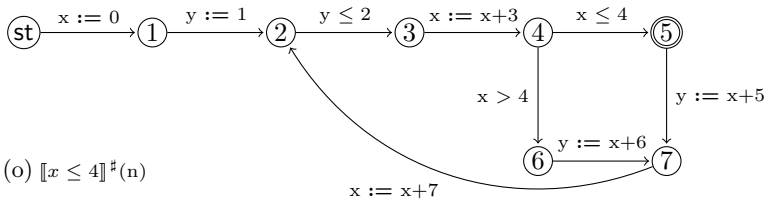

(m) $[\![y \leq 2]\!]^\sharp$(l)

Figure 10.10 – Next step, to program point *3*.

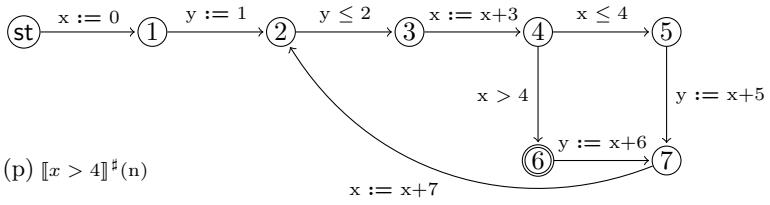



(n) $[\![x := x + 3]\!]^\sharp$(m)

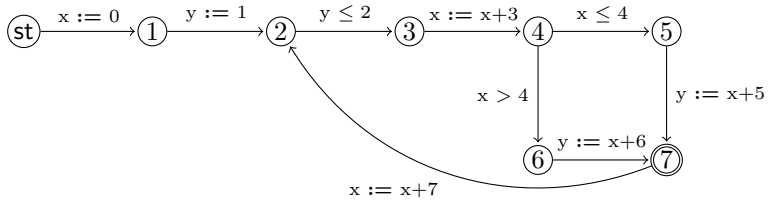
Figure 10.11 – Next step, to program point *4*.




(o) $[\![x \leq 4]\!]^\sharp(\mathrm{n})$

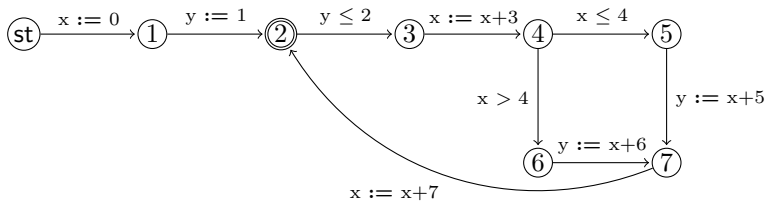



(p) $[\![x > 4]\!]^\sharp(\mathrm{n})$



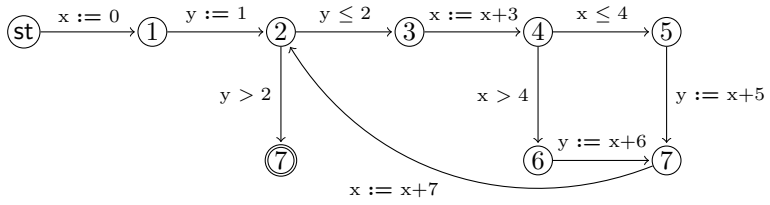

(q) $[\![y := x + 5]\!]^\sharp(\mathrm{o}) \sqcup^\sharp [\![y := x + 6]\!]^\sharp(\mathrm{p})$ $\left(= [\![y := x + 5]\!]^\sharp(\mathrm{o}) = [\![y := x + 6]\!]^\sharp(\mathrm{p})\right)$

Figure 10.12 – Next steps, up to program point 7.




(r) $(\mathrm{l}) \nabla [\![x := x + 7]\!]^\sharp(\mathrm{q})$ $\left(= (\mathrm{l}) = [\![x := x + 7]\!]^\sharp(\mathrm{q})\right)$

Figure 10.13 – Fixpoint reached!




(s) $[\![y > 2]\!]^\sharp(\mathrm{r})$

Figure 10.14 – Final result of Kleene iterations on our example.

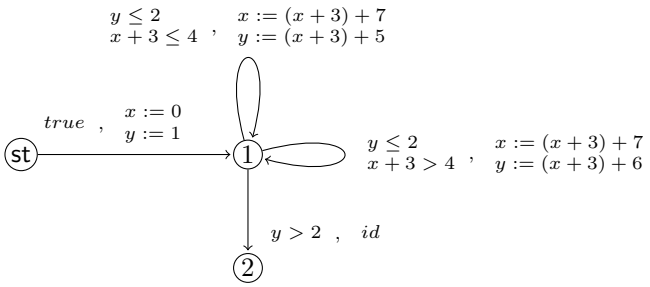



Figure 10.15 – Compact control flow graph for program of Figure 10.1, page 74.

### Recording Compact Control Flow Graphs

As noticed in Remark 14, page 64, it is preferable to apply policy iterations on compact graphs with vertices only for loop heads (plus start and ending points of the program) and edges bearing entire blocks of statements. Such a control flow graph can be seen on Figure 10.15 for our running example (program of Figure 10.1, page 74). Each edge of this graph is labeled with a pair $(c, a)$ meaning that the execution can go through the edge when the constraint $c$ is satisfied, in which case the assignments $a$ are applied. For instance, in Figure 10.1, the edge above the vertex 1 means that on vertex 1, execution can proceed to this same vertex when $y \leq 2$ and $x + 3 \leq 4$ and, if it does, the variable $x$ receives the value $x + 3 + 7$ and the variable $y$ takes the value $x + 3 + 5$. Constraint *true* stands for the constraint that is always true (i.e., no constraint) and assignment *id* denotes the identity function (i.e., no assignment, values of all variables are unchanged).

Whereas we could first record the classic control flow graph as seen above then transform it into its compact form, we can as well build the compact form on the fly. We will now show this on our running example. The basic principles remain the same than previously, except that we do not want to add a new vertex each time a new program point is visited by the Kleene analysis. To do this, we add a new special vertex fi which will be used as current point as long as nothing indicates the vertex should be kept in the final result (i.e., the vertex is a loop head). Instead of adding a new vertex and an edge to it, abstract operators for guard and assignment will just put their guard or assignment in all incoming edges of fi.

As usual, Kleene iteration start with the abstract value $\top$. Then, the assignment "x := 0" is applied, which gives a graph with two vertices st and fi and an edge between them, labeled with the constraint *true* and the assignment $x := 0$ (c.f., Figure 10.16). Since nothing indicates that the current point would be a loop head, we use the special vertex fi to convey the meaning that the current point is not intended to be kept during the following steps.

Since the current point is fi, considering the assignment "y := 1" does not add any new node but modify the incoming edge of fi to add the assignment in its label (c.f., Figure 10.17).

Program point *2* being a loop head, a widening is applied with the previous iterate at this point (entering the loop for the first time, this is $\bot$). Widening is interpreted by our domain as a signal for loop heads. So the current point being a loop head should be kept in the final result. To record this, the current point

is given a new name (other than fi). This is shown in Figure 10.18.

**Remark 19.** *Since for most abstract domains $\bot \nabla x = x$, it is common to drop this first widening. In our case, this would still work and would just produce a larger graph with the first loop iteration unrolled.*

Kleene iterations proceed to point *3* through the guard "y $\leq$ 2". This time, the current point is not fi and a new vertex is added with an edge to it labeled with the guard. Since nothing indicates that the new current point would be a loop head, we use fi (c.f., Figure 10.19).

Going to point *4*, the assignment "x := x+3" is considered. Since the current point is fi, no new node is added and the assignment takes place in the label of the incoming edge of fi. This is displayed on Figure 10.20.

Iterations then go on in a similar way to point *5*, as can be seen on Figure 10.21. Here, it is worth noting that, although the guard considered is "x $\leq$ 4", the variable $x$ experienced the assignment "x := x+3" along the incoming edge of fi. Constraints of edges being evaluated before their assignments (for an edge $(c, a)$, execution can take the edge and apply assignments $a$ only when the constraints $c$ are satisfied), we have to take this into account and label the edge with the constraint $x + 3 \leq 4$ rather than $x \leq 4$.

The guard "x $>$ 4" is considered between points *4* and *6* (c.f., Figure 10.22) and the abstract value at point *7* is computed using the join operator (which just performs the union of vertex and edges, c.f., Figure 10.23). It is worth noting on Figure 10.23i that, assignments on a same edge of the compact graph being performed simultaneously, the sequence "x := x+3; y := x+5" translates into the assignments $x := x + 3$ and $y := (x + 3) + 5$.

Program point *2* is then reached through the assignment "x := x+7" and, being at a widening point, the widening operator is applied with the result (d) of previous iteration. Similarly to what was done previously for classic graphs, this basically amounts to a join operator (union of vertices and edges, except for current points which are merged). Both operations can be seen on Figure 10.24.

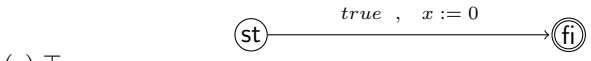
Kleene iterations proceed to program point *3* with the guard "y $\leq$ 2". Since the current point is not fi, we add a new vertex (at the same point with the classic graphs, we just had to move the current point along an edge labeled $y \leq 2$, but none of the outgoing edges of the current point is labeled $(y \leq 2, id)$ here, so we actually have to add a new vertex). Since nothing indicates it would be a loop head, we choose fi as new vertex, as shown on Figure 10.25.

The current point being now fi, following steps up to point *7* proceed by modifying the incoming edges of fi (c.f., Figure 10.26).

Finally, a last widening enables to reach a fixpoint (c.f., Figure 10.27) and program point *8* is reached through the guard "y $>$ 2" (c.f., Figure 10.28).

The graph on Figure 10.28 is indeed the same as on Figure 10.15.

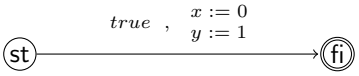
**Remark 20.** *When no reduction with another domain is performed, Klene iterations always reach a fixpoint after only two iterations, as in this introductory example. This may no longer be the case when additional informations are learned through reduced product from another domain (for instance, an interval domain may give increasing bounds on some variables until it reaches a fixpoint).*




(a) $\top$

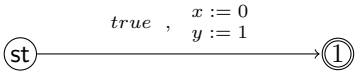
(b) $[\![x := 0]\!]^\sharp$(a)

Figure 10.16 – First step of Kleene iterations on our example.



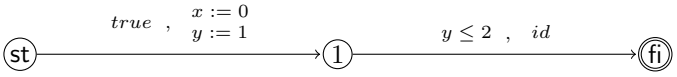

(c) $[\![y := 1]\!]^\sharp$(b)

Figure 10.17 – Next step of Kleene iterations.



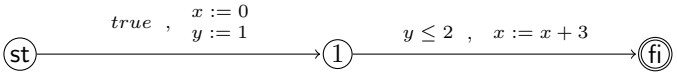

(d) $\perp \nabla$ (c)

Figure 10.18 – First widening.



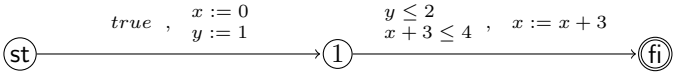

(e) $[\![y \leq 2]\!]^\sharp$(d)

Figure 10.19 – Next step, to program point *3*.



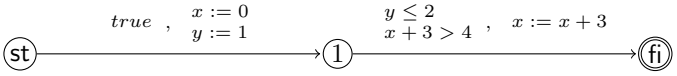

(f) $[\![x := x + 3]\!]^\sharp$(e)

Figure 10.20 – Next step, to program point *4*.



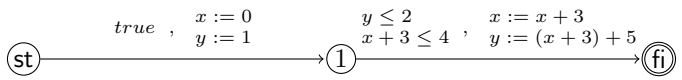

(g) $[\![x \leq 4]\!]^\sharp$(f)

Figure 10.21 – Next step, between program points *4* and *5*.

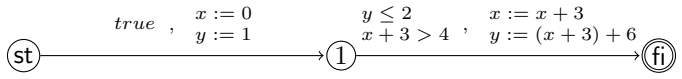



(h) $[\![x > 4]\!]^\sharp$(f)

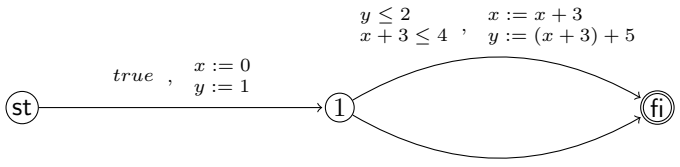
Figure 10.22 – Next step, between program points *4* and *6*.




(i) $[\![y := x + 5]\!]^\sharp$(g)

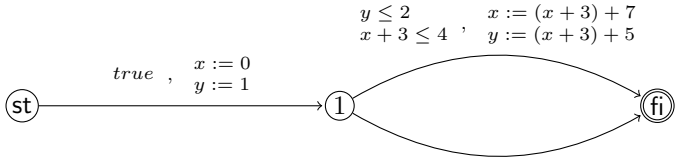



(j) $[\![y := x + 6]\!]^\sharp$(h)

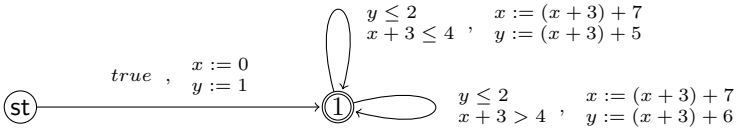



(k) (i) $\sqcup^\sharp$ (j)

Figure 10.23 – Next step, to program point *7*.

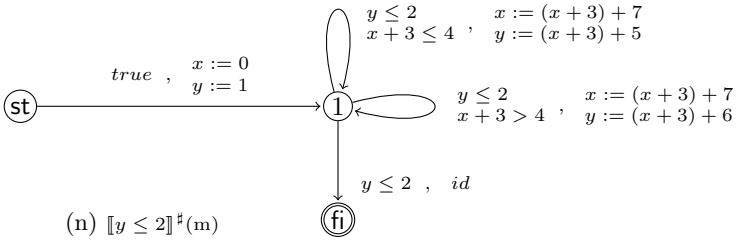



(l) $[\![x := x + 7]\!]^\sharp$(k)



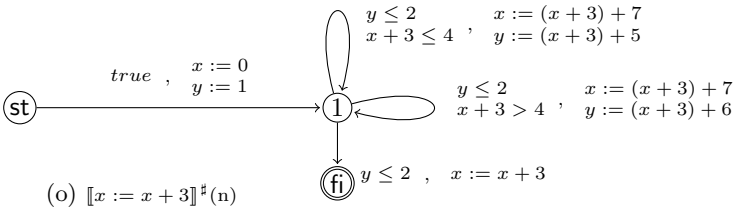

(m) (d) $\nabla$ (l)

Figure 10.24 – Next step, to program point *2*.

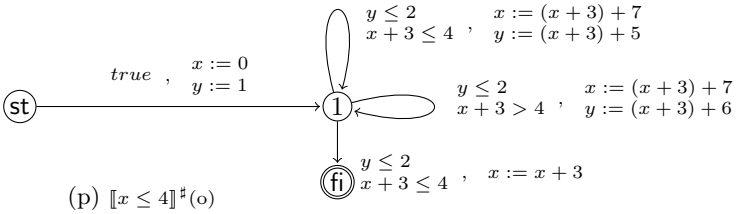



(n) $[\![y \leq 2]\!]^\sharp$(m)

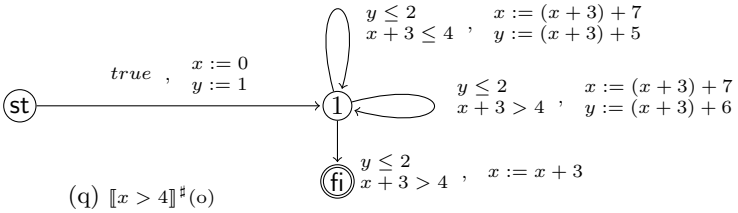
Figure 10.25 – Next step, to program point *3*.



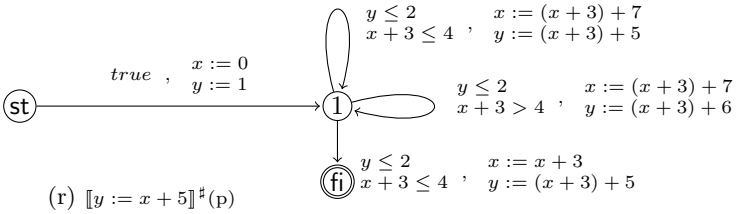

(o) $[\![x := x+3]\!]^\sharp$(n)



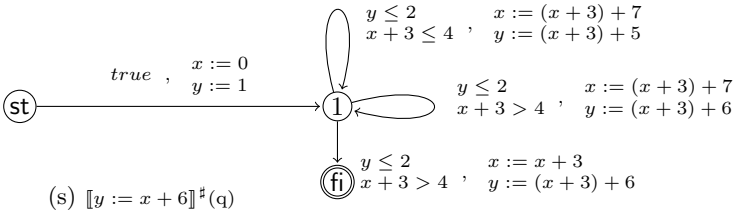

(p) $[\![x \leq 4]\!]^\sharp$(o)



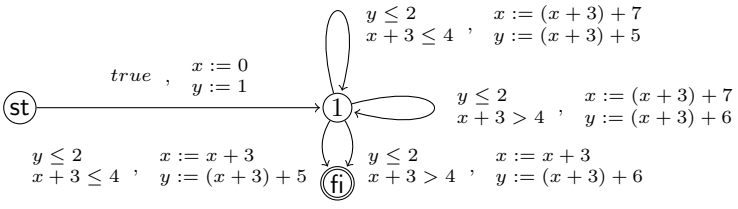

(q) $[\![x > 4]\!]^\sharp$(o)




(r) $[\![y := x+5]\!]^\sharp$(p)

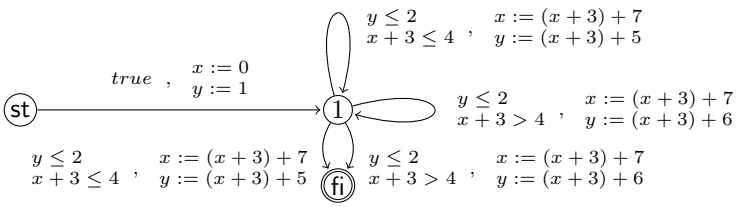



(s) $[\![y := x+6]\!]^\sharp$(q)

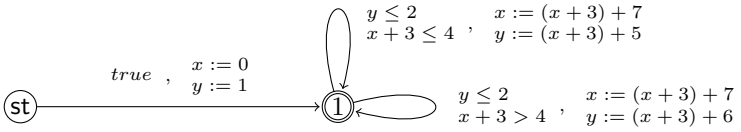



(t) (r) $\sqcup^\sharp$ (s)

Figure 10.26 – Next steps, up to program point 7.

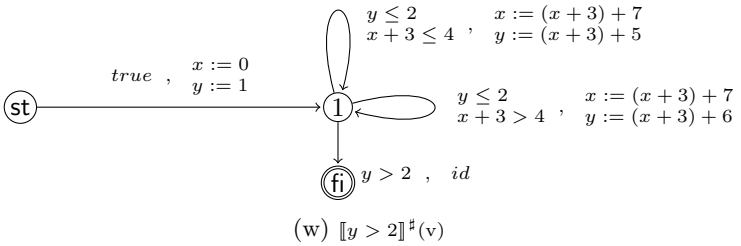



(u) $[\![x := x + 7]\!]^\sharp$(t)




(v) (m) $\nabla$ (u)

Figure 10.27 – Fixpoint reached!




(w) $[\![y > 2]\!]^\sharp$(v)

Figure 10.28 – Final result of Kleene iterations on our example.

## 10.2 Lattice Structure

This section now formally defines the lattice structure of the abstract domain informally exemplified above. The definition of a concretization function also allows to give a concrete meaning to the abstract values.

**Definition 32.** *Given a set $\mathbb{V}_{ex}$ of additional variables ($\mathbb{V}_{ex} \cap \mathbb{V} = \emptyset$), we define:*

- $\mathcal{A} := \mathbb{V} \to expr$*, functions from variables to expressions on* $\mathbb{V} \cup \mathbb{V}_{ex}$*;*
- $\mathcal{C} := expr \to \overline{\mathbb{R}}$*.*

**Example 43.** *The function $a \in \mathcal{A}$ such that $a(x) = (x+3)+7$, $a(y) = (x+3)+5$ and $a(v) = v$ for any other variable $v \in \mathbb{V}$ encodes the assignments*

$$\begin{array}{l} x := (x+3)+7 \\ y := (x+3)+5. \end{array}$$

*The function $c \in \mathcal{C}$ such that $c(y) = 2$, $c(x+3) = 4$ and $c(e) = +\infty$ for any other expression $e$ denotes the constraints*

$$\begin{array}{l} y \leq 2 \\ x+3 \leq 4. \end{array}$$

*As an other example, $c \in \mathcal{C}$ with $c(x) = 2$, $c(-x) = -1$, $c(y) = 42$ and $c(e) = +\infty$ for any other expression $e$ means*

$$\begin{array}{l} x \leq 2 \\ -x \leq -1 \\ y \leq 42 \end{array}$$

*or, equivalently,* $1 \leq x \leq 2 \wedge y \leq 42$*.*

*id* will denote the identity function, mapping every variable $x \in \mathbb{V}$ to the expression $x$ and *true* will denote the function in $\mathcal{C}$ mapping every expression to $+\infty$ (i.e., a constraint always satisfied). Variables $\mathbb{V}_{ex}$ will be used to model random assignments.

**Example 44.** *The random assignment "*x **:= ?**(1, 2)*" will be recorded with the assignment $x := z$ and the constraint $1 \leq z \leq 2$ with $z$ a fresh variable in $\mathbb{V}_{ex}$.*

**Definition 33.** *The order $\sqsubseteq^\sharp_{\mathcal{C}}$ on $\mathcal{C}$ is defined as the pointwise extension of the usual order on $\overline{\mathbb{R}}$: $c \sqsubseteq^\sharp_{\mathcal{C}} c'$ if for every expression $e$, $c(e) \leq c'(e)$.*

*The least upper bound $\bigsqcup^\sharp_{\mathcal{C}}$ on $\mathcal{C}$ is the pointwise extension of* max *on $\overline{\mathbb{R}}$: for a set of constraints $C \subseteq \mathcal{C}$, $\bigsqcup^\sharp_{\mathcal{C}} C$ is the function mapping each expression $e$ to* $\max\{c(e) \mid c \in C\}$*.*

*$\bot_{\mathcal{C}} = \bigsqcup^\sharp_{\mathcal{C}} \emptyset$ is the function in $\mathcal{C}$ mapping every expression to $-\infty$.*

**Remark 21.** *$c \sqsubseteq^\sharp_{\mathcal{C}} c'$ means that the constraint $c'$ is looser than the constraint $c$ in that satisfaction of $c$ implies satisfaction of $c'$.*

*The least upper bound $\bigsqcup^\sharp_{\mathcal{C}}$ is a (strict) overapproximation of the logical disjunction (for instance, $\bigsqcup^\sharp_{\mathcal{C}} \{ (x \leq 1 \wedge y \leq 2), (x \leq 2 \wedge y \leq 1) \} = x \leq 2 \wedge y \leq 2$ is implied by the disjunction $(x \leq 1 \wedge y \leq 2) \vee (x \leq 2 \wedge y \leq 1)$ but the reverse implication does not hold).*

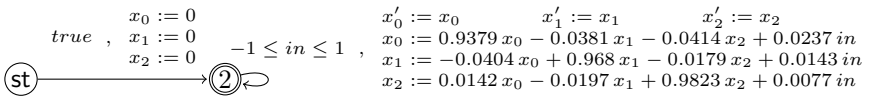



Figure 10.29 – Example of value in $\mathcal{G}$.

