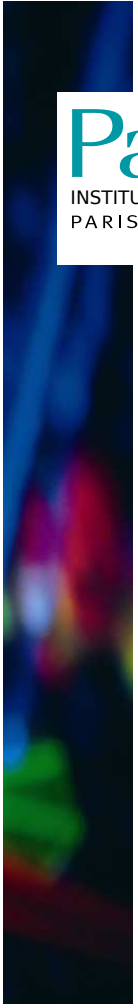


ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY

MINES ParisTech

École doctorale n°432 :
Sciences des Métiers de l'Ingénieur

Doctorat européen ParisTech

T H È S E

pour obtenir le grade de docteur délivré par

l'École nationale supérieure des mines de Paris

Spécialité « Informatique temps-réel, robotique et automatique »

présentée et soutenue publiquement par

Vivien MAISONNEUVE

le 6 février 2015

Analyse statique des systèmes de contrôle-commande
– Invariants entiers et flottants –

~ ~ ~

Static Analysis of Control-Command Systems
– Floating-Point and Integer Invariants –

Directeur de thèse : **François IRIGOIN**
Co-encadrement de la thèse : **Olivier HERMANT**

Jury

Nicolas HALBWACHS, Directeur de recherche, Verimag	Examinateur
Olivier HERMANT, Maître de recherche, CRI, MINES ParisTech	Examinateur
François IRIGOIN, Directeur de recherche, CRI, MINES ParisTech	Directeur de thèse
Matthieu MARTEL, Maître de conférences, DALI, Université de Perpignan	Rapporteur
Antoine MINÉ, Maître de conférences, DI, ENS Paris	Rapporteur
Philippe SCHNOEBELEN, Directeur de recherche, LSV, ENS Cachan	Président
Helmut SEIDL, Professeur, I2, Université technique de Munich	Rapporteur

T H È S E

MINES ParisTech
Centre de recherche en informatique
35 rue Saint-Honoré, 77305 Fontainebleau Cedex, France

Remerciements

Cette thèse a été préparée au Centre de recherche en informatique, mathématiques et systèmes de l'École nationale supérieure des mines de Paris (CRI, MINES ParisTech), sous la direction de François Irigoin et Olivier Hermant.

Je tiens d'abord à exprimer toute ma reconnaissance à mon directeur de thèse, François Irigoin, pour m'avoir accueilli au CRI et m'avoir proposé un sujet de thèse aussi riche et motivant, ainsi qu'à mon encadrant de thèse, Olivier Hermant, pour m'avoir suivi et guidé tout au long de ce travail. Leur aide, leurs conseils et leurs encouragements ont été inestimables.

Je remercie vivement les membres de mon jury de thèse. Je remercie mes rapporteurs, Matthieu Martel, Antoine Miné et Helmut Seidl, pour avoir accepté d'évaluer mon travail ; leurs commentaires ont été précieux pour améliorer ce manuscrit. Je remercie Philippe Schnoebelen, qui a été mon tuteur lors de mes études à l'ENS Cachan et qui m'a fait l'honneur de présider le jury. Je remercie aussi Nicolas Halbwachs, qui a accepté mon invitation à participer au jury.

Merci à Albert Cohen, Paul Feautrier, Laure Gonnord, Nicolas Petit et Sven Verdoolaege, pour les discussions que nous avons eues pendant ma thèse et l'aide qu'ils m'ont apportée. Merci également à Thierry Porcher et Philippe Ravier, de la société SILKAN, pour avoir mis à ma disposition le compilateur COLD.

Je remercie aussi chaleureusement toutes les personnes que j'ai eu le plaisir de côtoyer au CRI : Jacqueline Altimira, Mehdi Amini, Corinne Ancourt, Karim Barkati, Pierre Beauguitte, Danielle Bolan, Catherine Le Caër, Fabien Coelho, Laurent Daverio, Wajdi Farhani, Imré Frotier de la Messelière, Emilio Jesús Gallego Arias, Florian Gouin, Pierre Guillou, Pierre Jouvelot, Dounia Khaldi, Nelson Lossing, Robert Mahl, Claire Medrala, Amira Mensi, Benoît Pin, Antoniu Pop, José Afonso Sanches, Ronan Saillard, Arnaud Spiwack, Claude Tadonki, Karel De Vogeleer, Haisheng Wang et Pierre Wargnier.

Enfin, un grand merci aux membres de ma famille et à mes amis, qui m'ont aidé et encouragé tout au long de cette thèse.

Contents

Introduction

(English version follows.)