**Definition 34.** *Given a set $V$, $\mathsf{st} \in V$ and $\mathsf{fi} \in V$ ($\mathsf{fi} \neq \mathsf{st}$) and denoting $E$ the functions $V \times \mathcal{A} \times V \to \mathcal{C}$, we define the set of graphs $\mathcal{G}$ as:*

$$\mathcal{G} := \{ \top_{\mathcal{G}} \} \cup \left\{ (e,t) \in E \times V \;\middle|\; \begin{array}{l} t \neq st \wedge \forall v \in V, \forall a \in \mathcal{A}, e(\mathsf{fi}, a, v) = \bot_{\mathcal{C}} \\ \wedge\, t \neq \mathsf{fi} \Rightarrow \forall v \in V, \forall a \in \mathcal{A}, e(v, a, \mathsf{fi}) = \bot_{\mathcal{C}} \end{array} \right\}.$$

An element of $\mathcal{G}$ (other than $\top_{\mathcal{G}}$) is a pair $(e,t)$ with $e$ the edges of a graph and $t$ a vertex of this graph. $t$ indicates the point of program currently considered by the Kleene iterations (called *current point* in the running example of the previous section), acting like a kind of code pointer on $e$. The graph $e$ associates to each pair of points $v \in V$ and $v' \in V$ and assignment $a \in \mathcal{A}$ the constraint $e(v,a,v') \in \mathcal{C}$ to satisfy to take this transition and apply assignment $a$. $e(v,a,v') = \bot\mathcal{C}$ is a transition with a constraint that is never satisfied, it is then used to encode the absence of edge between vertices $v$ and $v'$ labeled with the assignment $a$. This way, if for all $a \in \mathcal{A}$, $e(v,a,v') = \bot_{\mathcal{C}}$, there is no edge between vertices $v$ and $v'$.

Among the graph vertices $V$ we distinguish two special vertices: $\mathsf{st}$ is the starting point of the program whereas $\mathsf{fi}$ will be used as temporary final point. We require two things about $\mathsf{fi}$: that it has no outgoing edge ($\forall v \in V, \forall a \in \mathcal{A}, e(\mathsf{fi}, a, v) = \bot_{\mathcal{C}}$) and that it is used only as current point (if $t \neq \mathsf{fi}$ then $\mathsf{fi}$ has no incoming edge: $t \neq \mathsf{fi} \Rightarrow \forall v \in V, \forall a \in \mathcal{A}, e(v, a, \mathsf{fi}) = \bot_{\mathcal{C}}$, hence $\mathsf{fi}$ is not connected to anything).

**Example 45.** *For the running example of the previous chapter, the result of Kleene iterations at loop head is the pair $(e,t)$ where $t$ is the vertex 2 and $e \in E$ is the function*

$$v, a, v' \mapsto \begin{cases} true & \text{if } (v,a,v') = (\mathsf{st}, a_1, 2) \\ -1 \leq in \leq 1 & \text{if } (v,a,v') = (2, a_2, 2) \\ \bot_{\mathcal{C}} & \text{otherwise} \end{cases}$$

*with $a_1$ the assignment*

$$\begin{array}{l} x_0 := 0 \\ x_1 := 0 \\ x_2 := 0 \end{array}$$

*and $a_2$ the assignment*

$$\begin{array}{l} x'_0 := x_0 \quad x'_1 := x_1 \quad x'_2 := x_2 \\ x_0 := 0.9379\,x_0 - 0.0381\,x_1 - 0.0414\,x_2 + 0.0237\,in \\ x_1 := -0.0404\,x_0 + 0.968\,x_1 - 0.0179\,x_2 + 0.0143\,in \\ x_2 := 0.0142\,x_0 - 0.0197\,x_1 + 0.9823\,x_2 + 0.0077\,in. \end{array}$$

*This can can be represented as on Figure 10.29. We chose a graphical representation for edges $e$, drawing only edges different from $\bot_{\mathcal{C}}$, while current point $t$ is represented by a doubly circled vertex. More precisely, we draw an edge labeled $(c,a)$ between $v$ and $v'$ when $e(v,a,v') = c \neq \bot_{\mathcal{C}}$. Furthermore, since only states reachable from the starting point are of interest in a control flow graph, only the connected component containing $\mathsf{st}$ is displayed ($\mathsf{fi}$ is not displayed for instance).*


Figure 10.30 – Example for order $\sqsubseteq^\sharp_{\mathcal{G}}$.

$\bot_{\mathcal{G}}$ will denote the graph $\left(\dot{\bot}_{\mathcal{C}}, \mathsf{fi}\right)$ where $\dot{\bot}_{\mathcal{C}} \in E$ is the constant function mapping every $(v, a, v') \in V \times \mathcal{A} \times V$ to $\bot_{\mathcal{C}}$.

**Remark 22** (A graph is either $\top_{\mathcal{G}}$ or of the form $(e, t)$)**.** *According to Definition 34, any graph $g \in \mathcal{G}$ is either $\top_{\mathcal{G}}$ or of the form $(e, t)$ with $e \in E$ and $t \in V$ (contrary to $\top_{\mathcal{G}}$, $\bot_{\mathcal{G}}$ is a particular case of the form $(e, t)$ with $e = \dot{\bot}_{\mathcal{C}}$ and $v = \mathsf{fi}$).*

An order $\sqsubseteq^\sharp_{\mathcal{G}}$ on $\mathcal{G}$ is basically defined as the pointwise extension on $E$ of $\sqsubseteq^\sharp_{\mathcal{C}}$ for values with the same current point $t$. Intuitively, a graph $(e, t)$ is smaller than another graph $(e', t)$ when the constraint to take any edge between two vertices $v$ and $v'$ with a same assignment $a$ are looser in $e'$ than in $e$ (i.e., $e(v, a, v') \sqsubseteq^\sharp_{\mathcal{C}} e'(v, a, v')$). This way, if execution can go through an edge in $e$, it can also go through it in $e'$. Thus any execution path leading to $t$ in $e$ can also lead to $t$ in $e'$ and the set of reachable values at point $t$ in $e'$ is larger than in $e$.

**Definition 35.** *The order $\sqsubseteq^\sharp_{\mathcal{G}}$ on $\mathcal{G}$ is defined as $x \sqsubseteq^\sharp_{\mathcal{G}} y$ when:*

- *$x = \bot_{\mathcal{G}}$ or $y = \top_{\mathcal{G}}$;*
- *or $x = (e, t)$ and $y = (e', t')$ and:*
  - *either $t = t'$ and $e \sqsubseteq^\sharp_E e'$;*
  - *or $t = \mathsf{fi}$ and $t' \neq \mathsf{fi}$ and $\mathrm{loop}(e, t') \sqsubseteq^\sharp_E e'$.*

*Where $\sqsubseteq^\sharp_E$ is the pointwise extension of $\sqsubseteq^\sharp_{\mathcal{C}}$ (i.e., the order on $E$ defined as $e \sqsubseteq^\sharp_E e'$ if for all $(v, a, v') \in V \times \mathcal{A} \times V$, $e(v, a, v') \sqsubseteq^\sharp_{\mathcal{C}} e'(v, a, v')$) and $\mathrm{loop}(e, t) \in E$ is defined as the following function:*

$$v, a, v' \mapsto \begin{cases} \bot_{\mathcal{C}} & \text{if } v' = \mathsf{fi} \\ \bigsqcup^\sharp_{\mathcal{C}} \{\, e(v, a, v'), e(v, a, \mathsf{fi}) \,\} & \text{if } v' = t \\ e(v, a, v') & \text{otherwise.} \end{cases}$$

The last case in the definition of $\sqsubseteq^\sharp_{\mathcal{G}}$ is more surprising than the previous ones. It basically preprocesses the first operand $x$ by redirecting all incoming edges of $\mathsf{fi}$ to the "current point" $t'$ of $y$, i.e. a loop is greater for the order than the same edge going to $\mathsf{fi}$. This is illustrated in Figure 10.30 and will be required later for the widening to be able to create loops in the graph.

**Lemma 5.** *$\sqsubseteq^\sharp_E$ is an order on $E$.*

*Proof.* $\sqsubseteq^\sharp_E$ is the pointwise extension of usual order on $\overline{\mathbb{R}}$. □

*Proof (definition 35).* To prove that $\sqsubseteq^\sharp_{\mathcal{G}}$ is and order on $\mathcal{G}$, we have to show that this relation is reflexive, antisymmetric and transitive.

*Reflexivity* Given $x \in \mathcal{G}$, let us prove that $x \sqsubseteq^\sharp_\mathcal{G} x$. If $x = \top_\mathcal{G}$ then $x \sqsubseteq^\sharp_\mathcal{G} x$. Otherwise, $x = (e, t)$ and according to the definition $x \sqsubseteq^\sharp_\mathcal{G} x$, since $t = t$ and $\sqsubseteq^\sharp_E$ is reflexive according to the previous lemma.

*Antisymmetry* Given $x, y \in \mathcal{G}$, assume $x \sqsubseteq^\sharp_\mathcal{G} y$ and $y \sqsubseteq^\sharp_\mathcal{G} x$. If $x = \top_\mathcal{G}$ then the only possibility for $x \sqsubseteq^\sharp_\mathcal{G} y$ is $y = \top_\mathcal{G}$, hence $x = y$. Otherwise, $x = (e, t)$ then $y = (e', t')$ ($y$ cannot be $\top_\mathcal{G}$ or $x$ would also be $\top_\mathcal{G}$) then the only possibility to have both $x \sqsubseteq^\sharp_\mathcal{G} y$ and $y \sqsubseteq^\sharp_\mathcal{G} x$ is in both cases the first bullet of the definition meaning $t = t'$ and $e \sqsubseteq^\sharp_E e'$ and $e' \sqsubseteq^\sharp_E e$. $\sqsubseteq^\sharp_E$ being an order according to the previous lemma, it is antisymmetric and $e = e'$. Finally $x = y$ in every cases.

*Transitivity* Given $x, y, z \in \mathcal{G}$, assume $x \sqsubseteq^\sharp_\mathcal{G} y$ and $y \sqsubseteq^\sharp_\mathcal{G} z$ and let us prove that $x \sqsubseteq^\sharp_\mathcal{G} z$. If $x = \top_\mathcal{G}$, then $y = \top_\mathcal{G}$ since $x \sqsubseteq^\sharp_\mathcal{G} y$ hence $z = \top_\mathcal{G}$ since $y \sqsubseteq^\sharp_\mathcal{G} z$. Finally $x \sqsubseteq^\sharp_\mathcal{G} z$. If $x = (e, t)$ and $y = \top_\mathcal{G}$ then $z = \top_\mathcal{G}$ and $x \sqsubseteq^\sharp_\mathcal{G} z$. If $x = (e, t)$ and $y = (e', t')$ and $z = \top_\mathcal{G}$, then $x \sqsubseteq^\sharp_\mathcal{G} z$. Thus, from now on, we assume $x = (e, t)$, $y = (e', t')$ and $z = (e'', t'')$. If $x = \bot_\mathcal{G}$ then $x \sqsubseteq^\sharp_\mathcal{G} z$. If $x \neq \bot_\mathcal{G}$ then neither $y$ nor $z$ can be $\bot_\mathcal{G}$. Thus we also assume now that neither $x$ nor $y$ nor $z$ can be $\bot_\mathcal{G}$. We know that $x \sqsubseteq^\sharp_\mathcal{G} y$ then:

- Either $t = t'$ and $e \sqsubseteq^\sharp_E e'$. Since $y \sqsubseteq^\sharp_\mathcal{G} z$:
  - Either $t' = t''$ and $e' \sqsubseteq^\sharp_E e''$ then $t' = t''$ and $e \sqsubseteq^\sharp_E e''$, hence $x \sqsubseteq^\sharp_\mathcal{G} z$.
  - Or $t' = \mathsf{fi}$ and $t'' \neq \mathsf{fi}$ and $\mathrm{loop}(e', t'') \sqsubseteq^\sharp_E e''$. According to the definition of loop, $e \sqsubseteq^\sharp_E e'$ implies $\mathrm{loop}(e, t'') \sqsubseteq^\sharp_E \mathrm{loop}(e', t'')$, hence $\mathrm{loop}(e, t'') \sqsubseteq^\sharp_E e''$. Thus $t = t' = \mathsf{fi}$, $t'' \neq \mathsf{fi}$ and $\mathrm{loop}(e, t'') \sqsubseteq^\sharp_E e''$, hence $x \sqsubseteq^\sharp_\mathcal{G} z$.
- Or $t = \mathsf{fi}$ and $t' \neq \mathsf{fi}$ and $\mathrm{loop}(e, t') \sqsubseteq^\sharp_E e'$. Since $y \sqsubseteq^\sharp_\mathcal{G} z$:
  - Either $t' = t''$ and $e' \sqsubseteq^\sharp_E e''$ implying $\mathrm{loop}(e, t') = \mathrm{loop}(e, t'') \sqsubseteq^\sharp_E e''$ hence $x \sqsubseteq^\sharp_\mathcal{G} z$.
  - Or $t' = \mathsf{fi}$ which is impossible. □

A least upper bound $\bigsqcup^\sharp_\mathcal{G}$ is defined likewise, based on the pointwise extension on $E$ of $\bigsqcup^\sharp_\mathcal{C}$. This makes $\mathcal{G}$ a complete lattice. Intuitively, for a set of graphs $G \in \mathcal{G}$, the reachable state space of $\bigsqcup^\sharp_\mathcal{G} G$ at its current point must contain the reachable state space of all graphs $g \in \mathcal{G}$ at their current point. If all graphs in $G$ have the same current point, this can simply be done by taking for each edge $(v, a, v')$ an overapproximation of the disjunction of the constraints on that edge in each $g \in G$ (thanks to $\bigsqcup^\sharp_\mathcal{C} \{e(v, a, v') \mid (e, t) \in G\}$).

**Definition 36.** *The least upper bound $\bigsqcup^\sharp_\mathcal{G} G$ of $G \subseteq \mathcal{G}$ is defined as $\bot_\mathcal{G}$ if $G = \emptyset$, as $\top_\mathcal{G}$ if $\top_\mathcal{G} \in G$ and by case analysis on the cardinal of the set of current points $\{t \mid \exists e \in E, (e, t) \in G\}$ otherwise:*

- *if $\{t \mid \exists e \in E, (e, t) \in G\} = \{\, t \,\}$*
  *then $\bigsqcup^\sharp_\mathcal{G} G = \left(\bigsqcup^\sharp_E \{e \in E \mid (e, t) \in G\}, t\right)$;*

- *if* $\{t \in V \mid \exists e \in E, (e,t) \in G\} = \{t, \mathsf{fi}\}$
  *then* $\bigsqcup^\sharp_{\mathcal{G}} G = \left(\bigsqcup^\sharp_E \{\mathrm{loop}(e,t) \mid \exists t' \in V, (e,t') \in G\}, t\right)$;
- *otherwise, the least upper bound is* $\top_{\mathcal{G}}$.

*Where* $\bigsqcup^\sharp_E$ *is the pointwise extension of* $\bigsqcup^\sharp_{\mathcal{C}}$ *(i.e., the least upper bound on* $E$ *defined for a subset* $S \subseteq E$ *as the function in* $E$ *mapping each* $(v,a,v') \in V \times \mathcal{A} \times V$ *to* $\bigsqcup^\sharp_{\mathcal{C}} \{e(v,a,v') \mid e \in S\}$*).*

**Example 46.** *Figure 10.31 illustrates the least upper bound of two values in* $\mathcal{G}$*.*

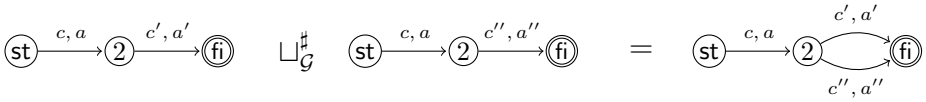



Figure 10.31 – Example of join computation ($c, c', c'' \in \mathcal{C}$, $a, a', a'' \in \mathcal{A}$, $a' \neq a''$).

It is worth noting that $\bigsqcup^\sharp_{\mathcal{G}}$ matches the join operator informally used in the introductory example in the previous section (c.f., for instance, Figure 10.23k, page 82).

*Proof (definition 36).* Given $G \subseteq \mathcal{G}$, the first thing to prove about $\bigsqcup^\sharp_{\mathcal{G}} G$ is that it is actually an element of $\mathcal{G}$. Meaning it is either $\top_{\mathcal{G}}$ or $(e,t)$ such that $t \neq st \wedge \forall v \in V, \forall a \in \mathcal{A}, e(\mathsf{fi}, a, v) = \bot_{\mathcal{C}} \wedge t \neq \mathsf{fi} \Rightarrow \forall v \in V, \forall a \in \mathcal{A}, e(v, a, \mathsf{fi}) = \bot_{\mathcal{C}}$.

- If $\bigsqcup^\sharp_{\mathcal{G}} G$ is $\top_{\mathcal{G}}$ or $\bot_{\mathcal{G}}$ then everything is fine.
- If $\{t \in V \mid \exists e \in E, (e,t) \in G\} = \{t\}$, then $t \neq \mathsf{st}$ since $G \subseteq \mathcal{G}$ and for all $(e,t')$ in $\mathcal{G}$, $\mathsf{fi}$ has neither incoming nor outgoing edges in graph $e$. Thus, for all $e$ in $\bigsqcup^\sharp_E \{e \in E \mid (e,t) \in G\}$, $\mathsf{fi}$ has neither incoming nor outgoing edges in $e$ and $\bigsqcup^\sharp_{\mathcal{G}} G = \left(\bigsqcup^\sharp_E \{e \in E \mid (e,t) \in G\}, t\right)$ is actually an element of $\mathcal{G}$.
- If $\{t \in V \mid \exists e \in E, (e,t) \in G\} = \{t, \mathsf{fi}\}$, then $t \neq \mathsf{st}$ since $G \subseteq \mathcal{G}$ and for all $(e,t')$ in $\mathcal{G}$, $\mathsf{fi}$ has no outgoing edge in graph $e$. Thus, for all $e$ in $\bigsqcup^\sharp_E \{\mathrm{loop}(e,t) \mid \exists t' \in V, (e,t') \in G\}$, $\mathsf{fi}$ has neither incoming nor outgoing edges in $e$ and $\bigsqcup^\sharp_{\mathcal{G}} G = \left(\bigsqcup^\sharp_E \{\mathrm{loop}(e,t) \mid \exists t' \in V, (e,t') \in G\}, t\right)$ is actually an element of $\mathcal{G}$.

Then, given $G \subseteq \mathcal{G}$, we have to prove that $G$ has a least upper bound for the order $\sqsubseteq^\sharp_{\mathcal{G}}$ and that it is $\bigsqcup^\sharp_{\mathcal{G}} G$.

- If $G = \emptyset$, then $\bot_{\mathcal{G}}$ is the least upper bound of $G$ since for all $x \in \mathcal{G}$, $\bot_{\mathcal{G}} \sqsubseteq^\sharp_{\mathcal{G}} x$.
- If $\top_{\mathcal{G}} \in G$, then $\top_{\mathcal{G}}$ is the only upper bound of $G$, hence the least one.
- Otherwise, if $\{t \in V \mid \exists e \in E, (e,t) \in G\} = \{t\}$, then $\bigsqcup^\sharp_E e_G$ is the least upper bound of $e_G$, where $e_G := \{e \in E \mid (e,t) \in G\}$. Thus $(\bigsqcup^\sharp_E e_G, t)$ is the least upper bound of $G$.

- Otherwise, if $\{t \in V \mid \exists e \in E, (e,t) \in G\} = \{t, \mathsf{fi}\}$, then $G$ can be partitioned in $G_t := G \cap (E \times \{t\})$ and $G_{\mathsf{fi}} := G \cap (E \times \{\mathsf{fi}\})$. On the one hand, according to the previous item, $\left(\bigsqcup_E^\sharp \{e \mid (e,t) \in G\}, t\right)$ is the least upper bound of $G_t$. On the other hand, $\left(\bigsqcup_E^\sharp \{\mathrm{loop}(e,t) \mid (e,\mathsf{fi}) \in G\}, t\right)$ is the least upper bound of $G_{\mathsf{fi}}$ having $t$ as current point. Since all $(e,t) \in G_t$ have $t \neq \mathsf{fi}$, they don't have any incoming edge on $\mathsf{fi}$, hence $\mathrm{loop}(e,t) = e$ and $\bigsqcup_E^\sharp \{e \mid (e,t) \in G\} = \bigsqcup_E^\sharp \{\mathrm{loop}(e,t) \mid (e,t) \in G\}$. Thus, the least upper bound of $G$ is $\left(\bigsqcup_E^\sharp \{\mathrm{loop}(e,t) \mid \exists t' \in V, (e,t') \in G\}, t\right)$.
- Otherwise, there are $t, t' \in V$ such that $t \neq t'$, $t \neq \mathsf{fi}$, $t' \neq \mathsf{fi}$ and both $t$ and $t'$ are in $\{t \in V \mid \exists e \in E, (e,t) \in G\}$. Thus, any upper bound of $G$ must be both in $\{\top_{\mathcal{G}}\} \cup (E \times \{t\})$ and $\{\top_{\mathcal{G}}\} \cup (E \times \{t'\})$, meaning the only upper bound of $G$ is $\top_{\mathcal{G}}$ which is then the least one. □

A concretization function is then required to formally give a meaning to abstract values. Intuitively, the concretization of a graph will be defined as the reachable state space at its current point.

**Definition 37** (concretization $\gamma_{\mathcal{G}}$)**.** *Given graph edges $e \in E$, we first define* $\mathrm{eqs}(e)$ *the following system of equations:*

$$\begin{cases} X_{\mathsf{st}} = \mathbb{V} \to \mathbb{R} \\ X_{v'} = \bigcup\limits_{\substack{v \in V, \\ a \in \mathcal{A}}} \left\{ x \mapsto [\![a(x)]\!](\rho) \;\middle|\; \rho \in X_v \wedge \bigwedge\limits_{p:expr} [\![p]\!](\rho) \leq e(v,a,v')(p) \right\} & \text{for all } v' \in V, v' \neq \mathsf{st}. \end{cases}$$

*The concretization function $\gamma_{\mathcal{G}} : \mathcal{G} \to 2^{(\mathbb{V}\to\mathbb{R})}$ is then defined as $\gamma_{\mathcal{G}}(\top_{\mathcal{G}}) = \mathbb{R}^{\mathbb{V}}$ and $\gamma_{\mathcal{G}}(e,t) = R_t$ where $(R_v)_{v\in V} \in \left(2^{(\mathbb{V}\to\mathbb{R})}\right)^V$ is the least solution[6] of* $\mathrm{eqs}(e)$ *(for the order $\dot{\subseteq}$, pointwise extension of $\subseteq$).*

It is worth noting that, like with any abstract domain, an abstract value $(e,t) \in \mathcal{G}$ overapproximates the reachable state space *at some program point* of the analyzed program. The code pointer $t$ is then used to locate this program point in the graph $e$.

**Property 10.** *$\gamma_{\mathcal{G}}$ is monotone: $\forall g, g' \in \mathcal{G}, g \sqsubseteq_{\mathcal{G}}^\sharp g' \Rightarrow \gamma_{\mathcal{G}}(g) \subseteq \gamma_{\mathcal{G}}(g')$.*

*Proof (property 10).* Given $g, g' \in \mathcal{G}$, we want to prove that $g \sqsubseteq_{\mathcal{G}}^\sharp g'$ implies $\gamma_{\mathcal{G}}(g) \subseteq \gamma_{\mathcal{G}}(g')$. For this, we analyze the various options for $g \sqsubseteq_{\mathcal{G}}^\sharp g'$ according to the definition of $\sqsubseteq_{\mathcal{G}}^\sharp$.

- If $g' = \top_{\mathcal{G}}$, then $\gamma_{\mathcal{G}}(g) \subseteq \mathbb{R}^{\mathbb{V}} = \gamma_{\mathcal{G}}(g')$.
- If $g = \bot_{\mathcal{G}}$, then $\gamma_{\mathcal{G}}(g) = \emptyset \subseteq \gamma_{\mathcal{G}}(g')$.
- If $g = (e,t)$ and $g' = (e',t')$ and $t = t'$ and $e \sqsubseteq_E^\sharp e'$, then for each edge in $V \times \mathcal{A} \times V$, the corresponding term will be smaller in the system of equations derived from $e$ than in the one derived from $e'$ hence a smaller fixpoint and $\gamma_{\mathcal{G}}(g) \subseteq \gamma_{\mathcal{G}}(g')$.

[6] The existence of which is guaranteed by Knaster-Tarski theorem since right hand sides of the equations form a monotone function on the complete lattice $\left(2^{(\mathbb{V}\to\mathbb{R})}\right)^V$.

- Otherwise, $g = (e, \mathsf{fi})$ and $g' = (e', t)$ with $t \neq \mathsf{fi}$ and $\mathrm{loop}(e, t) \sqsubseteq^\sharp_E e'$. Let us first show that $\gamma_\mathcal{G}(g) \subseteq \gamma_\mathcal{G}(\mathrm{loop}(e, t), t)$. According to the definition of $\mathcal{G}$, there are no outgoing edges on $\mathsf{fi}$, thus in the least solution $(R_v)_{v \in V}$, all $R_v$ with $v \neq \mathsf{fi}$ are not dependent on $R_\mathsf{fi}$. Then removing incoming edges of $\mathsf{fi}$ as in $\mathrm{loop}(e, t)$ does not change them and adding incoming edges to $t$ can only make $R_t$ larger. Moreover, the change from $e$ to $\mathrm{loop}(e, t)$ only moves terms from equation $X_\mathsf{fi}$ to $X_t$. Thus, $\gamma_\mathcal{G}(g) \subseteq \gamma_\mathcal{G}(\mathrm{loop}(e, t), t)$ and according to the previous item, $\gamma_\mathcal{G}(\mathrm{loop}(e, t), t) \subseteq \gamma_\mathcal{G}(e', t) = \gamma_\mathcal{G}(g')$ since $\mathrm{loop}(e, t) \sqsubseteq^\sharp_E e'$. □

As a corollary, the abstract join[7] $\sqcup^\sharp_\mathcal{G}$ is a sound overapproximation of the concrete $\cup$: $\forall x^\sharp, y^\sharp \in \mathcal{G}, \gamma_\mathcal{G}(x^\sharp) \cup \gamma_\mathcal{G}(y^\sharp) \subseteq \gamma_\mathcal{G}(x^\sharp \sqcup^\sharp_\mathcal{G} y^\sharp)$.

## 10.3 Abstract Operators

Now that our domain is defined, we have to give it abstract operators to mimic in the abstract the effects of concrete operations of our language, namely guards, assignments and random assignments.

### 10.3.1 Guards

To compute $[\![p \leq r]\!]^\sharp(g)$ for an expression $p$, a real $r \in \mathbb{R}$ and an abstract value $g \in \mathcal{G}$, we have to distinguish three cases as illustrated on Figure 10.32, page 94 and as already seen during the introductory example of previous section, page 79:

(a) when $g = \top_\mathcal{G}$, typically at starting point, we return a graph with the code pointer at $\mathsf{fi}$ and a unique edge between $\mathsf{st}$ and $\mathsf{fi}$ labeled with $(p \leq r, id)$ (c.f., Figure 10.32a, page 94);

(b) when $g = (e, \mathsf{fi})$ (i.e., the current point is $\mathsf{fi}$), we basically add the constraint $p \leq r$ to all incoming edges of $\mathsf{fi}$ (c.f., Figure 10.32b, page 94, or 10.21g and 10.22h, page 81);

(c) finally, when $g = (e, t)$ with $t \neq \mathsf{fi}$ (i.e., the current point is not $\mathsf{fi}$), we add an edge labeled with $(p \leq r, id)$ between $t$ and $\mathsf{fi}$ and change the code pointer from $t$ to $\mathsf{fi}$ (c.f., Figure 10.32c, page 94, or 10.19e, page 81).

**Definition 38.** *For any expression $p$, any real number $r \in \mathbb{R}$ and any abstract value $g \in \mathcal{G}$, $[\![p \leq r]\!]^\sharp(g)$ is defined by case analysis on $g$:*

$[\![p \leq r]\!]^\sharp(\top_\mathcal{G}) = (e, \mathsf{fi})$ *where $e$ is the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} r & \text{if } (v, a, v') = (\mathsf{st}, id, \mathsf{fi}), p' = p \\ +\infty & \text{if } (v, a, v') = (\mathsf{st}, id, \mathsf{fi}), p' \neq p \\ -\infty & \text{otherwise} \end{cases};$$

$[\![p \leq r]\!]^\sharp(e, \mathsf{fi}) = (e', \mathsf{fi})$ *where $e'$ is the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} \min(r, e(v, a, v')(p')) & \text{if } v' = \mathsf{fi}, p' = a(p) \\ e(v, a, v')(p') & \text{otherwise} \end{cases};$$

[7] No such property is required for the dual meet operator, since we only use the join during our Kleene iteration analyses (c.f., Chapter 3).

$[\![p \leq r]\!]^\sharp(e,t) = (e', \mathsf{fi})$ *where* $e'$ *is the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} r & \text{if } (v,a,v') = (t, id, \mathsf{fi}), p' = p \\ +\infty & \text{if } (v,a,v') = (t, id, \mathsf{fi}), p' \neq p \\ e(v,a,v')(p') & \text{otherwise} \end{cases} .$$

**Remark 23.** *The condition* $p' = a(p)$ *(instead of* $p' = p$*) in the second case is required since, on each edge, guards are evaluated* before *assignments according to the concretization function (Definition 37). For instance, after increasing a variable* $x$ *(assignment* $x := x + 1$*), the guard* $x \leq 2$ *must label the edge with* $x + 1 \leq 2$ *(yielding the concretization* $\{x + 1 \mid x + 1 \leq 2\} = \{x \mid x \leq 2\}$*) and not just* $x \leq 2$ *(which would lead to* $\{x + 1 \mid x \leq 2\} = \{x \mid x \leq 3\}$*). This is illustrated on Figure 10.33, page 94, and the fact was already mentioned in the introductory example (c.f., Figure 10.21g, page 81, for instance).*

**Property 11** (soundness)**.** *This abstract operator is sound with respect to the concrete semantics of guards: for all expression* $p$*, all real* $r \in \mathbb{R}$ *and all* $g \in \mathcal{G}$*,* $[\![p \leq r]\!](\gamma_{\mathcal{G}}(g)) \subseteq \gamma_{\mathcal{G}}\left([\![p \leq r]\!]^\sharp(g)\right)$*.*

*Proof (property 11).* Given an expression $p$, a real number $r$ and an abstract value $g \in \mathcal{G}$, we want to prove that $[\![p \leq r]\!](\gamma_{\mathcal{G}}(g)) \subseteq \gamma_{\mathcal{G}}\left([\![p \leq r]\!]^\sharp(g)\right)$.

- Case $g = \top_{\mathcal{G}}$: denoting $(e, \mathsf{fi}) := [\![p \leq r]\!]^\sharp(g)$, the system of equations $\mathrm{eqs}(e)$ amounts to
$$\begin{cases} X_{\mathsf{st}} = \mathbb{V} \to \mathbb{R} \\ X_{\mathsf{fi}} = \{\rho \mid \rho \in X_{\mathsf{st}} \wedge [\![p]\!](\rho) \leq r\} \end{cases}$$
which means that $\gamma_{\mathcal{G}}\left([\![p \leq r]\!]^\sharp(g)\right) = \{\rho \mid [\![p]\!](\rho) \leq r\}$. This is exactly $[\![p \leq r]\!](\gamma_{\mathcal{G}}(g)) = [\![p \leq b]\!]\left(\mathbb{R}^{\mathbb{V}}\right) = \{\rho \mid [\![p]\!](\rho) \leq r\}$, hence the inclusion.

- Case $g = (e, \mathsf{fi})$: denoting $(e', \mathsf{fi}) := [\![p \leq r]\!]^\sharp(g)$, we notice that the only modified edges between $e$ and $e'$ are incoming edges of $\mathsf{fi}$. Thus, all equations but $X_{\mathsf{fi}}$ are identical in $\mathrm{eqs}(e)$ and $\mathrm{eqs}(e')$. Since $\mathsf{fi}$ has no outgoing edges (neither in $e$ nor in $e'$), none of these identical equations depends from $X_{\mathsf{fi}}$ which means that all the $(R_v)_{v \in V}$ with $v \neq \mathsf{fi}$ in the least solutions of $\mathrm{eqs}(e)$ and $\mathrm{eqs}(e')$ are identical. The only difference then lies on $\mathsf{fi}$. Thus: $[\![p \leq r]\!](\gamma_{\mathcal{G}}(g))$

$$= [\![p \leq r]\!] \left( \bigcup_{\substack{a \in \mathcal{A} \\ v \in V, v \neq \mathsf{fi}}} \left\{ x \mapsto [\![a(x)]\!](\rho) \;\middle|\; \rho \in R_v \wedge \bigwedge_{p' : expr} [\![p']\!](\rho) \leq e(v,a,\mathsf{fi})(p') \right\} \right)$$

$$= \bigcup_{\substack{a \in \mathcal{A} \\ v \in V, v \neq \mathsf{fi}}} \left\{ x \mapsto [\![a(x)]\!](\rho) \;\middle|\; \begin{array}{l} \rho \in R_v \wedge \bigwedge_{p' : expr} [\![p']\!](\rho) \leq e(v,a,\mathsf{fi})(p') \\ \wedge [\![p]\!](x \mapsto [\![a(x)]\!](\rho)) \leq r \end{array} \right\}$$

$$= \bigcup_{\substack{a \in \mathcal{A} \\ v \in V, v \neq \mathsf{fi}}} \left\{ x \mapsto [\![a(x)]\!](\rho) \;\middle|\; \begin{array}{l} \rho \in R_v \wedge \bigwedge_{p' : expr} [\![p']\!](\rho) \leq e(v,a,\mathsf{fi})(p') \\ \wedge [\![a(p)]\!](\rho) \leq r \end{array} \right\}$$

$$= \bigcup_{\substack{a \in \mathcal{A} \\ v \in V, v \neq \mathsf{fi}}} \left\{ x \mapsto [\![a(x)]\!](\rho) \;\middle|\; \begin{array}{l} \rho \in R_v \wedge \bigwedge\limits_{p' : expr, p' \neq a(p)} [\![p']\!](\rho) \leq e(v, a, \mathsf{fi})(p') \\ \wedge [\![a(p)]\!](\rho) \leq \min(r, e(v, a, \mathsf{fi})\,(a(p)) \end{array} \right\}$$

$$= \bigcup_{\substack{a \in \mathcal{A} \\ v \in V, v \neq \mathsf{fi}}} \left\{ x \mapsto [\![a(x)]\!](\rho) \;\middle|\; \rho \in R_v \wedge \bigwedge_{p' : expr} [\![p']\!](\rho) \leq e'(v, a, \mathsf{fi})(p') \right\}$$

$$= \gamma_{\mathcal{G}}\,(e', \mathsf{fi}) = \gamma_{\mathcal{G}}\left([\![p \leq r]\!]^\sharp(g)\right).$$

- Case $g = (e, t)$ with $t \neq \mathsf{fi}$: proof similar to the previous item. □

**Remark 24** (Abstract operator is exact)**.** *As can be seen in the previous proof, the abstract operator is indeed exact (i.e., $[\![p \leq r]\!](\gamma_{\mathcal{G}}(g)) = \gamma_{\mathcal{G}}\left([\![p \leq r]\!]^\sharp(g)\right)$).*

### 10.3.2 Assignments

This is very similar to guards, modifying assignments instead of constraints on edges. The three cases are illustrated on Figure 10.34, page 94 and the first two were already seen respectively on Figures 10.16b and 10.17c, page 81.

**Definition 39.** *For any variable $x \in \mathbb{V}$, any expression $p$ and abstract value $g \in \mathcal{G}$, $[\![x := p]\!]^\sharp(g)$ is defined by case analysis on $g$:*

$[\![x := p]\!]^\sharp(\top_{\mathcal{G}}) = (e, \mathsf{fi})$ *with $e$ the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} +\infty & \text{if } (v, a, v') = (\mathsf{st}, id[x := p], \mathsf{fi}) \\ -\infty & \text{otherwise} \end{cases};$$

$[\![x := p]\!]^\sharp(e, \mathsf{fi}) = (e', \mathsf{fi})$ *with $e'$ the following function:*

$$v, a, v' \mapsto p' \mapsto \begin{cases} \max\limits_{a' \in \mathcal{A}} \{e(v, a', v')(p') \mid a'[x := a'(p)] = a\} & \text{if } v' = \mathsf{fi} \\ e(v, a, v')(p') & \text{otherwise} \end{cases};$$

$[\![x := p]\!]^\sharp(e, t) = (e', \mathsf{fi})$ *with $e'$ the following function:*

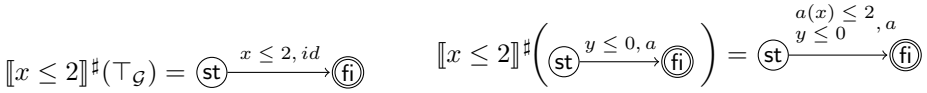
$$v, a, v' \mapsto p' \mapsto \begin{cases} +\infty & \text{if } (v, a, v') = (t, id[x := p], \mathsf{fi}) \\ e(v, a, v')(p') & \text{otherwise} \end{cases}.$$

**Property 12** (soundness)**.** *This abstract operator is sound with respect to the concrete semantics of assignments: for all variables $x \in \mathbb{V}$, for all expressions $p$ and all $g \in \mathcal{G}$, $[\![x := p]\!](\gamma_{\mathcal{G}}(g)) \subseteq \gamma_{\mathcal{G}}\left([\![x := p]\!]^\sharp(g)\right)$.*

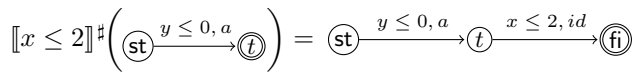
*Proof.* Similar to the proof of property 11. □

**Remark 25** (Abstract operator is exact)**.** *Again, this abstract operator is exact (i.e., $[\![x := p]\!](\gamma_{\mathcal{G}}(g)) = \gamma_{\mathcal{G}}\left([\![x := p]\!]^\sharp(g)\right)$).*

**Remark 26** (Reduced product from intervals domain)**.** *Although we do not formally define it to preserve a relative conciseness, let us briefly describe how our graph domain could benefit from information given by another domain. Consider a program containing, among others, two variables $x$ and $y$. We want a graph domain focusing only on $x$ and abstracting away all other variables.*




(a) case $g = \top_{\mathcal{G}}$ (b) case $g = (e, fi)$




(c) case $g = (e, t)$, $t \neq \mathsf{fi}$

Figure 10.32 – Examples of the three major cases for abstract guards.

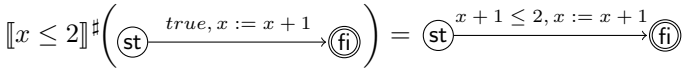
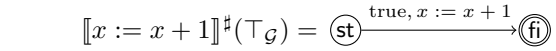
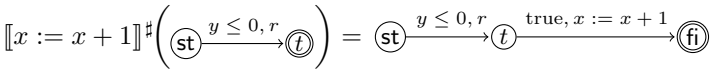



Figure 10.33 – Example of abstract guard after an assignment (c.f., Remark 23).




(a) case $g = \top_{\mathcal{G}}$


(b) case $g = (e, fi)$




(c) case $g = (e, t)$, $t \neq \mathsf{fi}$

Figure 10.34 – Examples of the three major cases for abstract assignments.

*First assume this domain has to handle the assignment $y := x + y$. It will simply ignore it since this does not impact $x$.*

*Then, assume this domain has to handle the assignment $x := x + y$ and, at this point, the intervals domain teaches us that $y$ lies in interval $[1, 2]$, then the graph domain will consider the assignment $x := x + z$ along with the constraint $1 \leq z \leq 2$ with $z \in \mathbb{V}_{ex}$ a fresh variable not appearing anywhere else in the graph.*

### 10.3.3 Random assignments

This abstract operator is kind of a merge between the two previous ones. The variable randomly assigned in range $[r_1, r_2]$ is first assigned to a fresh new variable $x \in \mathbb{V}_{ex}$ which is then constrained by $-x \leq -r_1$ and $x \leq r_2$.

**Definition 40.** *For any $x \in \mathbb{V}$, $r_1, r_2 \in \mathbb{R}$ and $g \in \mathcal{G}$, $[\![x := ?(r_1, r_2)]\!]^\sharp(g) := [\![-x_{ex} \leq -r_1]\!]^\sharp \left([\![x_{ex} \leq r_2]\!]^\sharp \left([\![x := x_{ex}]\!]^\sharp(g)\right)\right)$ with $x_{ex}$ a variable in $\mathbb{V}_{ex}$ not appearing anywhere in $g$.*

**Property 13** (soundness)**.** *This abstract operator is sound with respect to the concrete semantics of random assignments: for all variable $x \in \mathbb{V}$, all reals $r_1$ and $r_2$ and all $g \in \mathcal{G}$, $[\![x := ?(r_1, r_2)]\!](\gamma_{\mathcal{G}}(g)) \subseteq \gamma_{\mathcal{G}}\left([\![x := ?(r_1, r_2)]\!]^\sharp(g)\right)$.*

*Proof.* $x_{ex}$ being a fresh variable, the assignment followed by the two guards are semantically equivalent to the random assignment. □

**Remark 27** (Abstract operator is exact)**.** *Again, this abstract operator is exact (i.e., $[\![x := ?(r_1, r_2)]\!](\gamma_{\mathcal{G}}(g)) = \gamma_{\mathcal{G}}\left([\![x := ?(r_1, r_2)]\!]^\sharp(g)\right)$).*

## 10.4 Widening

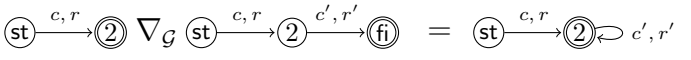
On numerical domains, widening discards information in order to enforce analysis termination, which is a source of imprecision. On the contrary, the graphs we are computing are finite objects which can be obtained without introducing imprecision. Thus, our widening operator only aims at closing loops in graphs.

None of the abstract operators we have seen up to now introduces new nodes in the graph (other than st and fi). This is done by the widening which plays a key role by introducing new nodes and closing loops on these nodes. Widening is actually the best place to create loops in our abstract control flow graphs since it is usually called at loop heads of programs during analyses[8]. Moreover, in most abstract interpreters, widening is the only indication an abstract domain gets from the presence of loops in the analyzed program.

To compute the widening $(e, t)\nabla(e', t')$ of two graphs, either:

- both $t$ and $t'$ are equal to fi: in this case, we create a new point $t''$ and redirect all incoming edges of fi to $t''$ in both $e$ and $e'$ before computing their join, this is illustrated on Figure 10.35a or on Figure 10.18d, page 81;
- one of the vertices $t$ and $t'$ is not fi and the other is (or $t = t' \neq$ fi), say $t \neq$ fi: in this case all incoming edges of fi are redirected to $t$ in $e'$ and a pointwise widening is done on each edge, Figures 10.35b or 10.24m;
- both $t$ and $t'$ are not fi and are different: in this case we return $\top_{\mathcal{G}}$.


(a) both code pointers equal to fi ($\bot_{\mathcal{G}} = (\dot{\bot}_{\mathcal{C}}, \mathsf{fi})$), typical case before entering a loop




(b) one code pointer not equal to fi, typical case after a first loop iteration

Figure 10.35 – Widening: introducing new nodes and loops in the graph.

**Definition 41.** *Assuming a constant $c \in \mathbb{N}$ and a widening $\nabla_s$ on lattice $\overline{\mathbb{R}}$ (equipped with the usual order) such that $-\infty \nabla_s -\infty = -\infty$, we define the widening $\nabla_{\mathcal{G}} : \mathcal{G} \times \mathcal{G} \to \mathcal{G}$ as:*

$$g \nabla_{\mathcal{G}} g' = \begin{cases} (\mathrm{loop}(e,t),t) \\ \sqcup^{\sharp}_{\mathcal{G}} (\mathrm{loop}(e',t),t) & \text{if } g = (e,\mathsf{fi}) \text{ and } g' = (e',\mathsf{fi}) \text{ with } t \notin \{\mathsf{st},\mathsf{fi}\} \text{ such that :} \\ & \forall v, \forall a, e(v,a,t) = e(t,a,v) = e'(v,a,t) = e'(t,a,v) = \bot_{\mathcal{C}} \\ \nabla_E(\mathrm{loop}(e,t), \\ \mathrm{loop}(e',t),t) & \text{if } g = (e,t) \text{ and } g' = (e',\mathsf{fi}) \text{ and } t \neq \mathsf{fi} \\ & \text{or } g = (e,\mathsf{fi}) \text{ and } g' = (e',t) \text{ and } t \neq \mathsf{fi} \\ & \text{or } g = (e,t) \text{ and } g' = (e',t') \text{ and } t = t' \neq \mathsf{fi} \\ \top_{\mathcal{G}} & \text{otherwise} \end{cases}$$

*with $\nabla_E : E \times E \times V \to \mathcal{G}$ the following function:*

$$e, e', t \mapsto \begin{cases} e'', t & \text{if } \#\{v,a,v' \mid e(v,a,v') \neq \bot_{\mathcal{C}} \vee e'(v,a,v') \neq \bot_{\mathcal{C}}\} \leq c \\ \top_{\mathcal{G}} & \text{otherwise} \end{cases}$$

*in which $e''$ stands for:*

$$v, a, v' \mapsto p \mapsto \begin{cases} e(v,a,v')(p) \nabla_s e'(v,a,v')(p) & \text{if } \mathrm{fin}(e(v,a,v')) \text{ and } \mathrm{fin}(e'(v,a,v')) \\ +\infty & \text{otherwise} \end{cases}$$

*with eventually* $\mathrm{fin}(c')$ *the predicate:* $(c' = \bot_{\mathcal{C}}) \vee (\#\{p \mid c'(p) < +\infty\} < +\infty)$.