Un système critique pour la vie ou pour la sécurité est un système dont la défaillance ou le dysfonctionnement peut grièvement blesser ou tuer des gens, détruire ou faire subir des dégâts importants à des équipements, ou causer des dommages environnementaux. Les méthodes et les outils de l'ingénierie de la sécurité permettent d'appréhender les risques de cette nature. Un système critique *est conçu pour perdre moins d'une vie par milliard (10^9) d'heures de fonctionnement [Fed88]. Sa conception fait général appel à l'évaluation probabiliste des risques, une méthode qui combine l'analyse des modes de défaillance, de leurs effets et de leur criticité (AMDEC) avec l'analyse par arbre de défaillance. Aujourd'hui, les systèmes de sécurité critiques sont de plus en plus souvent informatisés.*

L'ingénierie logicielle pour les systèmes critiques est particulièrement difficile. Elle repose sur plusieurs éléments. Le premier est le génie et la gestion des procédés. Le second est le choix des outils et d'un environnement appropriés pour le système. Cela permet au développeur de tester efficacement le système par simulation et d'observer son efficacité. Le troisième consiste à se soumettre à toutes les exigences légales et réglementaires de sécurité, telles que les exigences de la FAA pour l'aviation (DO-178B [RTC92, Joh98]). La définition d'une norme selon laquelle doit être développé un système contraint les concepteurs à respecter ces exigences. L'industrie aéronautique a réussi à produire des normes standard pour la production de logiciels d'aviation critiques. Des normes similaires existent dans l'industrie automobile (MISRA C [MIS98], ISO 26262 [KTDA12]), médicale (IEC 62304 [Jor06]) et nucléaire (IEC 61513). L'approche standard consiste à soigneusement coder, inspecter, documenter, tester, vérifier et analyser le système.

Correctement appliquée, cette approche permet d'atteindre un très haut niveau de sécurité : des logiciels critiques ont été utilisés durant des décennies dans l'industrie aérospatiale sans causer de désastre majeur. Malheureusement, ces procédés de développements sont excessivement lourds et coûteux : la vérification et la validation des logiciels critiques représentent ainsi plus de 50% du coût de son développement [WDD+12]. En raison de leur complexité et de leur coût, ces contraintes sont parfois mal respectées, ce qui conduit à des bogues avec des conséquences parfois dramatiques. Un exemple récent est le défaut du contrôleur électronique de la commande d'accélération des véhicules Toyota Camry, qui a causé plusieurs morts [Yos13]. De plus, les tests seuls ne sont pas suffisants : étant donné le nombre considérable de scénarios de tests, il est impossible de couvrir tous les cas et il n'y a aucune garantie qu'un bogue n'ait pas pu échapper aux tests.

Ces éléments expliquent l'intérêt académique et industriel actuel pour les méthodes formelles, qui fournissent des garanties mathématiques de correction, avec différents niveaux d'automatisation. Une approche possible est de certifier un système de production et un compilateur, puis de générer le code du système à partir des spécifications. Une autre consiste à utiliser des méthodes formelles pour générer les preuves qui garantissent que le code répond à sa spécification.

Contenu et contributions

Ce manuscrit est organisé en deux parties, chacune d'elles correspondant à une contribution distincte. La première partie porte sur les systèmes de contrôle. *Un système de contrôle est un dispositif ou un ensemble de dispositifs qui gère, commande, dirige ou régule le comportement d'autres dispositifs ou systèmes (chapitre 1). Des systèmes de contrôle industriels sont utilisés en production, pour contrôler des équipements ou des machines. La propriété formelle qui nous intéresse est la* stabilité *: en termes simples, c'est la propriété selon laquelle toutes les solutions d'un système dynamique qui commencent à proximité d'un point d'équilibre restent par la suite dans son voisinage. La théorie du contrôle énonce qu'un contrôleur linéaire est stable si et seulement si il admet une invariant quadratique — géométriquement, un ellipsoïde, appelé* fonction quadratique de Lyapunov *(chapitre 2), et propose des méthodes pour trouver un invariant de cette nature à partir de la description algébrique du système. Nous présentons un framework formel, générique, pour traduire des preuves de stabilité de Lyapunov codées en MATLAB avec des nombres réels vers du code C de bas niveau en arithmétique machine (chapitre 3) ; ainsi qu'un outil pour générer automatiquement les invariants de stabilité sur de l'arithmétique à virgule flottante (chapitre 4).*

*Il arrive aussi qu'il soit nécessaire de reconstruire automatiquement les invariants sur un programme donné. C'est pourquoi la seconde partie porte sur l'*analyse des relations linéaires *(chapitres 5, 6, 7), un framework d'interprétation abstraite couramment utilisé, consacré à la génération automatique d'invariants sous forme d'inégalités linéaires sur les variables numériques d'un programme. Nous présentons ALICe, un ensemble d'outils pour comparer les techniques de calcul automatique d'invariants affines sur les boucles (chapitre 8). Il s'accompagne d'un benchmark construit à partir de 102 cas de test provenant de la littérature sur les problèmes d'invariants et de terminaison de boucles, et s'interface avec trois outils d'analyse de programmes, utilisant des algorithmes différents : Aspic, iscc et PIPS. Les résultats expérimentaux montrent l'importance du codage du modèle et les performances en retrait de PIPS en présence de boucles concurrentes dans le programme analysé. Pour étudier ces problèmes, nous utilisons deux techniques de restructuration de programme (chapitre 9), prouvées correctes en Coq, et nous présentons plusieurs améliorations apportées aux algorithmes de PIPS pour le calcul d'invariant (chapitre 10).*

* * *

A life-critical system or safety-critical system is a system whose failure or malfunction may result in death or serious injury to people, loss of or severe damage to equipment or environmental harm. Risks of this sort are usually managed with the methods and tools of safety engineering. A *critical system* is designed to lose less than one life per billion (10^9) hours of operation [Fed88]. Typical design methods include probabilistic risk assessment, a method that combines failure mode and effects analysis (FMEA) with fault-tree analysis. Safety-critical systems are increasingly computer-based.

Software engineering for critical systems is particularly difficult. There are three aspects which can be carried out to support this discipline. First is process engineering and management. Secondly, selecting the appropriate tools and environment for the system. This allows the system developer to effectively test the system by emulation and observe its effectiveness. Thirdly, address any legal and regulatory requirements, such as FAA requirements for aviation (DO-178B [RTC92, Joh98]). By setting a standard for which a system is required to be developed under, it compels the designers to stick to the requirements. The avionics industry has succeeded in producing standard methods for producing critical avionics software. Similar standards exist for automotive (MISRA C [MIS98], ISO 26262 [KTDA12]), medical (IEC 62304 [Jor06]) and nuclear (IEC 61513) industries. The standard approach is to carefully code, inspect, document, test, verify and analyze the system.

Correctly applied, this approach allows to achieve a very high level of safety: critical software has been used for decades in the aerospace industry without any major disaster. Unfortunately, such development processes are excessively heavy and expensive: verification and validation for critical software represent more than 50% of software development costs [WDD+12]. Due to their complexity and cost, these constraints are sometimes poorly respected, which leads to bugs with dramatic consequences. A recent example is the electronic throttle control defect linked to unintended acceleration in Toyota Camry vehicles [Yos13] that caused deaths. Furthermore, testing alone is not enough: considering the huge number of test scenarios, exhaustive testing is impossible and there is not any guarantee that the processed test cases did not fail to reveal a specific bug.

These elements explain the current academic and industrial interest in formal methods; they provide mathematical insurances of correctness, with various levels of automation. One possible approach is to certify a production system, a compiler, and then generate the system's code from specifications. Another one is to use formal methods to generate proofs that the code meets requirements.

This thesis focuses on *model checking* and *abstract interpretation* (with a secondary use of proof assistants), an efficient family of formal methods to exhaustively and automatically check whether a model meets a given specification, given in the form of numerical properties. More precisely, these techniques compute *invariants* on a program, i.e. conditions on its states that are true during its execution, or during some portion of it. They are logical assertions that hold true during a certain phase of execution, and they are used to check whether the specification holds. Therefore, generating accurate invariants is a key element to prove the correctness of a program with respect to its specification.

Contents and Contributions

This manuscript in organized in two parts, each of them corresponding to distinct contributions. The first part is about *control systems*. A control system is a device, or set of devices, that manages, commands, directs or regulates the behavior of other devices or systems (Chapter 1). Industrial control systems are used in industrial production for controlling an equipment or a machine. The formal property we are interested in is *stability*: in simple terms, that all solutions of a dynamical system that start out near an equilibrium point stay in its neighborhood forever. Control theory states that a linear controller is stable if and only if it admits a quadratic invariant — geometrically speaking, an ellipsoid, called *quadratic Lyapunov functions* (Chapter 2), and offers practical ways to find such an invariant from the algebraic description of the system. We present a formalized, generic framework to translate Lyapunov stability proofs on MATLAB real-valued code to low-level C code using machine arithmetic (Chapter 3), and a tool that automatically generates stability invariants in the case of floating-point arithmetic (Chapter 4).

It is also sometimes necessary to automatically reconstruct the invariants on given implementations. This is why the second part focuses on *linear relation analysis* (Chapters 5, 6, 7), a widespread abstract interpretation framework devoted to the automatic discovery of invariant linear inequalities on the numerical variables of a program. We present ALICe, a toolset to compare automatic computation techniques of affine loop scalar invariants (Chapter 8). It comes with a benchmark that we built using 102 test cases found in the loop invariant and termination bibliography, and interfaces with three analysis programs, that rely on different techniques: Aspic, iscc and PIPS. Experimental results show the importance of model coding and the poor performances of PIPS on concurrent loops. To tackle these issues, we use two restructuring techniques whose correctness is proved in Coq (Chapter 9), and present various improvements made to PIPS algorithms for invariant computation (Chapter 10).

Part I

Linear Systems

Chapter 1

Control-Command Critical Systems

Un système dynymatique *est un concept en mathématiques où une règle fixe décrit la dépendance temporelle d'un point dans un espace géométrique. Les modèles mathématiques qui décrivent le mouvement du balancier d'une horloge, l'écoulement de l'eau dans un tuyau ou le nombre de poisson chaque été dans un lac sont des exemples de systèmes dynamiques.*

À tout instant, le système possède un état *défini par un ensemble de nombres réels (un vecteur), qui peut être représenté par un point dans l'*espace d'état *approprié. Une légere altération de l'état corespond à un légere altération de ces valeurs. La règle d'évolution du système dynamique est une règle fixe qui décrit quel état futur peut découler de l'état actuel. Cette règle est déterministe : dans un intervalle de temps donné, l'état actuel ne peut mener qu'à un seul état futur.*