The constant $c$ is mostly needed to enforce theoretical termination. In practice, the number of branches in the analyzed program being finite, any value larger than the number of edges in the control flow graph of the program will work. Again, the predicate "fin" plays a mere theoretical role and is always true during an actual computation since we do not manipulate infinite sets of constraints. More intuitively, the control flow graph of the analyzed program being finite, there is basically nothing to do to enforce termination and this widening is mostly here to add nodes for loop heads and close loops, i.e., to record loop heads of the analyzed program.

[8] It even *must* be called at least once per loop to ensure convergence of the analyses [Bou93].

*Proof (definition 41).* We have to prove for any $g, g' \in \mathcal{G}$ that $g \nabla_{\mathcal{G}} g' \in \mathcal{G}$. If $g = (e, \mathsf{fi}) \in \mathcal{G}$ and $g' = (e', \mathsf{fi}) \in \mathcal{G}$, $(\text{loop}(e,t),t)$ and $(\text{loop}(e',t),t)$ are in $\mathcal{G}$ for any $t \in V$. Thus $(\text{loop}(e,t),t) \sqcup^\sharp_{\mathcal{G}} (\text{loop}(e',t),t) \in \mathcal{G}$. $\top_{\mathcal{G}}$ is also in $\mathcal{G}$. It then remains to prove that $\nabla_E(\text{loop}(e,t), \text{loop}(e',t), t) \in \mathcal{G}$ when $g = (e,t)$ and $g' = (e', \mathsf{fi})$ or $g = (e, \mathsf{fi})$ and $g' = (e', t)$ or $g = (e,t)$ and $g' = (e', t')$ with $t = t' \neq \mathsf{fi}$. In those three cases, $\text{loop}(e,t)$ and $\text{loop}(e',t)$ do not have incoming nor outgoing edges on $\mathsf{fi}$ and this property is preserved by $\nabla_E$. □

**Property 14.** *$\nabla_{\mathcal{G}}$ is a widening on $\mathcal{G}$: for all $g, g' \in \mathcal{G}$, $g \sqcup^\sharp_{\mathcal{G}} g' \sqsubseteq^\sharp_{\mathcal{G}} g \nabla_{\mathcal{G}} g'$ and for all sequences $g \in \mathcal{G}^{\mathbb{N}}$, the sequence defined as $g'_0 = g_0$, $g'_{i+1} = g'_i \nabla_{\mathcal{G}} g_{i+1}$ is ultimately stationary.*

*Proof (property 14).* Given $g, g' \in \mathcal{G}$, let us prove that $g \sqcup^\sharp_{\mathcal{G}} g' \sqsubseteq^\sharp_{\mathcal{G}} g \nabla_{\mathcal{G}} g'$. This works well since the order $\sqsubseteq^\sharp_{\mathcal{G}}$ was specially crafted to that end.

- If $g = (e, \mathsf{fi})$ and $g' = (e', \mathsf{fi})$), then $g \sqsubseteq^\sharp_{\mathcal{G}} (\text{loop}(e,t),t)$ and $g' \sqsubseteq^\sharp_{\mathcal{G}} (\text{loop}(e',t),t)$. Thus, $g \sqcup^\sharp_{\mathcal{G}} g' \sqsubseteq^\sharp_{\mathcal{G}} (\text{loop}(e,t),t) \sqcup^\sharp_{\mathcal{G}} (\text{loop}(e',t),t) = g \nabla_{\mathcal{G}} g'$.
- If $g \nabla_{\mathcal{G}} g' = \top_{\mathcal{G}}$, the property trivially holds.
- Otherwise, $g \sqsubseteq^\sharp_{\mathcal{G}} (\text{loop}(e,t),t)$ and $g' \sqsubseteq^\sharp_{\mathcal{G}} (\text{loop}(e',t),t)$ and $(\text{loop}(e,t),t) \sqcup^\sharp_{\mathcal{G}} (\text{loop}(e',t),t) \sqsubseteq^\sharp_{\mathcal{G}} \nabla_E(\text{loop}(e,t), \text{loop}(e',t), t)$.

Given a sequence $g \in \mathcal{G}^{\mathbb{N}}$, we want to prove that the sequence $g'_0 = g_0$, $g'_{i+1} = g'_i \nabla_{\mathcal{G}} g_{i+1}$ is ultimately stationary. There is a vertex $t \in V$, $t \neq \mathsf{fi}$ such that there is $k \in \mathbb{N}$ such that for all $i \geq k$, either $g'_i = \top_{\mathcal{G}}$ or $\exists e \in E, g'_i = (e,t)$. If $g'_i$ is not $\top_{\mathcal{G}}$, its number of edges (not equal to $\bot_{\mathcal{C}}$) is bounded by the constant $c$. And for each edge, the number of constraint having a bound strictly less than $+\infty$ can only decrease and is finite thanks to the predicate fin. Finally, for each constraint, the widening $\nabla_s$ enforces the convergence. Thus, the sequence g' is ultimately stationary. □

## 10.5 Examples and Remarks

**Example 47** (nested loops)**.** *Figure 10.36 illustrates the handling of nested loops by the domain.*

```
i := 0; j := 0;
while i ≤ 9 do
    while j ≤ 9 do
        j := j+1
    od;
    i := i+1
od
```

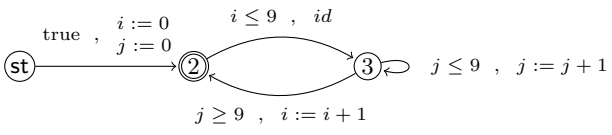



Figure 10.36 – Example of nested loops. The graph on the right is the result of Kleene iterations over the code on the left at head of the outer loop.

**Example 48.** *Figure 10.37, page 99, displays a Kleene iteration on the following program:*

```
x := ?(0, 1); y := ?(0, 1);
while −1 ≤ 0 do
  in := ?(0, 1);
  if in ≤ 0.9 then
    t := x;
    x := 0.2×t − 0.7×y + 0.5×in;
    y := 0.7×t + 0.2×y + 0.5×in
  else
    x := 10×in − 9;
    y := 10×in − 9
  fi
od
```

The presented domain allows to provide, for future use by a policy iterations abstract domain, a global view of the system, as a control flow graph. This graph will widely depend on the way iterations are performed, but results are usually rather expectable. For instance, use of a delayed widening will make the head of the loop in the graph preceded by a few unfolded iterations of the loop body, whereas interleaving unions between widenings will result in some loop unfolding.

Regarding the cost of all those operations on graphs, we can notice that it is very low with respect to the cost of policy iterations we will perform in the next section. However, there is a fundamental complexity issue since if-then-else statements can give rise to a graph with a number of edges exponential in the size of the analyzed program. This is an already known difficulty in policy iteration and other work [GM11, GM12] offers to rely on an implicit representation of the system of equations and to efficiently choose improving policies thanks to an SMT solver. It should be pretty easy to adapt the current graph domain to this setting.

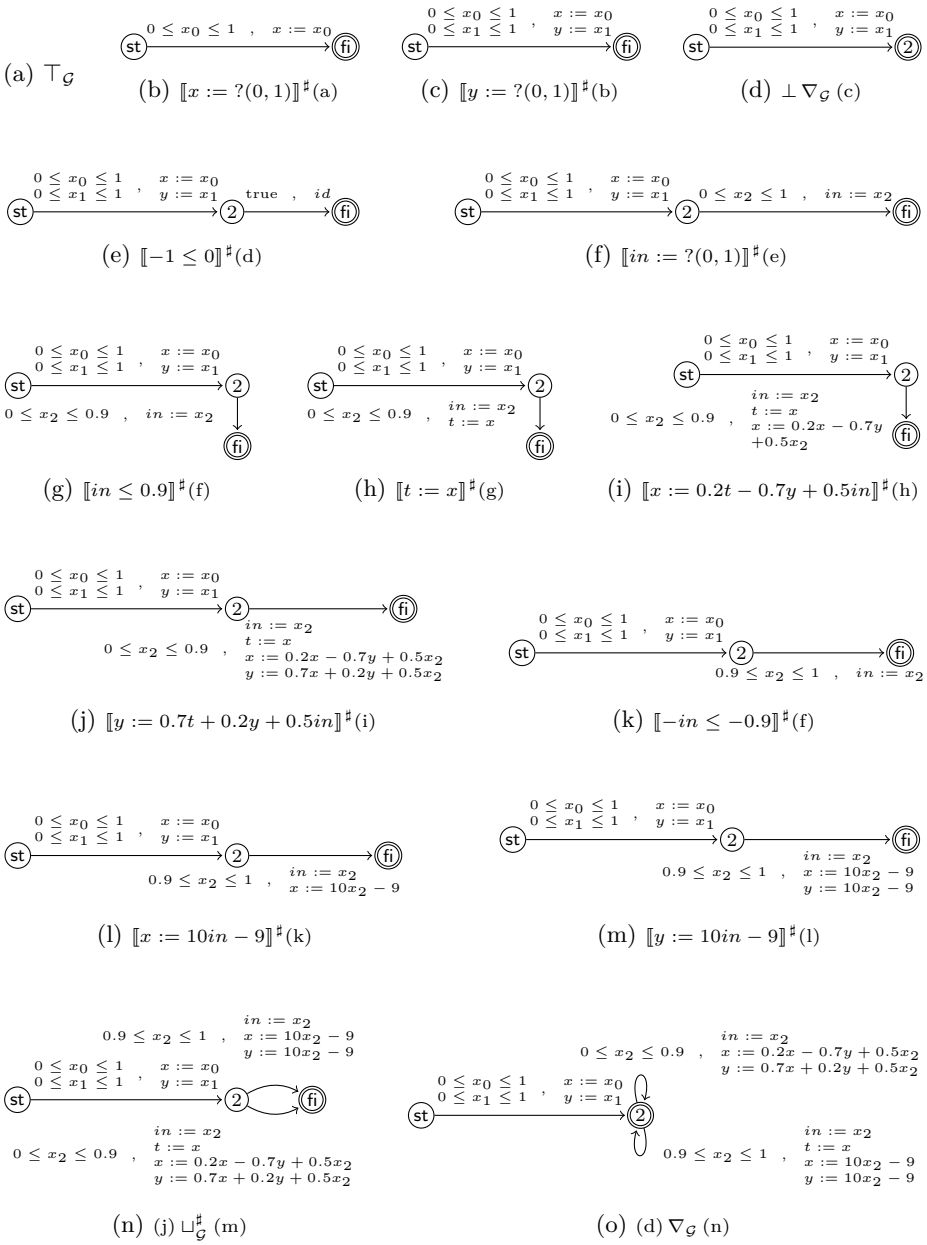



Figure 10.37 – Example of Kleene iterations on the program of Example 48. An additional iteration, not displayed here, will then prove that (o) is a fixpoint.

# Chapter 11

# Application to Compute Quadratic Invariants

## 11.1 Embedding Policy Iterations into an Abstract Domain

This chapter finally describes how to use the control flow graph domain of the previous chapter to embed policy iterations into a classic abstract domain.

The basic idea is to build a product of the control flow graph domain with a template domain. Policy iterations are then performed during widenings to reduce the template part according to the graph part.

### 11.1.1 Reduced product between Graph and Template Domains

First, the template domain $\mathcal{T}$ (c.f., Definition 30, page 30) is equipped with dummy abstract operators $[\![.]\!]^\sharp_{\mathcal{T}}$ for guards, assignments and random assignments that always return $\top_{\mathcal{T}} = (+\infty, \ldots, +\infty)$. Though perfectly useless, those operators are trivially sound.

**Definition 42** (policy iterations domain $\mathcal{P}_i$)**.** *We define the domain $\mathcal{P}_i$ as the product $\mathcal{G} \times \mathcal{T}$ of the domain of previous chapter with the above template domain.*

This means all operations on $\mathcal{P}_i$ are performed component by component. For instance, for $(g, b)$ and $(g', b') \in \mathcal{P}_i$, the join $(g, b) \sqcup^\sharp_{\mathcal{P}_i} (g', b')$ is defined as[1] $(g \sqcup^\sharp_{\mathcal{G}} g', b \sqcup^\sharp_{\mathcal{T}} b')$. The concretization function $\gamma_{\mathcal{P}_i} : \mathcal{P}_i \to 2^{\mathbb{V} \to \mathbb{R}}$ is the intersection of the concretizations of each component: $\gamma_{\mathcal{P}_i}(g, b) = \gamma_{\mathcal{G}}(g) \cap \gamma_{\mathcal{T}}(b)$.

At this point, this domain still looks completely useless. But all the policy iteration work will take place during its widening.

**Definition 43** (widening of $\mathcal{P}_i$)**.** *We define $\nabla_{\mathcal{P}_i} : \mathcal{P}_i \times \mathcal{P}_i \to \mathcal{P}_i$ as:*

$$(g, b) \nabla_{\mathcal{P}_i} (g', b') = \begin{cases} (g \nabla_{\mathcal{G}} g', \mathrm{PI}\,(g \nabla_{\mathcal{G}} g')) & \text{if } g' \sqsubseteq^\sharp_{\mathcal{G}} g \\ (g \nabla_{\mathcal{G}} g', \top_{\mathcal{T}}) & \text{otherwise} \end{cases}$$

[1] The join $\sqcup^\sharp_{\mathcal{T}}$ on $\mathcal{T} = \overline{\mathbb{R}}^n$ is simply the pointwise extension of the usual max on $\overline{\mathbb{R}}$.

*where* $\mathrm{PI}(g)$ *is the result of policy iterations applied on graph*[2] $g$ *(c.f., Chapter 9).*

The test $g' \sqsubseteq^\sharp_{\mathcal{G}} g$ is used to perform policy iterations only when the graph domain has stabilized, thus avoiding potentially costly, and rather useless, computations on yet incomplete graphs.

As usual with reduced products [CCF+06], $\nabla_{\mathcal{P}_i}$ is not a widening in the strict acceptation of the term, since it is not greater than the join of $\mathcal{P}_i$. However, it still satisfies the two following fundamental properties:

- it does not break the soundness of the analysis since for all $p, p' \in \mathcal{P}_i$: $\gamma_{\mathcal{P}_i}(p \sqcup^\sharp_{\mathcal{P}_i} p') \subseteq \gamma_{\mathcal{P}_i}(p \nabla_{\mathcal{P}_i} p')$;
- it ensures termination of the analysis: for all sequences $x \in {\mathcal{P}_i}^{\mathbb{N}}$, the sequence $y_0 = x_0$, $y_{i+1} = y_i \nabla_{\mathcal{P}_i} x_{i+1}$ is ultimately stationary.

Equipped with $\nabla_{\mathcal{P}_i}$ as widening operator, $\mathcal{P}_i$ finally offers a classic abstract domain interface to policy iterations.

### 11.1.2 Remarks on this Embedding

One may find the previous construction quite complicated and ask why not simply perform policy iterations aside classic Kleene iterations at each loop head. This seemingly simpler approach would however suffer from following drawbacks:

- it is not confined to an abstract domain, breaking the usual abstract interpreter framework [Jea10];
- this would hinder the use of reduced products to exchange information with other domains, since a more static approach would be unable to record those informations on the fly as our graph domain can.

Finally, due to first point, implementation could rapidly become more intricate.

## 11.2 How to Choose Appropriate Templates ?

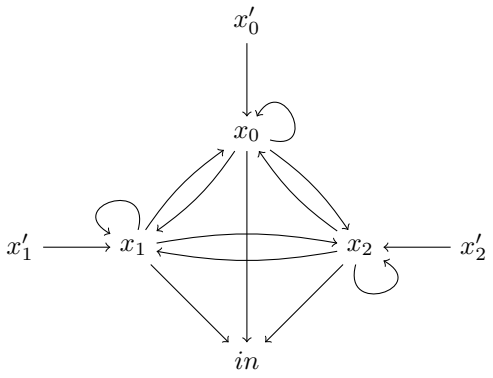
Given matrices $A$ and $B$ of a purely linear system $x_{k+1} = Ax_k + Bu_k$, Chapter 5 gave a method to compute matrices $P$ such that there is a $\lambda$ making the ellipsoid $\left\{x \;\middle|\; x^T P x \leq \lambda\right\}$ a reasonably tight invariant of the system.

This allows to design a simple quadratic template generation heuristic for programs. A template is simply generated (using methods of Chapter 5) for each pair of matrices $A$ and $B$ that can be extracted from each edge of the control flow graph. Indeed, such matrices $A$ and $B$ can be easily extracted from a control flow graph edge by considering strongly connected components of the relation "variable $y$ linearly depends on variable $z$" in the assignments of the edge.

**Example 49.** *Considering the control flow graph of Figure 10.29, page 86, the edge between vertices* st *and* 2 *holds the assignment* $x_0 := 0, x_1 := 0, x_2 := 0$. *This gives an empty dependency graph, hence without strongly connected components. Thus no template is generated for this edge. For the second edge, looping on vertex*

[2] More precisely on the system of equations introduced in Chapter 9.

2, *the assignment is* $x'_0 := x_0$, $x'_1 := x_1$, $x'_2 := x_2$, $x_0 := 0.9379\,x_0 - 0.0381\,x_1 - 0.0414\,x_2 + 0.0237\,in$, $x_1 := -0.0404\,x_0 + 0.968\,x_1 - 0.0179\,x_2 + 0.0143\,in$, $x_2 := 0.0142\,x_0 - 0.0197\,x_1 + 0.9823\,x_2 + 0.0077\,in$ *which gives the following dependency graph between variables:*




.

*This graph has one strongly connected component involving variables* $x_0$, $x_1$ *and* $x_2$ *which all depend on variable in. Thus, matrices* $A$ *and* $B$ *are extracted from the assignment such that*

$$\begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}_{k+1} = A \begin{bmatrix} x_0 \\ x_1 \\ x_2 \end{bmatrix}_k + B \left[ in \right]_k$$

*where*

$$A := \begin{bmatrix} 0.9379 & -0.0381 & -0.0414 \\ -0.0404 & 0.968 & -0.0179 \\ 0.0142 & -0.0197 & 0.9823 \end{bmatrix} \qquad B := \begin{bmatrix} 0.0237 \\ 0.0143 \\ 0.0077 \end{bmatrix}.$$

This way, we automatically compute quadratic templates from the control flow graph computed by the domain of Chapter 10, which, added to templates bounding each variable $y$, usually enable policy iterations to produce tight invariants. These results can then be easily exported to other domains (such as the intervals domain for instance) through reduced products.

Of course, this choice of considering edges independently is purely heuristic. The fact that an appropriate template is found for each edge does not guarantee to be able to bound the whole program, not even that this program is stable.

**Example 50.** *The two following programs are stable (they even converge toward the same state* $x = 0, y = 0$*):*

```
x := 0; y := 1;
while −1 ≤ 0 do
  x' := x; y' := y;
  x := 0.985×x − 0.3953×y;
  y := 0.0247×x + 0.985×y;
od
```

```
x := 0; y := 1;
while −1 ≤ 0 do
  x' := x; y' := y;
  x := 0.985×x − 0.0247×y;
  y := 0.3953×x + 0.985×y;
od
```

*but the following program combining them is not:*

```
x := 0; y := 1;
while −1 ≤ 0 do
  if (x+y) × (x−y) ≤ 0 then
    x' := x; y' := y;
    x := 0.985×x − 0.3953×y;
    y := 0.0247×x + 0.985×y;
  else
    x' := x; y' := y;
    x := 0.985×x − 0.0247×y;
    y := 0.3953×x + 0.985×y;
  fi
od.
```

*This is illustrated on Figure 11.1.*

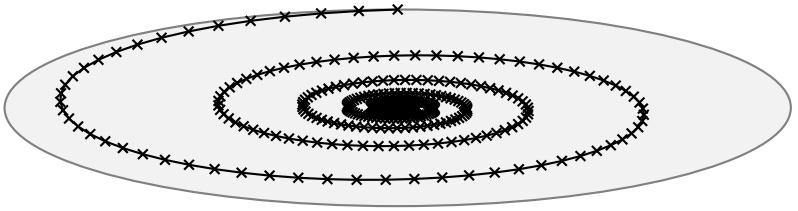
Even something seemingly as benign as a saturation can turn a stable system into something that is no longer stable.

**Example 51.** *On the two following programs, the one on the left is stable, whereas the one on the right (which is just the same program with an added saturation) no longer is.*

```
x := 1; y := 0;
while −1 ≤ 0 do
  x' := x; y' := y;
  x := 1.1168×x − 0.1647×y;
  y := 0.1647×x + 0.8533×y;
od
```

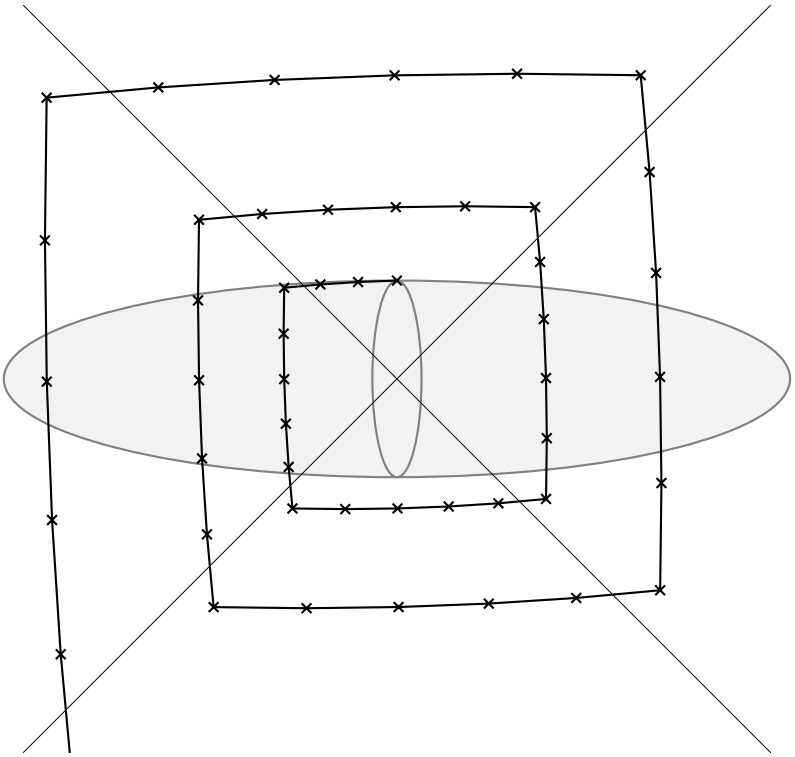
```
x := 0; y := 1;
while −1 ≤ 0 do
  x' := x; y' := y;
  x := 1.1168×x − 0.1647×y;
  y := 0.1647×x + 0.8533×y;
  if y ≤ −0.5 then
    y := −0.5;
  else y := y; fi
od
```

However, the heuristic remains reasonable since control systems usually relies on a purely linear core. Things can then be added around this core but are not expected to disturb it too much. An idea to improve on this point could be to replace the single Lyapunov equation in the methods of Chapter 5 by a system of equations with one equation for each edge of the control flow graph.