La théorie du contrôle *(ou* automatique*) est la branche interdisciplinaire de l'ingénierie et des mathématiques qui traite du comportement des systèmes dynamiques avec des entrées. L'entrée extérieure d'un système est appelée la* référence *(ou* consigne*). Quand une ou plusieurs variables de sortie d'un système doivent suivre une certaine consigne au cours du temps, le* contrôleur *manipule l'entrée du système de manière à obtenir l'état souhaité en sortie. Il s'agit d'un dispositif, historiquement mécanique, hydraulique, pneumatique et/ou électronique, plus récemment un microprocesseur ou un microcontrôleur, qui surveille et modifie les conditions de fonctionnement du système dynamique. La régulation de la température, de la pression, du débit ou de la vitesse sont des applications typiques des contrôleurs. Le régulateur à boules sur la figure 1.1 est l'un des premiers exemples de contrôleur mécanique, que l'on rencontre souvent sur les moteurs à vapeur : il contrôle la vitesse du moteur en régulant la quantité de vapeur admise, afin de maintenir une vitesse quasi-constante, indépendamment de la charge ou de la fourniture en combustible.*

Les sujets abordés en théorie du contrôle incluent la stabilité, la contrôlabilité et l'observabilité, la spécification du contrôle, l'idenfication et la robustesse du modèle. Parmi ceux-ci, un objectif courant est de calculer les solutions pour les actions de correction du contrôleur qui garantissent la stabilité du système, c'est-à-dire que le système restera sur le point qui aura été défini au lieu d'osciller autour de ce point. Une définition précise de la stabilité est donnée dans la section 1.3.

Bien que le champ d'application principal de l'automatique soit l'ingénierie des systèmes de contrôle, qui traite de la conception de systèmes de contrôle industriels, d'autres applications dépassent largement de ce cadre. L'automatique peut être utilisée pour étudier tout système avec rétroaction et trouve des applications dans des domaines aussi divers que la physiologie, l'életronique, la climatologie, les écosystèmes, la navigation, les réseaux de neurones ou encore la génétique.

* * *

A *dynamical system* is a concept in mathematics where a fixed rule describes the time dependency of a point in a geometrical space. Examples include the mathematical models that describe the swinging of a clock pendulum, the flow of water in a pipe, or the number of fish each springtime in a lake.

At any given time a dynamical system has a *state* defined by a set of real numbers (a vector) that can be represented by a point in an appropriate *state space* (a geometrical manifold). Small changes in the state of the system are represented by small changes in the numbers. The evolution rule of the dynamical system is a fixed rule that describes which future state can follow from the current state. The rule is deterministic; in other words, for a given time interval only one future state follows from the current state.

Control theory is the interdisciplinary branch of engineering and mathematics that deals with the behavior of dynamical systems with inputs. The external input of a system is called the *reference*. When one or more output variables of a system need to follow a certain reference over time, a *controller* manipulates the input to a system to obtain the desired output state of the system. It is a device, historically using mechanical, hydraulic, pneumatic or electronic techniques often in combination, but more recently in the form of a microprocessor or microcontroller, which monitors and physically alters the operating conditions of a given dynamical system. Typical applications of controllers are to hold settings for temperature, pressure, flow or speed. The centrifugal governor in Figure 1.1 is an early example of mechanical controller, commonly seen on steam engines, that controls the speed of an engine by regulating the amount of steam admitted, so as to maintain a near-constant speed, irrespective of the load or fuel-supply conditions.

Figure 1.1: Centrifugal governor at the Science Museum of London

Topics in control theory include stability, controllability and observability, control specification, model identification and robustness. Among them, an usual objective is to calculate solutions for a proper corrective action from the controller that result in system stability, that is, the system will hold the set point and not oscillate around it. A precise definition of system stability is given in Section 1.3.

Although a major application of control theory is control systems engineering, which deals with the design of process control systems for industry, other applications range far beyond this. As the general theory of feedback systems, control theory is useful wherever feedback occurs; a few examples are in physiology, electronics, climate modeling, machine design, ecosystems, navigation, neural networks, and gene expression.

1.1 Control-Command Systems

A *control-command systems* is an autonomous, self-controlled system that includes:

- The regulated parameter.

- The controller.
- *Sensors* to feed input data to the controller.
- *Actuators* through which the controller acts on the system to reach the desired state.

A complex example of control-command system is a modern airliner: parameters like the plane's altitude and speed are regulated by an on-board computer, using a set of sensors (speedometer, altimeter…), and adjusting the flight path by playing on actuators like engine thrust and flight control surfaces.

Plant Modeling

To design a control-command system, control engineers need to know precisely the effect actuators have on the system. For a physical system, it is required to design a model of the system behavior in its environment, using laws of mechanics and physics. This model is called a *plant*.

Linear systems are a particular class of control-command systems that can be described by linear equations (linear differential equations or difference equations). They are much easier to study than the general, nonlinear cases. Therefore, a nonlinear system is often linearized around an equilibrium point or trajectory (when possible), to obtain a linear system that accurately represents the nonlinear system in the neighborhood of this equilibrium point or trajectory.

From a mathematical perspective, the internal state $x_p \in \mathbb{R}^n$ of a linear plant (a vector containing the system characteristics in the environment: for a plane, it could be the speed, altitude, etc.) is ruled by a differential equation

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

where $A_p \in \mathbb{R}^{n \times n}$ and $B_p \in \mathbb{R}^{n \times p}$ are constant, real-valued matrices, and $u_p \in \mathbb{R}^p$ is the vector of system inputs (speed, altitude…).