(a) system 0 (stable)

(b) system 1 (stable)



(c) combination of systems 0 and 1 (not stable)

Figure 11.1 – A linear system with guards combining two stable purely linear systems is not necessarily stable.

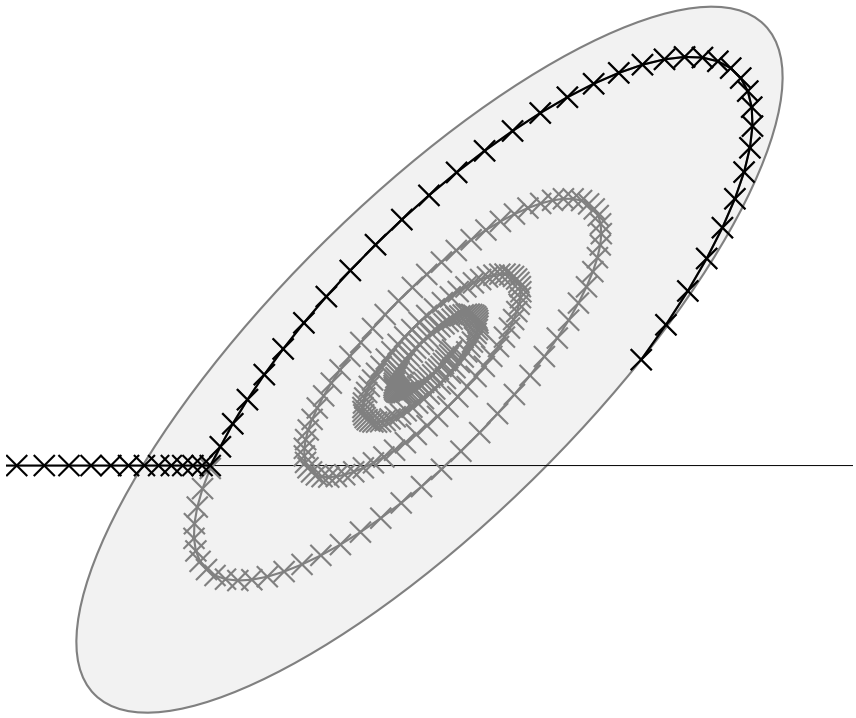


Figure 11.2 – A simple saturation can turn a stable system (light gray trace, for which the displayed ellipsoid is an invariant) in something no longer stable (dark gray trace, diverging to the left).

# Chapter 12

# Floating Point Issues

As in Chapter 6, two issues have to be addressed: the use of floating point computations in the analyzed program and in the analyzer.

## 12.1 Taking Rounding Errors Into Account

Taking into account rounding errors in the analyzed program has not yet been done (i.e., the program is considered as if all computations were performed using real numbers) and is left as future work. However, the method of Section 6.1 should be quite easily adaptable.

## 12.2 Checking Soundness of the Result

Policy iterations return a vector of values $(b_{i,j}) \in \overline{\mathbb{R}}^{V \times [\![1,n]\!]}$ (where $V$ is the set of vertices of the control flow graph an $n$ is the number of templates). This result was computed using floating point values and we have to check that it is actually an overapproximation of the least solution of the system of equations solved by policy iterations (introduced in Section 9.2).

This amounts to check that for each equation (i.e., for each vertex $v' \in V$ and each template $t_j$), the following inequality holds

$$b_{v',j} \geq \bigvee_{\substack{v \in V,\\ a \in \mathcal{A}}} \max \left\{ \begin{bmatrix} x \\ 1 \end{bmatrix}^T A_a^T P_{t_j} A_a \begin{bmatrix} x \\ 1 \end{bmatrix} \;\middle|\; \begin{array}{l} \bigwedge_{p:expr} \begin{bmatrix} x \\ 1 \end{bmatrix}^T P_p \begin{bmatrix} x \\ 1 \end{bmatrix} \leq e(v,a,v')(p) \\ \wedge \bigwedge_{j' \in [\![1,n]\!]} \begin{bmatrix} x \\ 1 \end{bmatrix}^T P_{t_{j'}} \begin{bmatrix} x \\ 1 \end{bmatrix} \leq b_{v,j'} \end{array} \right\}$$

where the $a \in \mathcal{A}$ represent assignments and $e \in E$ the edges of the control flow graph, as defined respectively in Definitions 32 and 34, page 85. $A_a$ is a matrix implementing the assignments $a$ (i.e., for all $x$, $A_a \left[x^T, 1\right]^T = \left[x'^T, 1\right]^T$ where $x'$ is the result of assignments $a$ on $x$ (we only consider linear assignments)) whereas $P_p$ translates the expression $p$ (i.e., unfolding $\left[x^T, 1\right] P_p \left[x^T, 1\right]^T$ gives the expression $p$ (we only consider at most quadratic templates and guards)). Thus, we have to check for each pair of vertices $v, v' \in V$ and each template $t_j$

that the following inequality holds

$$b_{v',j} \geq \max \left\{ \begin{bmatrix} x \\ 1 \end{bmatrix}^T A_a^T P_{t_j} A_a \begin{bmatrix} x \\ 1 \end{bmatrix} \;\middle|\; \begin{array}{l} \bigwedge_{p:expr} \begin{bmatrix} x \\ 1 \end{bmatrix}^T P_p \begin{bmatrix} x \\ 1 \end{bmatrix} \leq e(v,a,v')(p) \\ \wedge \bigwedge_{j' \in [\![1,n]\!]} \begin{bmatrix} x \\ 1 \end{bmatrix}^T P_{t_{j'}} \begin{bmatrix} x \\ 1 \end{bmatrix} \leq b_{v,j'} \end{array} \right\}.$$

We can restrict ourselves to the case where everything in the above inequality lies in $\mathbb{R}$ since constraints of the form $. \leq +\infty$ can just be ignored and constraints $. \leq -\infty$ make the above set empty, hence its max equal to $-\infty$.

This eventually amounts to check that for all $v, v' \in V$ and all $t_j$

$$\forall x, \bigwedge_{p:expr} \begin{bmatrix} x \\ 1 \end{bmatrix}^T P_p \begin{bmatrix} x \\ 1 \end{bmatrix} \leq e(v,a,v')(p) \;\wedge \bigwedge_{j' \in [\![1,n]\!]} \begin{bmatrix} x \\ 1 \end{bmatrix}^T P_{t_{j'}} \begin{bmatrix} x \\ 1 \end{bmatrix} \leq b_{v,j'}$$
$$\Rightarrow \begin{bmatrix} x \\ 1 \end{bmatrix}^T A_a^T P_{t_j} A_a \begin{bmatrix} x \\ 1 \end{bmatrix} \leq b_{v',j}.$$

After a relaxation, this can be checked by looking at the positive definiteness of a matrix, similarly to what was done in Section 6.2, page 53.

# Chapter 13

# Implementation and Experimental Results

Our proposal has been instantiated on the analysis of linear systems with guards admitting quadratic inductive invariants. These linear systems are widely present in critical embedded systems like aerospace control-command software but are hard to analyze with most abstract domains since they usually do not admit simple linear inductive invariants. The use of our framework enables a fully automatic analysis of such systems, relying on policy iterations with semi-definite programming, while other approaches either impose stronger restrictions on the class of analyzable programs or require more parameters to enable the analysis. To author knowledge, this is the first automatic use of policy iterations with quadratic templates integrated in a classic abstract interpretation based static analyzer and its first use without providing any information, such as templates, except the set of variables that have to be considered for the policy iteration domain, like for any other relational abstraction.

Section 13.1 first details some more or less crucial implementation points before Section 13.2 gives experimental results.

## 13.1 Implementation

This Section highlights a few features of our implementation of min- and max-policy iterations to compute quadratic invariants on linear systems. Some are simple hacks to improve analysis performances. Others were needed to achieve full automaticity or just to get any result at all on our benchmarks.

### 13.1.1 Templates

Template domains used by policy iteration require templates to be given prior to the analyses. This greatly limits the automaticity of the method. Section 11.2, page 102, gave a heuristic to generate interesting quadratic templates. Finally, as seen in the running example of Chapter 9, we add templates $x^2$ for each variable modified by the program. In the literature [AGG10, GS10, GSA+12], templates $x$ and $-x$ are often used but, since results are usually symmetrical (i.e. the same bound $b$ is obtained for both templates: $x \leq b$ and $-x \leq b$), templates

$x^2$ yield the same result (i.e. $x^2 \leq b^2$) making use of two times fewer templates for policy iteration[1], hence saving on computation costs.

### 13.1.2 Initial Value

In the policy iteration literature, system of equations require extra terms with initial values for each template at loop head. Although those values do not come totally out of the blue, computing them does not appear absolutely obvious. As seen in the running example of Chapter 9, page 63, we chose to replace them by an initial vertex (vertex 1 in Figure 9.1) initialized with bound $+\infty$ for each template (i.e., the program starts in an unknown memory state) and linked to loop head (vertex 2 in Figure 9.1) by an edge with initialization code. Thus, the previously mentioned initial values for each template will actually be computed by policy iteration and no longer need to be provided by the user.

Considering policy iteration themselves, max-policy iterations start from $(-\infty, \ldots, -\infty)$ whereas min-policies need to start from a postfixpoint. Such a postfixpoint could be computed through Kleene iterations using a simple widening with thresholds. However, just starting from a large value (for instance $10^6$) for the quadratic templates computed in 11.2 and $+\infty$ for all others often yields in practice the same results at a much lower cost.

### 13.1.3 Interval Constraints

To enable the use of semi-definite programming solvers, a relaxation must be used. It basically amounts to the following theorem.

**Theorem 4** (Lagrangian relaxation)**.** *Assume $f$ and $g_1, \ldots, g_k$ functions $\mathbb{R} \to \mathbb{R}$, if there exist $\lambda_1, \ldots, \lambda_k \in \mathbb{R}$ all non negative such that.*

$$\forall x, f(x) - \sum_i \lambda_i g_i(x) \geq 0 \tag{13.1}$$

*then*

$$\forall x, \left( \bigwedge_i g_i(x) \geq 0 \right) \Rightarrow f(x) \geq 0. \tag{13.2}$$

Semi-definite programming solvers being unable to directly handle Equation (13.2), they are fed with Equation (13.1). This usually works well. However the converse of Theorem 4 does not generally hold. In particular with a quadratic objective $f$ and two linear constraints $g_1$ and $g_2$.

**Example 52.** *We want to apply a relaxation on $x \in [1, 3] \Rightarrow -x^2 + 4x + 5 \geq 0$, that is Equation (13.2) with $f := x \mapsto -x^2 + 4x + 5$, $g_1 := x \mapsto x - 1$ and $g_2 := 3 - x$. Equation (13.1) then boils down to: $\forall x, -x^2 + (4 - \lambda_1 - \lambda_2)x + (5 + \lambda_1 - 3\lambda_2) \geq 0$. Unfortunately, not any non negative $\lambda_1, \lambda_2 \in \mathbb{R}$ satisfy this. This is depicted on left of Figure 13.1.*

This case is commonly encountered in practice, for instance with initial values of a program living in some range or with inputs bounded by an interval. Replacing the two linear constraints by an equivalent quadratic one constitutes an efficient workaround.

[1] Moreover, $x^2$ is easier to express in semi-definite programs than linear constraints.

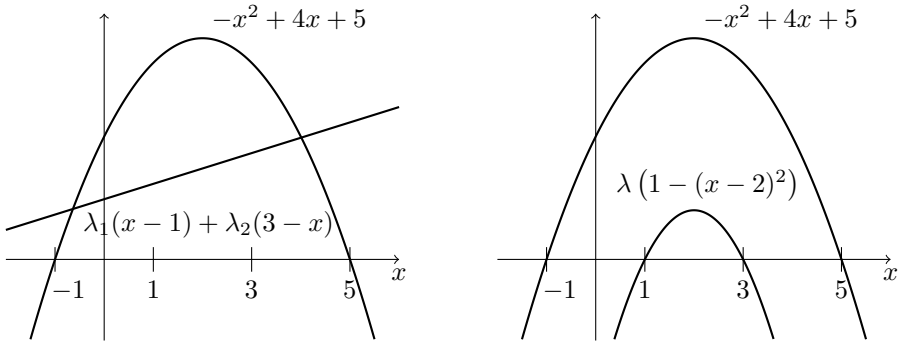



Figure 13.1 – Relaxation of interval constraints.

**Example 53.** *When constraints* $x - 1 \geq 0$ *and* $3 - x \geq 0$ *are replaced by the equivalent* $1 - (x-2)^2 \geq 0$*, relaxation works just fine (with relaxation coefficient* $\lambda = 2$ *for instance). This is depicted on right of Figure 13.1.*

### 13.1.4 Floating Point

For the sake of efficiency, the semi-definite programming solvers we use perform all their computations on floating point numbers and do not offer any strict soundness guarantee on their results. Section 12.2, page 107 already demonstrated how to check the soundness of these results. However, simply computing with floating point numbers and checking the final result would miserably fail since such a result is usually slightly unsound. To address this issue, equations have to be padded before being fed to the numerical solvers.

Padding the equations means for min-policies multiplying each temporary result $b_i$ by $(1+\epsilon)$ for some small $\epsilon$. For max-policies, all equations $\max\{p \mid q \leq c\}$ are basically replaced by $\max\{(1+\epsilon)p \mid q \leq (1+\epsilon)c\}$. In practice, while using solvers trying to achieve an accuracy of $10^{-8}$ on their results, a value of $10^{-4}$ for $\epsilon$ appears to be a good choice. The induced loss of accuracy on the final result is considered acceptable since bounds finally computed by our analysis are usually found to be at least a few percent larger than the actual maximum values reachable by the program. Finding a way to pad equations to get correct results, while still preserving a good accuracy, however remains somehow a black art.

Finally, a quick and dirty hack to recover a correct result in the rare event where the soundness check fails consists in multiplying the — probably false — result by a small constant (for instance 1.01) and checking again its soundness. This sometimes enable to get a better result than $\top$, despite the first check failure, at the very low cost of an additional check.

Although all this gives satisfying results. It would remain interesting to compare the cost/accuracy trade off when using the verified solver VSDP [JCK07] as already offered in the literature [AGG10].

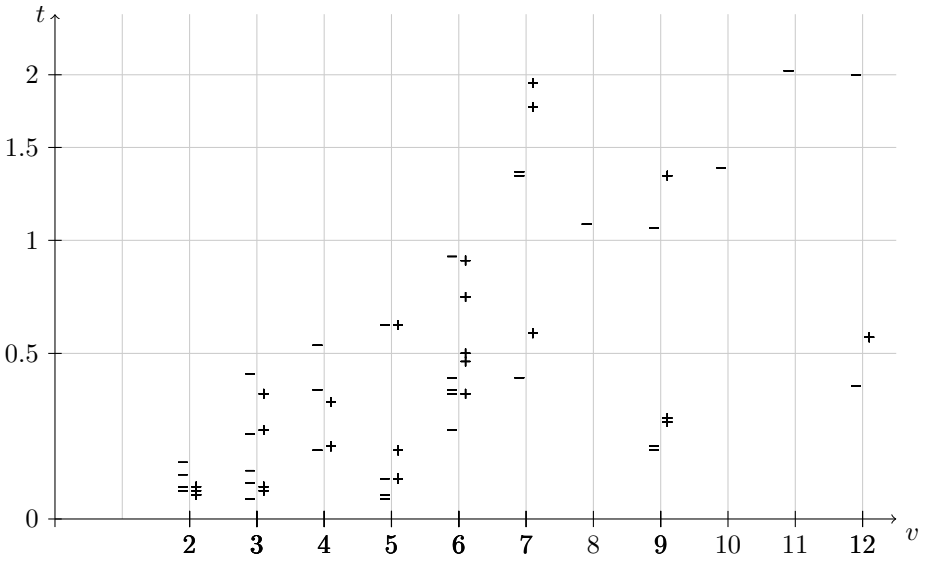



Figure 13.2 – Time ($t$ in seconds) spent performing min ($-$ signs) and max ($+$ signs) policy iterations depending on the number $v$ of variables in the analyzed program. Less $+$ than $-$ in a column indicates a failure of max-policies on a benchmark. All computations were performed on an Intel Core2 @ 2.66GHz.

## 13.2 Experimental Results

All the elements presented in this part of the document have been implemented as a new abstract domain in our static analyzer. Experiments were conducted on a set of stable linear systems. These systems were extracted from the literature [AGG10, Fer05, RJGF12, SB13]. The analyzer is released under GPL and available with all examples at `https://cavale.enseeiht.fr/phd2013/`.

As seen in Chapter 9.3, two methods exist to compute invariants by policy iterations, namely min and max policies. Figure 13.2 compares analysis times with min and max-policy iterations. In both cases, the number of iterations always remained reasonable. For min-policies, the number of iterations performed lies between 3 and 7 when the stopping criterion is a relative progress below $10^{-4}$ between two consecutive iterates. For max-policies, the number of iterations was between 4 and 7. As shown on Figure 13.2, computation times for min and max-policies are comparable (max-policies being slightly faster for small problems whereas they are outperformed by min-policies for larger ones[2]) but the actual difference appear on the largest benchmarks for which max-policies where unable to produce sound results while min-policies did[3]. Finally, it can be noticed that, when both methods work, results obtained with min and max-policies are the same. However, paradoxically enough, min-policies yield slightly more precise results. For all these reasons, min policies were made the default in our tool.

[2]This can be explained by min-policies solving smaller semi-definite programming problems [GSA+12, Conclusion] (c.f., Remarks 16, page 69 and 18, page 71).

[3]Again, this is explained by the fact that max-policies have to solve larger semi-definite programming problems, incurring more numerical difficulties.

Figure 13.2 only gave times for policy iterations. Total analysis times also include building the control flow graph and the equation system, computing appropriate templates and eventually checking soundness of the result. Time needed for control flow graph construction and soundness checking is very small compared to the time spent in policy iterations, whereas computing templates takes the same amount of magnitude in time than min-policies iteration. All this is detailed in Table 13.1.

Finally, Table 13.2 detail the bounds obtained for each benchmark and compares them with the maximum reachable values of the programs. It is worth noting that, for the purely linear systems, these bounds are usually better than the one obtained in Chapter 7 thanks to the extra templates bounding each variable. Bounds for 'Ex. 7' and 'Ex. 8' are also better than the one given in the literature they are taken from [AGG10, GS10, GSA+12] thanks to the good automatically generated quadratic templates.

**Example 54** (Reduced Product)**.** *Table 13.3 considers two linear systems chained, the output of the first one being used as input by the second one. This program is analyzed with two policy iteration domains communicating together via reduced product to and from the domain of intervals (the two domains do not share any variable). It is worth noting that total analyses time is just the sum of the times needed for the two separate analyses. In comparison, an analysis with one single domain for the whole program is much more expensive.*

| | $n$ | Total (s) | Templates (s) | Iterations (s) | Check (s) |
|---|---|---|---|---|---|
| Ex. 1 | 3 | 0.77 | 0.38 | 0.05 | 0.01 |
| | 4 | 1.29 | 0.51 | 0.59 | $\perp$ (0.00) |
| | 3 | 0.53 | 0.38 | 0.08 | 0.02 |
| Ex. 2 | 5 | 0.67 | 0.52 | 0.05 | 0.02 |
| | 6 | 1.14 | 0.52 | 0.36 | 0.12 |
| | 5 | 0.74 | 0.52 | 0.10 | 0.04 |
| Ex. 3 | 3 | 0.50 | 0.33 | 0.13 | 0.02 |
| | 4 | 0.99 | 0.31 | 0.51 | 0.09 |
| | 3 | 0.61 | 0.32 | 0.22 | 0.04 |
| Ex. 4 | 4 | 0.66 | 0.38 | 0.18 | 0.04 |
| | 5 | 1.25 | 0.37 | 0.59 | 0.14 |
| | 4 | 0.88 | 0.38 | 0.37 | 0.06 |
| Ex. 5 | 6 | 0.92 | 0.47 | 0.23 | 0.06 |
| | 7 | 2.37 | 0.48 | 1.31 | 0.26 |
| | 6 | 1.17 | 0.47 | 0.41 | 0.11 |
| Ex. 6 | 6 | 1.39 | 0.78 | 0.35 | 0.08 |
| | 7 | 2.70 | 0.78 | 1.28 | 0.28 |
| | 6 | 2.02 | 0.78 | 0.90 | 0.15 |
| Ex. 7 | 2 | 0.34 | 0.21 | 0.08 | 0.01 |
| | 3 | 0.79 | 0.21 | 0.43 | 0.10 |
| | 2 | 0.39 | 0.19 | 0.14 | 0.03 |
| Ex. 8 | 2 | 0.31 | 0.20 | 0.07 | 0.01 |
| | 3 | 1.46 | 0.21 | 0.25 | 0.09 |
| | 2 | 0.42 | 0.21 | 0.12 | 0.03 |

Table 13.1 – Result of the experiments: quadratic invariants inference. The examples are the same than in Chapter 7, page 55. For each example, the first line is for the bare linear system (as in Chapter 7), the second for the same system with an added saturation and the third with a reset. Column $n$ gives the number of program variables considered for policy iteration while column 'Total' gives the time spent for the whole analysis. The remaining columns detail the computation time: 'Templates' corresponds to the quadratic template computation, 'Iterations' to the actual policy iterations and 'Check' to the soundness checking. $\perp$ indicates failure of the soundness checking.

| | Bounds | Reachable |
|---|---|---|
| Ex. 1 | $15.99, 15.99$ | $14.84, 14.84$ |
| | $+\infty, +\infty$ | $12.31, 12.31$ |
| | $15.99, 15.99$ | $14.84, 14.84$ |
| Ex. 2 | $1.65, 1.65, 1.00, 1.00$ | $1.42, 1.42, 1.00, 1.00$ |
| | $2.43, 0.50, 1.00, 1.00$ | $1.04, 0.50, 1.00, 1.00$ |
| | $1.65, 1.65, 1.00, 1.00$ | $1.42, 1.42, 1.00, 1.00$ |
| Ex. 3 | $4.03, 20.41$ | $3.97, 20.00$ |
| | $4.14, 21.41$ | $2.04, 1.68$ |
| | $4.03, 20.41$ | $3.97, 20.00$ |
| Ex. 4 | $0.44, 0.37, 0.56$ | $0.32, 0.24, 0.22$ |
| | $0.46, 0.39, 0.59$ | $0.19, 0.11, 0.17$ |
| | $1.09, 1.09, 1.58$ | $1.00, 1.00, 1.00$ |
| Ex. 5 | $4.57, 4.72, 4.34, 4.38$ | $2.79, 2.73, 3.50, 3.30$ |
| | $3.53, 6.93, 5.50, 6.10$ | $1.28, 1.69, 3.31, 2.87$ |
| | $4.57, 4.72, 4.34, 4.38$ | $2.79, 2.73, 3.50, 3.30$ |
| Ex. 6 | $1.42, 1.10, 1.75, 1.81, 2.57$ | $1.42, 0.91, 1.44, 1.52, 2.14$ |
| | $1.03, 1.90, 3.07, 4.98, 6.66$ | $1.03, 0.65, 0.77, 0.88, 1.16$ |
| | $1.42, 1.77, 2.63, 3.14, 4.45$ | $1.42, 0.91, 1.44, 1.52, 2.14$ |
| Ex. 7 | $1.45, 1.06$ | $1.29, 1.00$ |
| | $1.00, 1.00$ | $1.00, 1.00$ |
| | $1.45, 1.06$ | $1.29, 1.00$ |
| Ex. 8 | $1.27, 1.27$ | $1.10, 1.00$ |
| | $1.00, 1.01$ | $1.00, 0.99$ |
| | $1.27, 1.27$ | $1.10, 1.00$ |

Table 13.2 – Result of the experiments: quadratic invariants inference. The examples are the same than in Table 13.1. Column 'Bounds' gives the bounds on absolute values of each variables inferred and proved by the tool whereas column 'Reachable' gives underapproximations of the maximum reachable values (obtained by random simulation) for comparison purpose.

| | $n$ | Total (s) | Templates (s) | Iterations (s) | Check (s) |
|---|---|---|---|---|---|
| Ex. 5 alone | 6 | 0.92 | 0.47 | 0.23 | 0.06 |
| Ex. 6 alone | 6 | 1.39 | 0.78 | 0.35 | 0.08 |
| one domain | 12 | 5.74 | 1.92 | 2.01 | 0.51 |
| two domains | 6+6 | 2.28 | 0.45+0.78 | 0.24+0.36 | 0.06+0.08 |

Table 13.3 – Result of the experiments: reduced product. Examples 5 and 6 from Table 13.1 are chained. The columns are the same than in Table 13.1. The first two lines recall the results for examples 5 and 6 alone, the line 'one domain' uses a single big policy iterations domain for the whole program whereas the line 'two domains' uses two smaller domains communicating together through reduced product.

# Chapter 14

# Conclusion

To the author's knowledge this part of this document presents the *first integration of policy iterations as a classic relational abstract domain*[1]. Such an integration in classic Kleene fixpoint iterations is enabled thanks to an *abstract domain* that *rebuilds the control flow graph* and allows the policy iteration algorithm to access a global view of the program (or the part of program the user wants to focus on with policy iterations) as a system of equations (which constitutes the usual input for policy iterations). This is then applied to build a domain of quadratic templates[2] thanks to a method, based on [RJGF12], to *synthesize meaningful ellipsoid templates* for a specific class of programs: stable guarded linear systems. A powerful abstract domain is thus provided, able to compute non linear invariants in a fully automatic way, while offering the same interface than relational abstractions such as polyhedra. Indeed, this abstract domain could be integrated in a numerical abstract domain library, such as Apron [JM09], without any modification of its interface.

Reduction between classic domains and our allows both to precisely represent the control flow graph provided to policy iterations and to inject the result of policy iterations within classic domains. It also enables the use of multiple policy iteration domains; for example when considering sequences of linear filters as in Example 54.

The experimental results show that this approach really extends the applicability of Kleene-based abstract interpreters to a wider class of invariants. When computing our analyzes we only provided the set of variables that have to be analyzed with policy iterations, without any other information like templates.

Finally the issue of floating point semantics should not be forgotten. The introduction of error terms has to be addressed.

[1] The other way round, i.e., integration of Kleene iterations into policy iterations, was already proposed [SJVG11]. However, this did not enable such a straightforward use of policy iterations into classic abstract interpretation based static analyzers.

[2] Of course, this is only a particular application, the same could be done with other policy iterations techniques such as policy iterations with linear templates.

# Part IV

# Polynomial Systems

# Chapter 15

# A Polynomial Template Domain

The previous parts of this document described how to compute quadratic invariants for linear systems (with linear or quadratic guards). This chapter offers an abstract domain to infer polynomial invariants, of degree potentially higher than two, on polynomial programs, of degree potentially higher than one.

The proposal is based on a template abstract domain that relies on polynomial global optimization procedures based on Bernstein polynomials. This polynomial representation admits a set of interesting properties for verification purposes. Our proposal describes how to rely on such polynomials to reason over systems expressed with polynomials. We also took care of the floating point implementation issues, providing a safe over-approximation of the reachable states of such systems. This contribution is a first step towards the general analysis of nonlinear systems, for example through polynomialization of system's equations via Taylor or Poisson expansions. This chapter is mostly constituted from a workshop paper [RG13b].

The chapter is structured as follows. The remainder of this introduction section presents notations and introduces a running example. Then Section 15.2 offers a basic overview of Bernstein polynomials based global optimization. Section 15.3 presents our instantiation of a template domain with polynomials. Finally Sections 15.4 and 15.5 compare the current approach with related work and conclude.

## 15.1 Notations

Given some $n \in \mathbb{N}^*$, we denote $n$-tuples $(x_1, \ldots, x_n)$ by bold letters $\boldsymbol{x}$. $\mathbf{0}$ stands for the tuple $(0, \ldots, 0)$. We extend order relation $\leq$ on tuples: $\boldsymbol{x} \leq \boldsymbol{y}$ if for all $i$ between 1 and $n$, $x_i \leq y_i$. The interval notation $[\mathbf{0}, \mathbf{1}]$ then represents all tuples $\boldsymbol{x}$ satisfying $\mathbf{0} \leq \boldsymbol{x} \leq \mathbf{1}$. For $\boldsymbol{x} \in \mathbb{R}^n$ and $\boldsymbol{i} \in \mathbb{N}^n$, $\boldsymbol{x}^{\boldsymbol{i}}$ is the monomial $\prod_{j=1}^{n} x_j^{i_j}$. Finally $\mathbb{R}[\boldsymbol{x}]$ is the set of polynomials with coefficients in $\mathbb{R}$ over variables $x_1, \ldots, x_n$. Polynomials $p \in \mathbb{R}[\boldsymbol{x}]$ are of the form $p(\boldsymbol{x}) = \sum_{\mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{d}} c_{\boldsymbol{i}} \boldsymbol{x}^{\boldsymbol{i}}$

```
x := 0; y := ?(0, 0.5);
while x ≤ 1 do
    y := y + 0.001 × (18x² − 18x + 3);
    x := x + 0.001;
    if y ≤ 0 then y := 0 else y := y fi od
```

Figure 15.1 – Example of program.

where $\boldsymbol{d} \in \mathbb{N}^n$ is called *degree* of the multivariate polynomial $p$[1].

**Example 55.** *Figure 15.1 presents an integrator — a basic construction of typical command control systems — performing a discrete integration of the polynomial* $18x^2 - 18x + 3$*. On this example, the intervals abstract domain do not allow to bound the variable* $y$*, the polyhedra domain gives the bound* $y \leq 3.5$[2] *and the domain from Part III is of no use here, whereas the domain developed in this chapter will allow to prove that* $y \leq 0.833$ *(the actual maximum being between 0.7901 and 0.7902).*

## 15.2 Bernstein Polynomials Optimization

The goal of this section is to introduce polynomial global optimization based on Bernstein polynomials [GJS03, MN12, NA11, RN09, Smi09] by giving the basic principles and algorithms. This will then be used in the next section at the core of a polynomial template abstract domain.

Bernstein polynomials were first introduced in 1912 by Sergei Natanovich Bernstein to constructively prove Weierstrass' theorem. However, their use really took off in the early sixties with the work of Paul de Faget de Casteljau and Pierre Étienne Bézier to describe car bodyworks in the French car industry. They are now a fundamental tool in the field of Computer Aided Geometric Design and are also used in other domains, such as global optimization [GJS03, MN12, NA11, RN09, Smi09], for their nice properties when representing polynomials on closed intervals. The interested reader is referred to [Far12] for a detailed overview.

### 15.2.1 Bernstein Polynomials

**Definition 44** (Bernstein basis)**.** *Any polynomial* $p \in \mathbb{R}[\boldsymbol{x}]$ *can be written*

$$p(\boldsymbol{x}) = \sum_{\boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{d}} b_{p,\boldsymbol{i}} B_{\boldsymbol{d},\boldsymbol{i}}(\boldsymbol{x}) \qquad \text{with} \qquad B_{\boldsymbol{d},\boldsymbol{i}}(\boldsymbol{x}) = \prod_{j=1}^{n} \binom{d_j}{i_j} x_j^{i_j} (1 - x_j)^{d_j - i_j}.$$

When considering a polynomial $p$ on the box $[\boldsymbol{0}, \boldsymbol{1}]$, the coefficients $b_{p,\boldsymbol{i}}$ of this representation have the interesting property of bounding $p$.

[1] It should be noted that $\boldsymbol{d}$ is a tuple, which differs from the usual notion of degree of polynomials defined as $\max \left\{ \sum_{j=1}^{n} i_j \;\middle|\; \boldsymbol{0} \leq \boldsymbol{i} \leq \boldsymbol{d}, c_{\boldsymbol{i}} \neq 0 \right\}$.

[2] When the nonlinear assignment $y := y + 0.001(18x^2 - 18x + 3)$ is abstracted by $y := y + [-0.018, 0]x + 0.003$ knowing that $0 \leq x \leq 1$.

**Property 15.** *For any polynomial $p$, for all $\boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]$,*

$$\min\{b_{p,\boldsymbol{i}} \mid \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{d}\} \leq p(\boldsymbol{x}) \leq \max\{b_{p,\boldsymbol{i}} \mid \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{d}\}.$$

*Moreover, $p(\boldsymbol{i}) = b_{p,\boldsymbol{i}}$ for all $\boldsymbol{i} \in \mathcal{C}_{\boldsymbol{d}}$ where we call* endpoints *the indices in the set $\mathcal{C}_{\boldsymbol{d}} = \{\boldsymbol{i} \in \mathbb{N}^n \mid \forall j, 0 \leq j \leq n \Rightarrow (i_j = 0 \vee i_j = d_j)\}$.*

This property can be generalized to any box $[\boldsymbol{a}, \boldsymbol{b}]$ by considering the polynomial $p'(\boldsymbol{x}) = p\left(\sigma_{[\boldsymbol{a},\boldsymbol{b}]}(\boldsymbol{x})\right)$ with $\sigma_{[\boldsymbol{a},\boldsymbol{b}]}$ mapping each $x_i$ to $a_i + (b_i - a_i)x_i$.

### 15.2.2 Optimization Problem

We are targeting the following polynomial global optimization problem:

$$\max\{p(\boldsymbol{x}) \mid q_1(\boldsymbol{x}) \leq b_1 \wedge \ldots \wedge q_k(\boldsymbol{x}) \leq b_k\} \tag{15.1}$$

where $p$, $q_1, \ldots, q_k \in \mathbb{R}[\boldsymbol{x}]$ are multivariate polynomials such that for all $i \in \{1, \ldots, n\}$, the polynomials $x_i$ and $-x_i$ are included in the set $\{q_1, \ldots, q_k\}$ and $b_1, \ldots, b_k$ are constant bounds in $\mathbb{R}$. That is the maximum value reached by a polynomial on a feasible set defined by a box and potential polynomial constraints. This can be $-\infty$ if the feasible set is empty and a value in $\mathbb{R}$ otherwise.

**Definition 45.** *We will later denote (15.1) as* $\mathrm{Opt}(p; q_1, \ldots, q_k)(b_1, \ldots, b_k)$.

**Example 56.** $\mathrm{Opt}\left(x_1 + x_2; x_1, -x_1, x_2, -x_2, x_1^2 - x_2\right)(1, 0, 1, 0, 0) = 1$ *for instance, and* $\mathrm{Opt}\left(x_1 + x_2; x_1, -x_1, x_2, -x_2, x_1^2 - x_2\right)(1, 0, 1, 0, -2) = -\infty$.

### 15.2.3 Branch and Bound Algorithm

Our goal is to compute an upper bound of the previously defined value (15.1) within some accuracy $\epsilon$. Let us assume we are considering polynomial $p$ under polynomial constraints $q_1 \leq 0, \ldots, q_k \leq 0$ on the unit box $[\mathbf{0}, \mathbf{1}]$ (this is equivalent to an arbitrary box $[\boldsymbol{a}, \boldsymbol{b}]$ up to an affine transformation). According to Property 15, if there exist a $q_j$ such that $\min\left\{b_{q_j,\boldsymbol{i}} \;\middle|\; \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{d}\right\} > 0$ then the constraint $q_j \leq 0$ is not satisfiable on the box $[\mathbf{0}, \mathbf{1}]$, hence the result $-\infty$. Otherwise $\mathit{bound} := \max\{b_{p,\boldsymbol{i}} \mid \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{d}\}$ is an upper bound of the result. And according to the second part of Property 15, $\mathit{reached} := \max\left\{b_{p,\boldsymbol{i}} \;\middle|\; \boldsymbol{i} \in \mathcal{C}_{\boldsymbol{d}}, \forall j, b_{q_j,\boldsymbol{i}} \leq 0\right\}$ is a lower bound. If $\mathit{bound} - \mathit{reached} \leq \epsilon$ then $\mathit{bound}$ is an upper bound with the expected accuracy and we are done, otherwise we subdivide the box $[\mathbf{0}, \mathbf{1}]$ into two smaller boxes and recursively apply the same criteria on them.

Given the $d + 1$ Bernstein coefficients $b_{p,i}$ of a single variable polynomial $p$ of degree $d$, the *de Casteljau's algorithm* computes the Bernstein coefficients of polynomials $p_{left}$ and $p_{right}$ such that $p_{left}(x) = p(\frac{x}{2})$ and $p_{right}(x) = p(\frac{x+1}{2})$ for all $x \in [0, 1]$. That is, the polynomial $p$ on $[0, 1]$ is *subdivided* into $p_{left}$ on $[0, \frac{1}{2}]$ and $p_{right}$ on $[\frac{1}{2}, 1]$. This is performed in $\Theta(d^2)$ arithmetic operations and can be done on any variable for a multivariate polynomial. The interested reader is referred to [MN12] for more details.

Using this, algorithms 1 and 2 allow to numerically solve the optimization problem (15.1) within some accuracy $\epsilon$[3]. The convergence of this procedure is known to be quadratic which means tight bounds can "rapidly" be obtained through it. This algorithm was already presented in [NA11] along with heuristics to improve its performance.

[3] Provided a big enough bound $s$ on recursion depth.

**Algorithm 1** Auxiliary function for max_under_constraints (Algorithm 2). Polynomial $p$ is the objective function whereas *constraints* is a set of polynomial constraints. $\boldsymbol{d}$ is the degree of those polynomials. $k$ and $s$ are respectively current and maximum allowed recursion depth while $\epsilon$ is the required accuracy. Finally *reached* is a lower bound of final result used to prune the search tree. The returned value is a pair $(r, b)$ satisfying $r \leq \max\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}] \wedge \forall c \in constraints, c(\boldsymbol{x}) \leq 0\} \leq b$. multipoly is the type of multivariate polynomials.

max_under_constraints_rec($p$ : multipoly, *constraints* : multipoly set,
$\boldsymbol{d}$ : int tuple, $s$ : int, $k$ : int, $\epsilon$ : real, *reached* : real)
  $this\_reached \leftarrow \max\{b_{p,\boldsymbol{i}} \mid \boldsymbol{i} \in \mathcal{C}_{\boldsymbol{n}}, \forall c \in constraints, b_{c,\boldsymbol{i}} \leq 0\}$
    ▷ Here:
  $this\_reached \leq \max\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}] \wedge \forall c \in constraints, c(\boldsymbol{x}) \leq 0\}$.
  **if** $\exists c \in constraints, \min\{b_{c,\boldsymbol{i}} \mid \mathbf{0} \leq \boldsymbol{i} \leq \boldsymbol{d}\} > 0$ **then**
    $this\_bound \leftarrow -\infty$ ▷ Constraint $c$ is not satisfiable.
  **else**
    $this\_bound \leftarrow \max\{b_{p,\boldsymbol{i}} \mid \boldsymbol{i} \leq \boldsymbol{n}\}$
  **end if**
    ▷ Here: $\max\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}] \wedge \forall c \in constraints, c(\boldsymbol{x}) \leq 0\} \leq this\_bound$.
  **if** $k \geq s \vee this\_bound \leq this\_reached + \epsilon \vee this\_bound \leq reached + \epsilon$ **then**
    ▷ If maximum recursion depth or required accuracy is reached
    ▷ or if greater value was already reached on a previously
    ▷ explored part of the domain, stop here.
    **return** $(this\_reached, this\_bound)$
  **end if**
  $j \leftarrow \text{varsel}(p, \boldsymbol{d}, k)$ ▷ Function varsel returns the index $j$ of one of the
    ▷ $n$ variables $x_j$ of polynomials $p$ and *constraints*.
  $\{l\}, \{r\} \leftarrow \text{subdiv}(\{p\}, j)$ ▷ Function subdiv(*pols*, $j$) applies
  $cl, cr \leftarrow \text{subdiv}(constraints, j)$ ▷ de Casteljau's algorithm to each
    ▷ polynomial in set *pols* on variable $x_j$.
  $reached \leftarrow \max(reached, this\_reached)$
  $rl, bl \leftarrow$ max_under_constraints_rec$(l, cl, \boldsymbol{d}, s, k+1, \epsilon, reached)$
  $reached \leftarrow \max(reached, rl)$
  $rr, br \leftarrow$ max_under_constraints_rec$(r, cr, \boldsymbol{d}, s, k+1, \epsilon, reached)$
  **return** $(\max(rl, rr), \max(bl, br))$

**Algorithm 2** max_under_constraints$(p, constraints, \boldsymbol{lbs}, \boldsymbol{ubs}, s, \epsilon)$ computes an upper bound of $\max\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\boldsymbol{lbs}, \boldsymbol{ubs}] \wedge \forall c \in constraints, c(\boldsymbol{x}) \leq 0\}$, algorithm stops when either a bound of accuracy $\epsilon$ or recursion depth $s$ is reached.

max_under_constraints($p$ : multipoly, *constraints* : multipoly set, $\boldsymbol{lbs}$ : int tuple,
$\boldsymbol{ubs}$ : int tuple, $s$ : int, $\epsilon$ : real)
  $p, constraints, \boldsymbol{d} \leftarrow \text{translate}(p, constraints, \boldsymbol{lbs}, \boldsymbol{ubs})$
    ▷ Function translate first translates polynomials from box $[\boldsymbol{lbs}, \boldsymbol{ubs}]$ to
    ▷ the unit box $[\mathbf{0}, \mathbf{1}]$ then to Bernstein basis, it also returns their degree $\boldsymbol{d}$.
  $\_, bound \leftarrow$ max_under_constraints_rec$(p, constraints, \boldsymbol{d}, s, 0, \epsilon, -\infty)$
  **return** *bound*

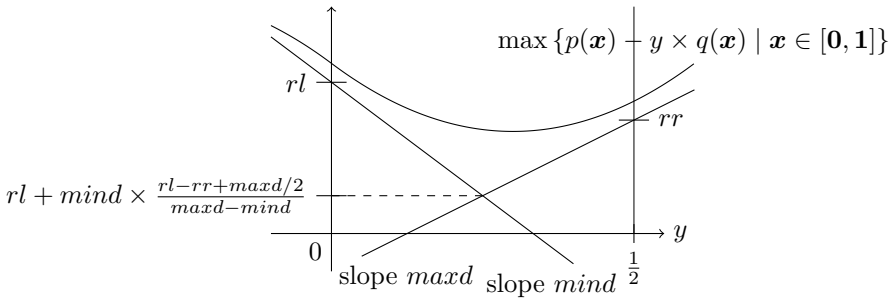



Figure 15.2 – Illustration of Property 17.

### 15.2.4 Lagrangian Relaxation

Similarly to what was done in the previous parts of this document with quadratic constraints, a lagrangian relaxation could be used.

**Property 16** (Relaxation scheme)**.** *For any polynomials $p$ and $q$, any box $[\boldsymbol{a}, \boldsymbol{b}]$ and any real $c$, if there exist a real $y \geq 0$ such that:*

$$\forall \boldsymbol{x} \in [\boldsymbol{a}, \boldsymbol{b}], p(\boldsymbol{x}) - y \times q(\boldsymbol{x}) \leq c$$

*then:*

$$\forall \boldsymbol{x} \in [\boldsymbol{a}, \boldsymbol{b}], q(\boldsymbol{x}) \leq 0 \Rightarrow p(\boldsymbol{x}) \leq c.$$

The reverse direction does not hold but if for some real number $y$, the polynomial $p - y \times q$ is bounded on the box $[\boldsymbol{a}, \boldsymbol{b}]$, this bound also holds for $\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\boldsymbol{a}, \boldsymbol{b}] \wedge q(\boldsymbol{x}) \leq 0\}$. By trying various values for $y$, we can then obtain a bound on the constrained problem while only computing bounds of polynomials without constraints.

The following property will allow to safely ignore some intervals while looking for appropriate values for relaxation variable $y$.

**Property 17.** *For any polynomials $p$ and $q$, if there exist $mind, maxd, rl, rr \in \mathbb{R}$ such that:*

- $\forall \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}],\ mind \leq -q(\boldsymbol{x}) \leq maxd$*;*
- $\max\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]\} \geq rl$ *and* $\max\left\{p(\boldsymbol{x}) - \frac{1}{2} \times q(\boldsymbol{x}) \;\middle|\; \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]\right\} \geq rr$*;*

*then, for all $y \in \left[0, \frac{1}{2}\right]$:*

$$\max\{p(\boldsymbol{x}) - y \times q(\boldsymbol{x}) \mid \boldsymbol{x} \in [\mathbf{0}, \mathbf{1}]\} \geq rl + mind \times \frac{rl - rr + maxd/2}{maxd - mind}.$$

*Proof.* The result follows from the fact that $\frac{d(p-yq)}{dy} = -q$ is bounded by $mind$ and $maxd$. □

This property, illustrated on Figure 15.2, allows one to prune branches while looking for a value of $y$ leading to an interesting bound as done in Algorithms 3 and 4. This can be generalized to more than one constraint by adding as many relaxation variables $y$ as constraints.