In the setting defined by the internal state x_p and the input values u_p, the output y_p (interpreted as the system action) follows the equation

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

where C_p, D_p are constant, real-valued matrices.

Controller Design

Control theory offers various methods to develop a controller corresponding to a particular plant model. In the usual case of a linear controller, the controller internal state x_c follows the equation

$$\dot{x}_c = A_c x_c + B_c u_c \tag{1.1}$$

where u_c is the controller input, that typically comes from input sensors y_p (speed and altitude measurements) and commands y_d (cockpit controls), and equals the output of the physical system, minus the command $(y_p - y_d)$. The controller computes the output

$$y_c = C_c x_c + D_c u_c \tag{1.2}$$

that is applied to actuators and is equal to the plant input u_p (Figure 1.3).

Controller Implementation

Once the controller is designed, they are several ways to implement it:

- Hardware implementation: either with a purely mechanic implementation — as is the centrifugal governor in Figure 1.1 — or with analog electronics.
- Software implementation: running a program on a numerical controller, as on most modern devices.

In this document, we focus exclusively on software controllers. To implement a controller, it is necessary to discretize its internal state equation (1.1), in order to be able to recompute its internal state x_c at a fixed frequency:

$$x_{c_{k+1}} = A'_c x_{c_k} + B'_c u_{c_k}. \tag{1.3}$$

Typical frequencies for embedded systems are about 100 Hz.

Eventually, the discrete controller equations (1.2), (1.3) can be coded, for instance in MATLAB:

```
Ac = [0.4990, -0.0500; 0.0100, 1.0000];
Bc = [1; 0];
Cc = [564.48, 0];
Dc = -1280;
xc = zeros(2, 1);
while (1)
    receive(uc);
    xc = Ac*xc + Bc*uc;
    yc = Cc*xc + Dc*yc;
    send(yc);
    wait_next_tick();
end
```

Listing 1.1: A toy linear controller by Feron [Fer10]

This is a specific, numerical code, consisting of a main loop and nested computations; it looks very different from the source code of a compiler for example. Of course, the true controller of a large scale system is something much more complex: real-scale software may require hundreds of thousands of lines of C code (about 170,000 source lines of generated C code for an Airbus A340, and 760,000 lines for an Airbus A380).

1.2 Open-Loop and Closed-Loop Controllers

An *open-loop controller* (also called a non-feedback controller), is a type of controller that computes its input into a system using only the current state and its model of the system.

A characteristic of the open-loop controller is that it does not use feedback to determine if its output has achieved the desired goal of the input. This means that the system does not observe the output of the processes that it is controlling. Consequently, a true open-loop system may not compensate for disturbances in the system.

An open-loop controller is shown in Figure 1.2.

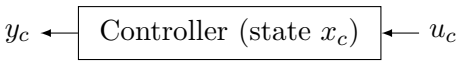


Figure 1.2: Open-loop controller

To overcome the limitations of the open-loop controller, control theory introduces feedback. A *closed-loop controller* uses feedback to control states or outputs of a dynamical system. Its name comes from the information path in the system: process inputs (e.g., fuel flow in the jet engines) have an effect on the process outputs (e.g., engine thrust), which is measured with sensors and processed by the controller; the result (the control signal) is "fed back" as input to the process, closing the loop.

Closed-loop controllers have the following advantages over open-loop controllers:

- Disturbance rejection.
- Guaranteed performance even with model uncertainties, when the model structure does not match perfectly the real process and the model parameters are not exact, unstable processes can still be stabilized.
- Reduced sensitivity to parameter variations.
- Improved reference tracking performance.

A closed-loop controller is shown in Figure 1.3.

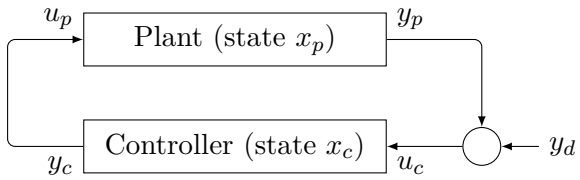


Figure 1.3: Closed-loop stability

1.3 Stability Proofs

Stability theory addresses the stability of solutions of differential equations and of trajectories of dynamical systems under small perturbations of initial conditions.

When focusing on stability, two types of properties are of major interest: closed-loop stability and open-loop stability.

- *Closed-loop stability* expresses that the controller manages to stabilize the plant, assuming that the physical system is correctly described by the model: if the command y_d is bounded, then the internal states x_p and x_c and the outputs y_p and y_c of the plant and the controller remain bounded.
- *Open-loop stability* means that all variables in the controller are bounded, assuming its inputs are bounded: if the controller input u_c is bounded, its internal state x_c and its output y_c remain bounded.

Both properties are interesting: closed-loop stability offers guarantees on the plant state, whereas an open-loop stable controller is ensured not to diverge disastrously if there is a temporary malfunction in its input sensors. This is why both open-loop and closed-loop stability are often required simultaneously in critical systems. Both types of stability are considered in this dissertation.