This algorithm is an alternative to the one of the previous Section 15.2.3 which appears to be more efficient in cases when the objective function is close

**Algorithm 3** Auxiliary function for max_relaxation (Algorithm 4). $pyq$ is the polynomial $p - yq$ with $p$ the objective function and $q$ the constraint. $\boldsymbol{d}$ is the degree of this polynomial. $k$ and $s$ are respectively current and maximum recursion depth. $\epsilon$ is the accuracy within which upper bounds are computed for each value of $y$ tested. $rl$ and $rr$ are lower bound of the maximum of $pyq$ for respective values 0 and 1 of $y$. $mind$ and $maxd$ are bounds on $\frac{d(pyq)}{dy}$. The result is an upper bound of $pyq$ for some value of $y \in (0, 1)$.

max_relaxation_rec($pyq$ : multipoly, $\boldsymbol{d}$ : int tuple, $s$ : int, $k$ : int, $\epsilon$ : real, $rl$ : real, $rr$ : real, $mind$ : real, $maxd$ : real)
  $l, r \leftarrow$ subdiv$(pyq, y)$
  ▷ First get a bound for $y = \frac{1}{2}$.
  $this\_reached, this\_bound \leftarrow$ max_under_constraints_rec$(r[y \leftarrow 0], \emptyset, \boldsymbol{d}, s, 0, \epsilon, -\infty)$
  ▷ Then look for a smaller bound with $y \in \left(0, \frac{1}{2}\right)$.
  **if** $k + 1 < s \ \wedge \ rl + mind \times \frac{rl - this\_reached + maxd/2}{maxd - mind} < bound - \epsilon$ **then**
    $bl \leftarrow$ max_relaxation_rec$(l, \boldsymbol{d}, s, k + 1, \epsilon, rl, this_reached, mind/2, maxd/2)$
    **if** $bl < this\_bound$ **then**
      $this\_bound \leftarrow bl$
    **end if**
  **end if**
  ▷ Finally look for a smaller bound with $y \in \left(\frac{1}{2}, 1\right)$.
  **if** $k + 1 < s \ \wedge \ rr - maxd \times \frac{rr - this\_reached - mind/2}{maxd - mind} < bound - \epsilon$ **then**
    $br \leftarrow$ max_relaxation_rec$(r, \boldsymbol{d}, s, k + 1, \epsilon, this_reached, rr, mind/2, maxd/2)$
    **if** $br < this\_bound$ **then**
      $this\_bound \leftarrow br$
    **end if**
  **end if**
  **return** $this\_bound$

**Algorithm 4** max_relaxation$(p, q, \boldsymbol{lbs}, \boldsymbol{ubs}, s, relaxdom, \epsilon)$ computes an upper bound of $\max\{p(\boldsymbol{x}) \mid \boldsymbol{x} \in [\boldsymbol{lbs}, \boldsymbol{ubs}] \wedge q(\boldsymbol{x}) \leq 0\}$. To do this, it looks for a non negative real $y0$ such that the maximum of $p - y0 \times q$ is the smallest possible. An upper bound of $p - y0 \times c$ is returned. $s$ determines the number of values tried for $y0$ ($2^s + 1$ value equally spaced between 0 and $relaxdom$, including 0 and $relaxdom$) whereas $\epsilon$ determines the precision with which the bound is computed for each try for $y0$.

max_relaxation($p$ : multipoly, $q$ : multipoly, $\boldsymbol{lbs}$ : int tuple, $\boldsymbol{ubs}$ : int tuple, $s$ : int, $relaxdom$ : real, $\epsilon$ : real)
  $mind \leftarrow$ max_under_constraints$(q, \emptyset, \boldsymbol{lbs}, \boldsymbol{ubs}, s, \epsilon)$
  $maxd \leftarrow$ max_under_constraints$(-q, \emptyset, \boldsymbol{lbs}, \boldsymbol{ubs}, s, \epsilon)$
  $pyq \leftarrow p - y \times q$ ▷ $y$ is a fresh variable
  $pyq, \_, \boldsymbol{d} \leftarrow$ translate$(pyq, \emptyset, (0, \boldsymbol{lbs}), (relaxdom, \boldsymbol{ubs}))$ ▷ $y \in [0, relaxdom]$
  $rl, bl \leftarrow$ max_under_constraints_rec$(pyq[y \leftarrow 0], \emptyset, \boldsymbol{d}, s, \epsilon, -\infty)$
  $rr, br \leftarrow$ max_under_constraints_rec$(pyq[y \leftarrow 1], \emptyset, \boldsymbol{d}, s, \epsilon, -\infty)$
  **if** $bl < br$ **then**
    $bound \leftarrow bl$
  **else**
    $bound \leftarrow br$
  **end if**
  **if** $mind < 0 \wedge maxd > 0$ **then**
    $this\_bound \leftarrow$ max_relaxation_rec$(pyq, \mathbf{d}, s, 0, \epsilon, rl, rr, mind, maxd)$
  **end if**
  **return** min$(bound, this_bound)$

to the optimal value on a large part of the edge of the feasible space. Those cases can indeed require a high number of subdivisions to test the satisfiability of a constraint which can be avoided by the relaxation. Moreover, they can typically appear when analyzing reactive systems where a single iteration of the system induces relatively small changes. However, unlike the previous algorithm, it cannot guarantee the accuracy of the upper bound it returns.

## 15.3 Polynomial Template Abstract Domain

This section presents an abstract domain working on polynomial templates. The definition of the domain, the abstract operators and possible widenings are rather standard for template domains [SSM05]. The main interest lies in the possibility to precisely handle polynomial templates and assignments.

### 15.3.1 Lattice Structure

Given a set $P = \{p_1, \ldots, p_k\}$ of $k \in \mathbb{N}$ polynomials over $n$ variables, we consider the template domain $\mathcal{T}$ with templates $P$, as introduced in Definition 30, page 30.

**Example 57.** *Figure 15.3 presents an abstract value in $\mathcal{T}$.*

As can be noticed on Figure 15.3, the concretization of a value in $\mathcal{T}$ is a subset of $\mathbb{R}^n$ that is not necessarily convex. This is an uncommon feature since most numerical abstract domains are convex (there however exist a few non

$$\begin{array}{rcl} x & \leq & 1.001 \\ -x & \leq & 0 \\ y & \leq & 0.833 \\ -y & \leq & 0 \\ y - 6x^3 + 9x^2 - 3.2x & \leq & 0.5 \end{array}$$

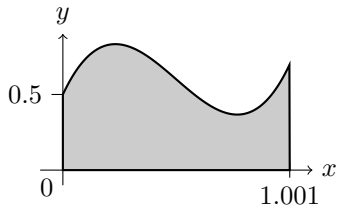



Figure 15.3 – Example of abstract value $(1, 0, 0.833, 0, 0.5) \in \mathcal{T}$ with $P = \{ x, -x, y, -y, y - 6x^3 + 9x^2 - 3.2x \}$. This value is an invariant at loop head for the program of Figure 15.1.

convex numerical abstract domains, for instance tropical polyhedra [AGG08] which are convex in tropical algebra but not in classical algebra).

### 15.3.2 Abstract Operators

To be able to perform abstract interpretation analyses with our new domain on programs, we now need to define abstract operators for guards, assignments and random assignments.

#### Guards

We define the abstract semantic of a guard $q \leq r$ on $\boldsymbol{b} \in \mathcal{T}$ with $q$ a $n$ variables polynomial in $\mathbb{R}[\boldsymbol{x}]$ and $r \in \mathbb{R}$ as:

$$[\![q \leq r]\!]^\sharp(\boldsymbol{b}) = \boldsymbol{b'} \qquad \text{with} \qquad b'_i = \mathrm{Opt}(p_i; p_1, \ldots, p_k, q)(b_1, \ldots, b_k, r)\,.$$

**Example 58.** *Still using the templates* $P = \{ x, -x, y, -y, y - 6x^3 + 9x^2 - 3.2x \}$, $[\![x \leq 1]\!]^\sharp(10, 0, 10, 0, 0.5) = (1, 0, 0.833, 0, 0.5)$. *This is illustrated on (k) and (l) in Figure 15.4.*

**Property 18** (soundness)**.** *This abstract operator is sound with respect to the concrete semantics of guards: for all polynomial* $q \in \mathbb{R}[\boldsymbol{x}]$*, all* $r \in \mathbb{R}$ *and all* $\boldsymbol{b} \in \mathcal{T}$*,* $[\![q \leq r]\!](\gamma_{\mathcal{T}}(\boldsymbol{b})) \subseteq \gamma_{\mathcal{T}}\left([\![q \leq r]\!]^\sharp(\boldsymbol{b})\right)$.

#### Assignments

We define the abstract semantic of an assignment $x_i := q$ on $\boldsymbol{b} \in \mathcal{T}$ with $q$ a $n$ variables polynomial in $\mathbb{R}[\boldsymbol{x}]$ as:

$$[\![x_i := q]\!]^\sharp(\boldsymbol{b}) = \boldsymbol{b'} \qquad \text{with} \qquad b'_j = \mathrm{Opt}(p_j[x_i \leftarrow q(\boldsymbol{x})]; p_1, \ldots, p_k)(\boldsymbol{b})\,.$$

**Example 59.** $[\![y := 0]\!]^\sharp(1.001, -0.001, 0, 0.002, 0.133) = (1.001, -0.001, 0, 0, 0.133)$. *This is illustrated on (n) and (o) in Figure 15.4.*

We can notice that this definition can (and in practice will) increase the degree of objective polynomials of the optimization problems to solve. This has an impact on the cost of solving these problems. This cost is then dependent on the composition of degrees of polynomials of the set of templates and polynomials appearing in the analyzed program.

**Property 19** (soundness)**.** *This abstract operator is sound with respect to the concrete semantics of assignments: for all variables $x_i$, for all polynomial $q \in \mathbb{R}[\boldsymbol{x}]$ and all $\boldsymbol{b} \in \mathcal{T}$, $[\![x := q]\!](\gamma_{\mathcal{T}}(\boldsymbol{b})) \subseteq \gamma_{\mathcal{T}}\left([\![x := q]\!]^\sharp(\boldsymbol{b})\right)$.*

#### Random assignments

We define the abstract semantic of a random assignment $x_i := ?(r_1, r_2)$ on $\boldsymbol{b} \in \mathcal{T}$ with $r_1, r_2 \in \mathbb{R}$ as:

$$[\![x_i := ?(r_1, r_2)]\!]^\sharp(\boldsymbol{b}) = \rho(\boldsymbol{b'}) \qquad \text{with} \qquad b'_i = \begin{cases} r2 & \text{if } p_i = x_i \\ -r1 & \text{if } p_i = -x_i \\ +\infty & \text{if } x_i \text{ appears in } p_i \\ b_i & \text{otherwise.} \end{cases}$$

where $\rho : \mathcal{T} \to \mathcal{T}$ is defined as: $\rho(\boldsymbol{b}) = \boldsymbol{b'}$ with $b'_i = \mathrm{Opt}(p_i; p_1, \ldots, p_k)(\boldsymbol{b})$.

**Example 60.** $[\![y := ?(0, 0.5)]\!]^\sharp(0, 0, +\infty, +\infty, +\infty) = (0, 0, 0.5, 0, 0.5)$. *This is illustrated on (b) and (c) in Figure 15.4.*

**Property 20** (soundness)**.** *This abstract operator is sound with respect to the concrete semantics of assignments: for all variables $x_i$, for all $r_1, r_2 \in \mathbb{R}$ and all $\boldsymbol{b} \in \mathcal{T}$, $[\![x := ?(r_1, r_2)]\!](\gamma_{\mathcal{T}}(\boldsymbol{b})) \subseteq \gamma_{\mathcal{T}}\left([\![x := ?(r_1, r_2)]\!]^\sharp(\boldsymbol{b})\right)$.*

### 15.3.3 Widening

The domain $\mathcal{T}$ has infinite ascending chains. A widening is then required to enforce termination of the analyses.

Assuming any widening $\nabla_s$ on lattice $\overline{\mathbb{R}}$ equipped with the usual order, its pointwise extension gives a widening on $\mathcal{T}$.

However, since the implementation with Bernstein polynomials definitely jumps to $\top$ as soon as one of the bounds $b_i$ with $p_i$ of the form $x_j$ or $-x_j$ is infinite, a widening not jumping directly to $+\infty$, such as a widening with thresholds, is actually required to get any practical result[4].

### 15.3.4 Implementation considerations

Optimization problems $\mathrm{Opt}(p; q_1, \ldots, q_k)(\boldsymbol{b})$ can be solved (within some accuracy $\epsilon$) with algorithms of Sections 15.2.3 or 15.2.4. However, those algorithms require each variable $x_i$ appearing in polynomials $p, q_1, \ldots, q_k$ to be bounded. That is, polynomials $x_i$ and $-x_i$ have to be among $q_1, \ldots, q_k$ and the bounds associated to them in tuple $\boldsymbol{b}$ must be in $\mathbb{R}$ (i.e. neither $-\infty$ nor $+\infty$). In case one of those bounds is $-\infty$, we just returns $-\infty$ which is the exact result. But in case $+\infty$, we have to conservatively overapproximate the result either by acting as the identity function in the case of guards[5] or by returning $+\infty$ in the case of assignments.

Parameter $\epsilon$ unfortunately plays a key role in the trade off between precision and cost of the analysis. A large value enables a faster analysis but may result

[4] The goal being more to avoid directly jumping to $+\infty$ rather than stopping on precise fixpoints, the choice of thresholds should not be critical. For instance, a sequence such as the $(10^n)_{n\in\mathbb{N}}$ is a reasonnable solution.

[5] Since guards do not modify any variable, the identity function is always a sound, although very coarse, overapproximation.

in a far less precise result if for instance we miss a fixpoint whereas a small value leads to more precise result but induces a more costly analysis. In contrast, parameters $s$ and $relaxdom$ of algorithm of Section 15.2.4, defining the domain in which the relaxation value is looked for, play a less critical role since this domain is rapidly cut down, preventing a large domain to induce much overhead.

Last but not least the choice of arithmetic for all computations performed by algorithms of Section 15.2 is critical for soundness of the analyses and can have an important impact on performances. Floating point arithmetic is definitely the fastest but is not an option since it doesn't guarantee soundness of the result. Rational arithmetic with arbitrary precision integers is sound but expensive. A workaround could be to compute potentially unsound results with floating point and check the final result with a rational implementation. There even exists such an implementation fully proved in the theorem prover PVS [MN12][6]. But finally, we found out that a good compromise is to use floating point interval arithmetic [Rum10]. This is only about two times slower than a pure floating point – unsound – implementation. Given the good numerical stability properties of Bernstein polynomials [Far12, §6], this works very well in practice.

It is also interesting to notice that those branch and bound algorithms should be rather easy to parallelize.

### 15.3.5 Example

**Example 61.** *Figure 15.4 displays an analysis on the running example (Figure 15.1), still with* $P = \{ x, -x, y, -y, y - 6x^3 + 9x^2 - 3.2x \}$ *and with a widening with thresholds* $\{ 10, 100, 1000 \}$. *A descending iteration from the fixpoint obtained then gives the invariant displayed on Figure 15.3 at loop head.*

*On this example, the template polynomial* $y - 6x^3 + 9x^2 - 3.2x$ *allows one to bound the variable* $y$ *by* $0.833$ *while an analysis with the polyhedra domain yields* $y \leq 3.5$. *This is rather tight since the actual maximal value can be proven by hand to be between* $0.788$ *and* $0.792$.

*It is interesting to notice that consecutive assignments are considered together (Figures 15.4e and 15.4m). This is an easy way to improve both the precision of the analysis (by avoiding intermediate abstraction, hence loss of precision[7]) and its cost (by avoiding redundant computations). An alternative solution would have been to use the mechanism proposed in [SSM05] to compute local templates in a similar way to a weakest precondition calculus [Dij75], but this offers only the precision and not the cost improvement of previous solution.*

*This analysis takes 1.8s with the algorithm of Section 15.2.3 and 0.2s with the alternative algorithm of Section 15.2.4, both implemented with floating point interval arithmetic and running on an Intel Core2 @ 2.66GHz.*

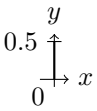
## 15.4 Related Work

Very few works address the analysis of nonlinear properties.

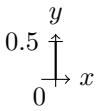
[6] The implementation presented in this paper only allows one to check the results of alternative algorithm of Section 15.2.4 but, according to personal communication with the authors, they more recently implemented a more general version of the algorithm of Section 15.2.3, allowing to also check the results of this algorithm.

[7] See for instance [GM11] for a more detailed discussion on this point.

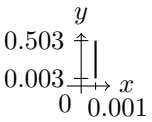

(a) $\top$




(b) $[\![x := 0]\!]^\sharp$(a) = $(0, 0, +\infty, +\infty, +\infty)$




(c) $[\![y := ?(0, 0.5)]\!]^\sharp$(b) = $(0, 0, 0.5, 0, 0.5)$

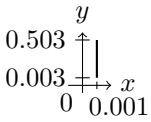



(d) $[\![x \leq 1]\!]^\sharp$(c) = $(0, 0, 0.5, 0, 0.5)$

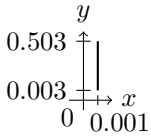
(e) $[\![y := y + 0.001(18x^2 - 18x+3); x := x+0.001]\!]^\sharp$(d) = $(0.001, -0.001, 0.503, -0.003, 0.5)$

(f) $[\![y \leq 0]\!]^\sharp$(e) = $\bot$

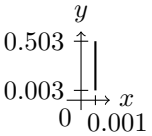
(g) $[\![y := 0]\!]^\sharp$(f) = $\bot$



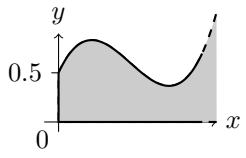

(h) $[\![y \geq 0]\!]^\sharp$(e) = (e)



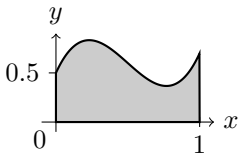

(i) $[\![y := y]\!]^\sharp$(h) = (e)



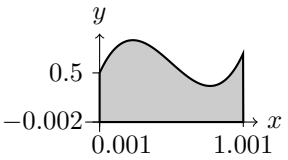

(j) (g) $\sqcup^\sharp$ (i) = (e)



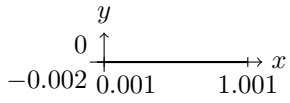

(k) (c) $\nabla$ (j) = $(10, 0, 10, 0, 0.5)$



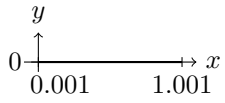

(l) $[\![x \leq 1]\!]^\sharp$(k) = $(1, 0, 0.833, 0, 0.5)$



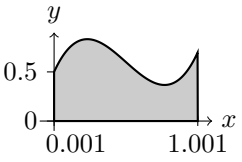

(m) $[\![y := y + 0.001(18x^2 - 18x + 3); x := x + 0.001]\!]^\sharp$(l) = $(1.001, -0.001, 0.833, 0.002, 0.5)$



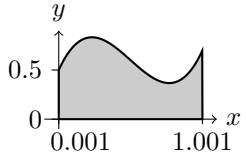

(n) $[\![y \leq 0]\!]^\sharp$(m) = $(1.001, -0.001, 0, 0.002, 0.133)$



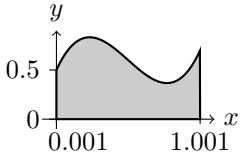

(o) $[\![y := 0]\!]^\sharp$(n) = $(1.001, -0.001, 0, 0, 0.133)$



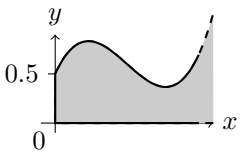

(p) $[\![y \geq 0]\!]^\sharp$(m) = $(1.001, -0.001, 0.833, 0, 0.5)$




(q) $[\![y := y]\!]^\sharp$(p) = (p)




(r) (o) $\sqcup^\sharp$ (q) = (p)




(s) (k) $\nabla$ (r) = (k)
(fixpoint reached!)

Figure 15.4 – Example of analysis on the running example (Figure 15.1).

The use of numerical optimization procedures in template domains was introduced by Sankaranarayanan, Sipma and Manna [SSM05]. They used linear programming to implement template polyhedra but already mentioned the possibility to use semi-definite programming to infer nonlinear invariants. Semi-definite programming along with policy iterations enables to infer quadratic invariants on linear programs as already discussed in Chapter 4 and in Part III of this document. This method is definitely more efficient than branch and bound algorithms with Bernstein polynomials on this common subclass of quadratic invariants. However, it does not deal with other properties than quadratic properties for linear systems.

Sankaranarayanan, Sipma and Manna [SSM04] as well as Rodriguez-Carbonell and Kapur [RCK04] use Gröbner bases to generate invariant polynomial equalities, hence a disjoint class of properties compared to the one considered in this chapter. Cachera, Jensen, Jobin and Kirchner [CJJK12] offer an alternative – more efficient – algorithm not using Gröbner bases but still targeting equalities.

Bagnara, Rodriguez-Carbonell and Zaffanella [BRCZ05] generate polynomial invariants. They only applied the technique to quadratic invariants but seem confident that some improvements could leverage it to at least cubic invariants. The huge advantage of this work compared to the one presented in this chapter is that it is fully automatic, not requiring polynomial templates to be given prior to any analysis. However, the techniques are of very different nature and it is expectable that each one is able to generate invariants which won't be found by the other.

This work is not the first one to make use of Bernstein polynomials in the scope of verification. Clauss and Tchoupaeva [CT04] already offered to use Bernstein polynomials to get linear approximations of polynomials on some interval for compilation purpose.

Finally Dang and Salinas [DS09] and Dang and Testylier [DT12] also used them for hybrid system analysis. A major difference with the present proposal is that, since they have to perform a very high number of iterations, they cannot afford branch and bound algorithms. That is why templates are limited to linear templates handled with some smart and efficient method.

## 15.5 Conclusion and Perspectives

This chapter proposed a polynomial template domain relying on Bernstein polynomials. To the best of authors' knowledge it is the first template abstract domain able to deal with arbitrary polynomial systems admitting polynomial invariants.

In the author's view, the main weakness of the current chapter is the need of templates to perform the analysis. In [RJGF12] we proposed a technique to synthesize meaningful quadratic templates. An identified future work would be to adapt this approach to general polynomial systems [KHJ13].

On the efficiency/precision issue, it would be very interesting to replace Kleene iterations by policy iterations. However, this does not look very tractable. An alternative solution could be to stick to Kleene iterations but to analyze code "en bloc". Some specific classes of polynomials could also be handled differently. For instance in the particular case of convex polynomials, convex

optimization would certainly be more efficient than branch and bound algorithms with Bernstein polynomials.

To conclude, this work is a first contribution using Bernstein polynomials within an abstract domain. Lot of directions are opened to increase the efficiency of this proposal or ease its application. An exciting future work enabled by this precise analysis of polynomial systems is the analysis of more general nonlinear functions that could be abstracted by polynomials through a Taylor or Poisson expansion.

# Part V

# Conclusion and Perspectives

The goal of this thesis was to analyze control command systems. Most of them, including the ones in flight commands of aircrafts, are linear or piecewise linear. It is well known from control theorists developing these controllers that they are stable if and only if they admit a quadratic invariant, that is an ellipsoid. However, most currently available static analysis techniques rely on linear invariants. To give worthy results on these systems, they usually require heavy unrolling, which can become problematic in presence of guards that commonly appear due to the addition of things like saturations or antiwindups to purely linear controllers. A few quadratic invariants inference methods do exist but are either restricted to a small class of systems or offer a limited level of automation, requiring suitable templates to be provided prior to the analysis.