Chapter 2

Lyapunov Stability Proofs

La stabilité constitue un attribut essentiel des systèmes de contrôle, en particulier lorsque la sécurité des personnes est en jeu, comme c'est le cas par exemple en médecine ou en aéronautique (chapitre 1). Les développements récents, comme les technologies de contrôle adaptatif, utilisent principalement des critères de robustesse, de stabilité et de performance pour justifier leur pertinence dans des applications de contrôle critiques. Il existe de nombreux théorèmes, motivés par de telles applications, qui assurent la stabilité et la performance d'un système sous diverses hypothèses et dans divers contextes. La théorie de stabilité de Lyapunov [Lya92] joue à cet égard un rôle essentiel.

L'implémentation logicielle de bas niveau d'une loi de contrôle peut être inspectée par les outils d'analyse disponible pour aider au développement de programmes informatiques critiques. La technique d'analyse de programme la plus simple consiste à effectuer plusieurs simulations, en incluant dans certains cas une représentation logicielle ou matérielle du système asservi dans la boucle. Cependant, les simulations ne peuvent fournir d'information que sur un nombre limité de comportements du système. Les méthodes plus avancées comprennent le model checking et l'interprétation abstraite, par exemple en utilisant Astrée [CCF+15]. Ces méthodes prennent en entrée les programmes informatiques eux-mêmes et renvoient en sortie des certificats garantissant le bon comportement des programmes par rapport aux spécifications choisies. Une autre possibilité est d'utiliser des techniques de démonstration assistée de théorèmes, proposées par des outils comme Coq, Isabelle ou PVS [INR15, PNW14, Com15]. Ces assistants de preuve peuvent être utilisés pour établir des propriétés de programmes et plus généralement des constructions mathématiques. Le model checking, l'interprétation abstraite et les assistants de preuve sont tous trois utilisés pour vérifier des applications critiques.

Dans [Fer10], Feron étudie comment des informations de domaine sur les système de contrôle — et, en particulier, les preuves de stabilité de Lyapunov sur une modélisation théorique de haut niveau d'un système — peuvent être transposées vers des certificats de comportement de logiciels de contrôle lisibles et vérifiables par un ordinateur, en s'appuyant sur le système de preuve de Floyd et Hoare [Pel01], appliqué à du pseudo-code MATLAB (voir les sections 2.2 et 2.3). Feron présente des implémentations en boucle ouverte et en boucle fermée de son contrôleur, selon que l'action du système sur son environnement et la rétroaction sont modélisées ou non. Mais les erreurs causées par l'arithmétique à virgule flottante ne sont pas considérées.

Dans cette partie, nous abordons cette question en présentant une approche automatique pour traduire des invariants de preuves de stabilité de Lyapunov sur du pseudo-code à arithmétique réelle, comme celui fourni dans l'article de Feron, vers des invariants similaires sur du code machine en prenant en compte les erreurs d'arrondi introduites par l'arithmétique à virgule flottante. Nous utilisons ces invariants pour vérifier si les conditions de stabilité sont toujours respectées, auquel cas la stabilité du système à virgule flottante est établie. Cette approche est basée sur un framework théorique générique, décrit plus loin dans ce document, dont la sûreté

est prouvée en Coq ; elle est également implentée sous la forme d'un outil automatique.

Cette partie de la thèse est organisée comme suit. Dans le reste de ce chapitre, nous décrivons l'exemple de système dynamique du second ordre utilisé par Feron dans [Fer10], avec le contrôleur correspondant. Nous présentons ensuite l'analyse par Feron du contrôleur en boucle ouverte avec des nombres réels. Dans les chapitres qui suivent, nous présentons notre schéma générique de traduction (chapitre 3, page 29) et son implémentation dans une bibliothèque Python, et nous l'utilisons par la suite pour réécrire l'analyse de Feron en arithmétique à virgule flottante (chapitre 4, page 37). Enfin, on traitera le cas du système en boucle fermée. Cette partie se termine par une discussion sur la généralité de cette approche.

* * *

Stability constitutes an essential attribute of control systems, especially when human safety is involved, as in medical or aeronautical domains (Chapter 1). Modern system developments, such as adaptive control technologies, rely on robust stability and performance criteria as the primary justification for their relevance to safety-critical control applications. Motivated by such applications, there exist many theorems that ensure system stability and performance under various assumptions and in various settings. Lyapunov's stability theory [Lya92] plays a critical role in that regard.

The low-level software implementation of a control law can be inspected by analysis tools available to support the development of safety-critical computer programs. The simplest program analysis technique consists in performing several simulations, sometimes including a software or hardware representation of the controlled system in the loop. However, simulations provide information about only a finite number of system behaviors. More advanced methods include model checking and abstract interpretation, e.g. using Astrée [CCF+15]. In these methods, inputs are the very computer programs and outputs are certificates of proper program behavior along the chosen criterion. Another possibility is to use theorem-proving techniques, supported by tools such as Coq, Isabelle or PVS [INR15, PNW14, Com15]. These proof assistants can be used to establish properties of programs and more general mathematical constructs. Model checking, abstract interpretation, and theorem-proving tools are all used to verify safety-critical applications.

In [Fer10], Feron investigates how control-system domain knowledge and, in particular, Lyapunov-theoretic proofs of stability for the high-level theoretical modeling, can be migrated towards computer-readable and verifiable certificates of control software behavior by relying on Floyd's and Hoare's proof system [Pel01], applied to MATLAB pseudo-code (see Sections 2.2 and 2.3). Feron presents both open-loop and closed-loop implementations of his controller, i.e. depending whether the system action on the environment and its feedback are modeled or not. But errors resulting from the use of floating-point arithmetic are not considered.

In this part, we address this issue by presenting an automatic approach to translate Lyapunov-theoretic stability proof invariants on pseudo-code with real arithmetic, as provided in Feron's article, to similar invariants on machine code that take into account rounding errors introduced by floating-point arithmetic. We use them to verify whether the stability conditions still hold, in which case system stability with floating-point numbers is established. This approach is based on a generic theoretical framework, the soundness of which is proved in Coq, described further in this document, and is implemented as an automatic tool.

This part of the dissertation is organized as follows. In the remainder of this chapter, we describe the second-order dynamical system example used by Feron in [Fer10], with the corresponding controller. Next, Feron's analysis of the open-loop controller with real numbers is presented. In the next chapters, we introduce our generic translation scheme (Chapter 3, page 29) and its implementation as a Python library, and use it thereafter to rewrite Feron's analysis with

floating-point arithmetic (Chapter 4, page 37). Finally, the case of the closed-loop system is handled. This part concludes with a discussion on the generality of this approach.

2.1 Lyapunov Stability Theory

Various types of stability may be discussed for the solutions of differential equations describing dynamical systems. The most important type is the one concerning the stability of solutions near to a point of equilibrium. This may be discussed by the theory of Lyapunov. In simple terms, if the solutions that start out near an equilibrium point x_e stay near x_e forever, then x_e is *Lyapunov stable*. More strongly, if x_e is Lyapunov stable and all solutions that start out near x_e converge to x_e, then x_e is *asymptotically stable*.

Lyapunov functions are scalar functions that may be used to prove the stability of an equilibrium. The existence of Lyapunov functions is a necessary and sufficient condition to establish the stability of a discrete, time-invariant closed system, meaning that its state variable will remain bounded during the execution [HC11].

Informally, a Lyapunov function is a function that takes positive values everywhere except at the equilibrium in question, and decreases (or is non-increasing) along every trajectory of the control system's differential equation f. It is a function $V : \mathbb{R}^n \to \mathbb{R}$ that satisfies the following properties:

$$V(0) = 0 \quad \text{and} \quad \forall x \in \mathbb{R}^n \setminus \{0\}, V(x) > 0 \quad \text{and} \quad \lim_{\|x\| \to \infty} V(x) = \infty \tag{2.1}$$

$$\forall x \in \mathbb{R}^n, V(f(x)) - V(x) \leq 0. \tag{2.2}$$

In the case of discretized linear systems, i.e. systems whose internal state is defined according to Equation (1.3):

$$x_{k+1} = Ax_k + Bu_k$$

and assuming that

$$\|u_k\|_\infty \leq 1, \tag{2.3}$$

it can be shown that the Lyapunov function V can be taken as a quadratic form of the state variable of the system:

$$V(x) = x^\intercal P x$$

with $P \in \mathbb{R}^{n \times n}$ a symmetric matrix such that

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

$$A^\intercal P A - P \prec 0 \tag{2.5}$$

where the notation "$P \succ 0$" means that P is a positive-definite matrix, i.e., for all non-zero vector x, ensuring that $x^\intercal P x > 0$. Equation (2.3) expresses that values coming from input sensors are usually bound into a given range. The bound value 1 is chosen with no loss of generality since the matrix B can always be altered to account for different bounds.

2.2 Motivating Example by Feron

We consider the first system described in the article of Feron [Fer10]. It is a dynamical system composed of a single mass and a single spring shown in Figure 2.1.

The position input y of the mass is available for feedback control. The signal y_d is the reference signal, that is, the desired position to be followed by the mass. We do not show (and Feron does not) that the distance $|y_d - y|$ is bounded. y_d has to be understood as a command.

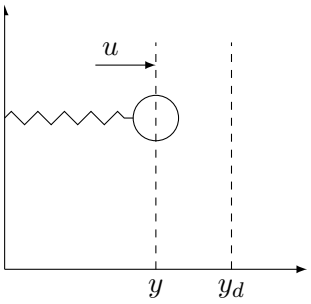


Figure 2.1: Mass-spring system

A discrete-time MATLAB implementation of the controller, using real numbers, is provided in [Fer10]. The source code is shown in Listing 2.1. The process to produce this code from the system modeling is covered in detail in Feron's paper.

```
Ac = [0.4990, -0.0500; 0.0100, 1.0000];
Bc = [1; 0];
Cc = [564.48, 0];
Dc = -1280;
xc = zeros(2, 1);
receive(y, 2); receive(yd, 3);
while (1)
    yc = max(min(y - yd, 1), -1);
    skip;
    u = Cc*xc + Dc*yc;
    xc = Ac*xc + Bc*yc;
    send(u, 1);
    receive(y, 2);
    receive(yd, 3);
    skip;
end
```

Listing 2.1: Pseudocode of the controller

In this code, the `skip` statement is a null operation: when it is executed, nothing happens. It is useful as a placeholder when a statement is required syntactically to be surrounded by invariants, but no code needs to be executed.