The first two parts of this thesis (Parts II and III) offered a fully automatic method to infer quadratic invariants for purely linear systems and some piecewise linear systems with guards. This is achieved thanks to the *policy iteration* technique using semi-definite programming numerical solvers to compute *quadratic properties*. This technique had to cope with two serious drawbacks. First, policy iterations appeared rather orthogonal to classic abstract interpretation. Then, they required appropriate templates to be given prior to any analysis. Part III tries to address the first issue by integrating policy iterations in a classic abstract domain whereas Part II offers a template generation heuristic to answer the second one. This heuristic is based on ideas from control theory, namely *quadratic Lyapunov functions*.

Thus, starting from a source code, the implemented abstract domain first extracts a control flow graph of the part of this code targeted by the analysis (the remaining being abstracted through reduced products with other abstract domains such as the intervals domain). From this graph, a suitable quadratic template can then be automatically computed which allows policy iterations to compute an invariants. This invariant can eventually be projected to intervals for instance and shared with other domains through reduced products.

To the extent of author knowledge, our static analyzer, using the resulting abstract domain, is the only one able to fully automatically infer invariants on some of our benchmarks, which are furthermore often rather tight[8]. Moreover, the classic abstract domain interface should enable an easy integration in any static analyzer based on the abstract interpretation framework and easy cooperation with already existing abstract domains, through reduced products. Analysis times proved to remain reasonable on small benchmarks. Of course, this would not scale to huge systems but it might be possible to handle actual critical systems by focusing on such small systems composing them.

The numerical solvers used being implemented — for efficiency reasons — with floating point computations, they do not offer any formal guarantee of correctness about their results. An a posteriori verification was then developed which proved efficient on our benchmarks. A method to take into account floating point rounding errors in the analyzed programs was also sketched. Both aspects are sometimes omitted in some related work but are mandatory to get full confidence in the soundness of the synthesized invariants.

[8]It would nevertheless be worth comparing these results with those given by other state of the art static analyzers. For instance, the abstract interpreter Fluctuat (http://www.lix.polytechnique.fr/Labo/Sylvie.Putot/fluctuat.html) could return better results on some benchmarks thanks to its unrolling mechanism.

Regarding perspectives, the template generation heuristic offered considers various behaviors of the analyzed controller (saturated, not saturated, reset,...) independently. As shown by our benchmarks, this can sometime fails. Models of closed loop systems (i.e., including both the controller and the plant), constitute interesting cases for which this appears unsuitable, since plants are commonly unstable, leading to problematic behaviors as soon as a saturation is introduced. Future work should then address this issue. A better template generation heuristic could allow to perform closed loop stability proofs. However, there is another obstacle to such proofs since control theorist only possess a continuous model of the plant which has to be discretized, although this discretization can first be done by hand.

Furthermore, it could be interesting to study extensions of the method to other properties than stability such as performance and robustness. Although performance properties can look pretty different from stability properties, this is far less true for robustness. Indeed, a stability proof can be made with uncertainties on some parameters, becoming de facto a robustness proof.

It could also be nice to prove the most critical parts of our abstract domain using a theorem prover such as Coq. As already noticed, to establish soundness of the analysis, only the final checking part would have to be proven. In particular, there is nothing to prove about policy iterations or semi-definite programming. Thus, although large, the amount of work required could remain within reasonable bounds.

On a practical perspective, the abstract domain could be integrated in a more mature abstract interpreter such as IKOS[9]. It could also be more closely integrated in the SMT based model checker developed at ONERA as an invariant synthesizing oracle which could greatly help it by providing numerical invariants difficult to obtain through symbolic methods.

On a longer term and more theoretical view, the fruitful collaboration with control theory could be pursued, trying to import in static analysis notions such as common Lyapunov functions or piecewise linear quadratic Lyapunov functions [JR98] which could enable analysis of more complex systems.

Another direction would be to target more than linear (or piecewise linear) systems. To that end, the last part of the document (Part IV) offered a template abstract domain to infer polynomial invariants on polynomial programs. This is definitely a very prospective work. First, the method need to be better assessed by running more benchmarks. Then, the crucial question of the choice of templates remains to be addressed, maybe by looking at methods known from control theorists, based on sum of squares techniques [KHJ13]. Finally, other than polynomial functions might be studied through Taylor or Poisson expansions for instance.

[9] http://ti.arc.nasa.gov/opensource/ikos/

# Bibliography


[AFP09] Fernando Alegre, Éric Féron, and Santosh Pande. Using ellipsoidal domains to analyze control systems software. 2009. http://arxiv.org/abs/0909.1977.

[AGG08] Xavier Allamigeon, Stephane Gaubert, and Eric Goubault. Inferring min and max invariants using max-plus polyhedra. In *SAS*, pages 189–204, 2008.

[AGG10] Assalé Adjé, Stéphane Gaubert, and Éric Goubault. Coupling policy iteration with semi-definite relaxation to compute accurate numerical invariants in static analysis. In *ESOP*, pages 23–42, 2010.

[BCE+03] A. Benveniste, P. Caspi, S. Edwards, N. Halbwachs, P. Le Guernic, and R. De Simone. The synchronous languages 12 years later. In *Proceedings of The IEEE*, pages 64–83, 2003.

[BEEFB94] Stephen Boyd, Laurent El Ghaoui, Éric Féron, and Venkataramanan Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*, volume 15 of *SIAM*. SIAM, Philadelphia, PA, June 1994.

[Bie09] Armin Biere. Bounded model checking. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 14, pages 457–481. IOS Press, February 2009.

[Bou93] François Bourdoncle. Efficient chaotic iteration strategies with widenings. In Dines Bjørner, Manfred Broy, and Igor Pottosin, editors, *Formal Methods in Programming and Their Applications*, volume 735 of *LNCS*, pages 128–141. Springer Berlin / Heidelberg, 1993. 10.1007/BFb0039704.

[BRCZ05] Roberto Bagnara, Enric Rodríguez-Carbonell, and Enea Zaffanella. Generation of basic semi-algebraic invariants using convex polyhedra. In *SAS*, pages 19–34, 2005.

[BSC12] Olivier Bouissou, Yassamine Seladji, and Alexandre Chapoutot. Acceleration of the abstract fixpoint computation in numerical program analysis. *J. Symb. Comput.*, 47(12):1479–1511, 2012.

[BSST09] Clark Barrett, Roberto Sebastiani, Sanjit Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn J. H. Heule,

Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185, chapter 26, pages 825–885. IOS Press, February 2009.

[BV04] Stephen Boyd and Lieven Vandenberghe. *Convex optimization*. Cambridge university press, 2004.

[CC77] Patrick Cousot and Radhia Cousot. Abstract interpretation: A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *POPL*, pages 238–252, 1977.

[CC79] Patrick Cousot and Radhia Cousot. Systematic design of program analysis frameworks. In *POPL*, pages 269–282, 1979.

[CC92] Patrick Cousot and Radhia Cousot. Abstract interpretation frameworks. *J. Log. Comput.*, 2(4):511–547, 1992.

[CCF+06] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, David Monniaux, and Xavier Rival. Combination of abstractions in the Astrée static analyzer. In M. Okada and I. Satoh, editors, *ASIAN*, pages 272–300. Springer, 2006.

[CCF+09] Patrick Cousot, Radhia Cousot, Jérôme Feret, Laurent Mauborgne, Antoine Miné, and Xavier Rival. Why does astrée scale up? *Formal Methods in System Design*, 35(3):229–264, 2009.

[CGG+05] Alexandru Costan, Stephane Gaubert, Eric Goubault, Matthieu Martel, and Sylvie Putot. A policy iteration algorithm for computing fixed points in static analysis of programs. In *CAV*, pages 462–475, 2005.

[CH78] Patrick Cousot and Nicolas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL*, pages 84–96, 1978.

[CJJK12] David Cachera, Thomas P. Jensen, Arnaud Jobin, and Florent Kirchner. Inference of polynomial invariants for imperative programs: A farewell to Gröbner bases. In *SAS*, pages 58–74, 2012.

[Cou99] Patrick Cousot. The calculational design of a generic abstract interpreter. In M. Broy and R. Steinbrüggen, editors, *Calculational System Design*, NATO ASI Series F. IOS Press, Amsterdam, 1999.

[Cou05] Patrick Cousot. Proving program invariance and termination by parametric abstraction, lagrangian relaxation and semidefinite programming. In *VMCAI*, pages 1–24, 2005.

[CT04] Philippe Clauss and Irina Tchoupaeva. A symbolic approach to Bernstein expansion for program analysis and optimization. In *CC*, pages 120–133, 2004.

[Dij72] Edsger W. Dijkstra. The humble programmer. *Commun. ACM*, 15(10):859–866, 1972.

[Dij75] Edsger W. Dijkstra. Guarded commands, nondeterminacy and formal derivation of programs. *Commun. ACM*, 18(8):453–457, 1975.

[DS09] Thao Dang and David Salinas. Image computation for polynomial dynamical systems using the Bernstein expansion. In *CAV*, pages 219–232, 2009.

[DT12] Thao Dang and Romain Testylier. Reachability analysis using the Bernstein expansion over polyhedra. In *Numerical Software Verification*, 2012.

[Far12] Rida T. Farouki. The Bernstein polynomial basis: A centennial retrospective. *Computer Aided Geometric Design*, 29(6):379 – 419, 2012.

[FBG+] Eric Feron, Guillaume Brat, Pierre-Loic Garoche, Pete Manolios, and Marc Pantel. Formal methods for areospace applications. FM-CAD 2012 tutorial.

[Fer04] Jérôme Feret. Static analysis of digital filters. In *ESOP*, number 2986 in LNCS. Springer, 2004.

[Fer05] Jérôme Feret. Numerical abstract domains for digital filters. In *International workshop on Numerical and Symbolic Abstract Domains (NSAD)*, 2005.

[FG10] Paul Feautrier and Laure Gonnord. Accelerated invariant generation for c programs with aspic and c2fsm. *Electr. Notes Theor. Comput. Sci.*, 267(2):3–13, 2010.

[GGP09] Khalil Ghorbal, Eric Goubault, and Sylvie Putot. The zonotope abstract domain taylor1+. In *CAV*, pages 627–633, 2009.

[GGTZ07] Stephane Gaubert, Eric Goubault, Ankur Taly, and Sarah Zennou. Static analysis by policy iteration on relational domains. In *ESOP*, pages 237–252, 2007.

[GJS03] Jürgen Garloff, Christian Jansson, and Andrew P Smith. Lower bound functions for polynomials. *Journal of Computational and Applied Mathematics*, 157(1):207 – 225, 2003.

[GM11] Thomas Martin Gawlitza and David Monniaux. Improving strategies via smt solving. In *ESOP*, pages 236–255, 2011.

[GM12] Thomas Martin Gawlitza and David Monniaux. Invariant generation through strategy iteration in succinctly represented control flow graphs. *Logical Methods in Computer Science*, 8(3), 2012.

[GP11] Éric Goubault and Sylvie Putot. Static analysis of finite precision computations. In *VMCAI*, pages 232–247, 2011.

[GR06] Denis Gopan and Thomas W. Reps. Lookahead widening. In *CAV*, pages 452–466, 2006.

[Gra92] Philippe Granger. Improving the results of static analyses of programs by local decreasing iterations. In Rudrapatna Shyamasundar, editor, *Foundations of Software Technology and Theoretical Computer Science*, volume 652 of *Lecture Notes in Computer Science*, pages 68–79. Springer Berlin Heidelberg, 1992.

[GS07a] Thomas Gawlitza and Helmut Seidl. Precise fixpoint computation through strategy iteration. In *ESOP*, pages 300–315, 2007.

[GS07b] Thomas Gawlitza and Helmut Seidl. Precise relational invariants through strategy iteration. In *CSL*, pages 23–40, 2007.

[GS10] Thomas Martin Gawlitza and Helmut Seidl. Computing relaxed abstract semantics w.r.t. quadratic zones precisely. In *SAS*, pages 271–286, 2010.

[GSA+12] Thomas Martin Gawlitza, Helmut Seidl, Assalé Adjé, Stéphane Gaubert, and Eric Goubault. Abstract interpretation meets convex optimization. *J. Symb. Comput.*, 47(12):1416–1446, 2012.

[HC08] Wassim M. Haddad and Vijay S. Chellaboina. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2008.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous dataflow programming language lustre. *Proceedings of the IEEE*, 79(9):1305–1320, September 1991.

[HH12] Nicolas Halbwachs and Julien Henry. When the decreasing sequence fails. In *SAS*, pages 198–213, 2012.

[Hig96] Nicholas J. Higham. *Accuracy and Stability of Numerical Algorithms*. Society for Industrial and Applied Mathematics, Philadelphia, PA, USA, 1996.

[JCK07] Christian Jansson, Denis Chaykin, and Christian Keil. Rigorous error bounds for the optimal value in semidefinite programming. *SIAM J. Numerical Analysis*, 46(1):180–200, 2007.

[Jea00] Bertrand Jeannet. *Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones*. PhD thesis, Institut National Polytechnique de Grenoble, 2000.

[Jea03] Bertrand Jeannet. Dynamic partitioning in linear relation analysis: Application to the verification of reactive systems. *Formal Methods in System Design*, 23(1):5–37, 2003.

[Jea10] Bertrand Jeannet. Some experience on the software engineering of abstract interpretation tools. *Electr. Notes Theor. Comput. Sci.*, 267(2):29–42, 2010.

[JM09] Bertrand Jeannet and Antoine Miné. Apron: A library of numerical abstract domains for static analysis. In *CAV*, pages 661–667, 2009.

[JR98] Mikael Johansson and Anders Rantzer. Computation of piecewise quadratic lyapunov functions for hybrid systems. *Automatic Control, IEEE Transactions on*, 43(4):555–559, 1998.

[KHJ13] Milan Korda, Didier Henrion, and Colin N Jones. Convex computation of the maximum controlled invariant set for polynomial control systems. *arXiv preprint arXiv:1303.6469*, 2013.

[Kna28] Bronislaw Knaster. Un thórèm sur les fonctions d'ensembles. *Ann. Soc. Polon. Math.*, 6, 1928. with Alfred Tarski.

[Lya47] Aleksandr Mikhailovich Lyapunov. Problème général de la stabilité du mouvement. *Annals of Mathematics Studies*, 17, 1947.

[Min01] Antoine Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[Min04] Antoine Miné. Relational abstract domains for the detection of floating-point run-time errors. In *ESOP*, volume 2986 of *LNCS*, pages 3–17. Springer, 2004. http://www.di.ens.fr/~mine/publi/article-mine-esop04.pdf.

[MN12] César Muñoz and Anthony Narkawicz. Formalization of a representation of Bernstein polynomials and applications to global optimization. *Journal of Automated Reasoning*, 2012.

[Mon05] David Monniaux. Compositional analysis of floating-point linear numerical filters. In *CAV*, pages 199–212, 2005.

[Mon08] David Monniaux. The pitfalls of verifying floating-point computations. *ACM Trans. Program. Lang. Syst.*, 30(3), 2008.

[NA11] P.S.V. Nataraj and M. Arounassalame. Constrained global optimization of multivariate polynomials using Bernstein branch and prune algorithm. *Journal of Global Optimization*, 49:185–212, 2011.

[NDEG95] Ramine Nikoukhah, François Delebecque, and Laurent El Ghaoui. LMITOOL: a Package for LMI Optimization in Scilab User's Guide. Research Report RT-0170, INRIA, 1995.

[RCK04] Enric Rodríguez-Carbonell and Deepak Kapur. Automatic generation of polynomial loop invariants: Algebraic foundations. In *Proceedings of the 2004 international symposium on Symbolic and algebraic computation*, ISSAC, pages 266–273, New York, NY, USA, 2004. ACM.

[RFM05] Mardavij Roozbehani, Éric Féron, and Alexandre Megretski. Modeling, optimization and computation for software verification. In *HSCC*, pages 606–622, 2005.

[RG13a] Pierre Roux and Pierre-Loïc Garoche. Integrating policy iterations in abstract interpreters. In *ATVA*, 2013.

[RG13b] Pierre Roux and Pierre-Loïc Garoche. A polynomial template domain based on bernstein polynomials. In *NSV*, 2013.

[RJGF12] Pierre Roux, Romain Jobredeaux, Pierre-Loïc Garoche, and Éric Féron. A generic ellipsoid abstract domain for linear time invariant systems. In *HSCC*, pages 105–114, 2012.

[RN09] Shashwati Ray and P.S.V. Nataraj. An efficient algorithm for range computation of polynomials using the bernstein form. *Journal of Global Optimization*, 45(3):403–426, 2009.

[RTC92] RTCA. *Software Considerations in Airborne systems and Equipment Certification*, 1992.

[Rum06] Siegfried M. Rump. Verification of positive definiteness. *BIT Numerical Mathematics*, 46:433–452, 2006.

[Rum10] Siegfried M. Rump. Verification methods: Rigorous results using floating-point arithmetic. *Acta Numerica*, 19:287–449, May 2010.

[SB13] Yassamine Seladji and Olivier Bouissou. Numerical abstract domain using support functions. In *NFM*, 2013.

[SJ11] Peter Schrammel and Bertrand Jeannet. Logico-numerical abstract acceleration and application to the verification of data-flow programs. In *SAS*, pages 233–248, 2011.

[SJVG11] Pascal Sotin, Bertrand Jeannet, Franck Védrine, and Eric Goubault. Policy iteration within logico-numerical abstract domains. In *ATVA*, pages 290–305, 2011.

[Smi09] AndrewPaul Smith. Fast construction of constant bound functions for sparse polynomials. *Journal of Global Optimization*, 43(2-3):445–458, 2009.

[SSM04] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Non-linear loop invariant generation using Gröbner bases. In *Proceedings of the 31st ACM SIGPLAN-SIGACT symposium on Principles of programming languages*, POPL, pages 318–329, New York, NY, USA, 2004. ACM.

[SSM05] Sriram Sankaranarayanan, Henny B. Sipma, and Zohar Manna. Scalable analysis of linear systems using mathematical programming. In *VMCAI*, pages 25–41, 2005.

[Tar55] Alfred Tarski. A lattice-theoretical fixpoint theorem and its applications. *Pacific Journal of Mathematics*, 5(2), 1955.

[Tea] Scilab Team. Scilab. http://www.scilab.org.

[VB96] Lieven Vandenberghe and Stephen Boyd. Semidefinite programming. *SIAM Review*, 38(1):49–95, 1996.

[Win93] Glynn Winskel. *The formal semantics of programming languages - an introduction*. Foundation of computing series. MIT Press, 1993.

[Yan92] Qinping Yang. *Minimum Decay Rate of a Family of Dynamical Systems*. PhD thesis, Stanford University, 1992.

En 1987, l'Airbus A320 est le premier avion commercial à voler avec des commandes de vol numériques. Boeing suit la même voie moins de dix ans plus tard avec le premier vol de son B777 en 1994 et de nos jours la conception d'un grand avion de ligne ne s'envisage plus sans commandes de vol numériques. Elles commencent même à être introduites dans les avions d'affaire et les hélicoptères.

Les systèmes critiques comme les commandes de vol peuvent entraîner des désastres en cas de dysfonctionnement. Ils doivent donc subir une sévère certification qui est actuellement réalisée par du test intensif. On pourrait s'en satisfaire dans la mesure où après plusieurs décennies d'usage quotidien de milliers d'avions, aucune perte de vie humaine ne peut être imputée à un défaut logiciel dans un système de commande de vol numérique. De tels processus sont toutefois terriblement lourd et coûteux. De plus, le nombre de cas à tester étant bien trop grand, le test exhaustif est hors d'atteinte et il n'y a aucune garantie que le cas de test choisis parviennent à mettre en évidence tous les bugs. Tout cela explique l'intérêt porté à la fois par le monde industriel et académique aux méthodes de preuve formelle capable d'apporter, plus ou moins automatiquement, une preuve mathématique de correction. Parmi elles, cette thèse s'intéresse particulièrement à l'interprétation abstraite, une méthode efficace pour générer automatiquement des preuves de propriétés numériques qui sont essentielles dans notre contexte.

Il est bien connu des automaticiens que les contrôleurs linéaires sont stables si et seulement si ils admettent un invariant quadratique (un ellipsoïde, d'un point de vue géométrique). Ils les appellent fonction de Lyapunnov quadratique et une première partie propose d'en calculer automatiquement pour des contrôleurs donnés comme paire de matrices. Ceci est réalisé en utilisant des outils de programmation semi-définie. Les aspects virgule flottante sont pris en compte, que ce soit dans les calculs effectués par le programme analysé ou dans les outils utilisés pour l'analyse.

Toutefois, le véritable but est d'analyser des programmes implémentant des contrôleurs (et non des paires de matrices), incluant éventuellement des réinitialisation ou des saturations, donc non purement linéaires. L'itération sur les stratégies est une technique d'analyse statique récemment développée et bien adaptée à nos besoins. Toutefois, elle ne se marrie pas facilement avec les techniques classiques d'interprétation abstraite. La partie suivante propose une interface entre les deux mondes.

Enfin, la dernière partie est un travail plus préliminaire sur l'usage de l'optimisation globale sur des polynômes basée sur les polynômes de Bernstein pour calculer des invariants polynomiaux sur des programmes polynomiaux.

In 1987, the Airbus A320 performed its first flight, being the first commercial aircraft with digital flight commands. Boeing followed the same path less than ten years later with the maiden flight of its B777 in 1994 and nowadays the design of a large commercial aircraft cannot be envisioned anymore without digital flight commands. They even begin to be introduced in business jets and helicopters.

Critical Systems such as flight commands may have disastrous results in case of failure. Therefore, they have to meet stringent certification requirements. This is currently enforced by means of extensive testing. This can be considered successful in the sense that after decades of daily operations of thousands of aircrafts no loss of life can be imputed to a software failure in a numerical flight command system. Such process are nonetheless terribly heavy and expensive. Furthermore, the number of cases to test being way too large, exhaustive testing is intractable and there is no guarantee that the chosen test cases do not fail to expose a bug. All this explains the interest of both the industrial and the academic communities in formal methods able to more or less automatically deliver mathematical proof of correctness. Among them, this thesis will particularly focus on abstract interpretation, an efficient method to automatically generate proofs of numerical properties which are essential in our context.

It is well known from control theorists that linear controllers are stable if and only if they admit a quadratic invariant (geometrically speaking, an ellipsoid). They call these invariants quadratic Lyapunov functions and a first part offers to automatically compute such invariants for controllers given as a pair of matrices. This is done using semi-definite programming optimization tools. It is worth noting that floating point aspects are taken care of, whether they affect computations performed by the analyzed program or by the tools used for the analysis.

However, the actual goal is to analyze programs implementing controllers (and not pairs of matrices), potentially including resets or saturations, hence not purely linears. The policy iteration technique is a recently developed static analysis techniques well suited to that purpose. However, it does not marry very easily with the classic abstract interpretation paradigm. The next part tries to offer a nice interface between the two worlds.

Finally, the last part is a more prospective work on the use of polynomial global optimization based on Bernstein polynomials to compute polynomial invariants on polynomials systems.