Apart from the mechanical system state observation y and the desired system output y_d, variables in this code are:

- $x_c = \left(\begin{smallmatrix} xc_1 \\ xc_2 \end{smallmatrix}\right) \in \mathbb{R}^2$ is the discrete-time controller state;
- $y_c \in [-1, 1]$ is the bounded output tracking error, i.e. the input $(y - y_d)$ is passed through a saturation function to avoid variable overflow in the controller;
- $u \in \mathbb{R}$ is the mechanical system input (and the controller output at the same time), i.e. the action to be performed according to the controller.

Constants A_c, B_c, C_c and D_c are the discrete-time controller state, input, output and feedthrough matrices. The commands `send` and `receive` are basically I/O: they respectively send and receive data given in the commands' first argument through a specific channel given by the commands' second argument.

2.3 Open-Loop Stability Proof with Reals

The stability proof of this system relies on Lyapunov theory. As seen in Section 1.3 and Section 2.1, a system is Lyapunov stable if all states x_c reachable from an initial starting state belonging to a bounded neighborhood V of an equilibrium point x_e remain in V. Lyapunov theory provides constraints that must be satisfied by such a V. On linear systems, they are equations that can be solved using linear matrix inequalities [BEGFB94]. Commonly, V is an ellipsoid.

In our case, to prove Lyapunov stability, we need to show that at any time, x_c belongs to the set $\mathcal{E}_P$ chosen by Feron:

$$\mathcal{E}_P = \{x \in \mathbb{R}^2 \mid x^\intercal \cdot P \cdot x \leq 1\}, P = 10^{-3} \begin{pmatrix} 0.6742 & 0.0428 \\ 0.0428 & 2.4651 \end{pmatrix}$$

using $\mathcal{E}_P$ as the stability neighborhood V.

This set is the full ellipse shown in Figure 2.2, centered around 0 and slightly slanted:

$$x_c \in \mathcal{E}_P \iff 0.6742x_{c1}^2 + 0.0856x_{c1}x_{c2} + 2.4651x_{c2}^2 \leq 1000.$$

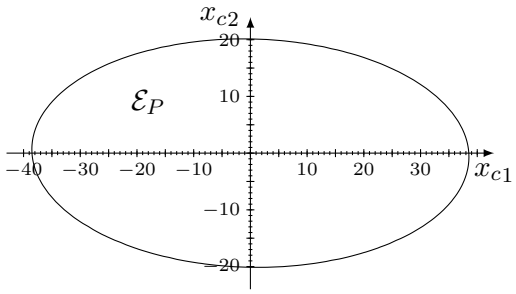


Figure 2.2: The stability domain $\mathcal{E}_P$

A stability proof of the controller code is provided in [Fer10], using Floyd-Hoare program annotation technique [Hoa69, Flo93, Pel01]: each program instruction comes with an invariant. The program annotated by Feron is reproduced below.

```
5  xc = zeros(2,1);
   % x_c ∈ E_P
6  receive(y, 2); receive(yd, 3);
   % x_c ∈ E_P
7  while (1)
       % x_c ∈ E_P
8      yc = max(min(y - yd, 1), -1);
       % x_c ∈ E_P, y_c^2 ≤ 1
9      skip;
       % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10     u = Cc*xc + Dc*yc;
       % (x_c; y_c) ∈ E_{Q_μ}, u^2 ≤ (C_c D_c) · Q_μ^{-1} · (C_c D_c)^{-1}
11     xc = Ac*xc + Bc*yc;
       % x_c ∈ E_R, R = [(A_c B_c) · Q_μ^{-1} · (A_c B_c)^T]^{-1},
       % u^2 ≤ (C_c D_c) · Q_μ^{-1} · (C_c D_c)^{-1}
12     send(u, 1);
       % x_c ∈ E_R
```

```
13    receive(y, 2);
      % x_c ∈ E_R
14    receive(yd, 3);
      % x_c ∈ E_R
15    skip;
      % x_c ∈ E_P
16  end
```

Listing 2.2: Pseudocode with proof annotations

Most of the steps of the proof rely on algebraic arguments. For example, the invariant loosening on Line 9:

```
      % x_c ∈ E_P,   y_c^2 ≤ 1
9     skip;
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
```

means

$$x_c \in \mathcal{E}_P \wedge y_c^2 \leq 1 \Longrightarrow \left(\begin{smallmatrix} x_c \\ y_c \end{smallmatrix}\right) \in \mathcal{E}_{Q_\mu},$$

with $Q_\mu = \left(\begin{smallmatrix} \mu P & 0_{2\times 1} \\ 0_{1\times 2} & 1-\mu \end{smallmatrix}\right)$ and $\mu = 0.9991$.

The correctness of this assertion stems from the fact that, given any value of $\mu \in [0, 1]$, the domain $\mathcal{E}_{Q_\mu}$ is a solid ellipsoid, centered around 0, and whose intersection with the planes $y_c = 1$ and $y_c = -1$ is equal to $\mathcal{E}_P$. Consequently, the solid bounded cylinder $\mathcal{C} = \{\left(\begin{smallmatrix} x_c \\ y_c \end{smallmatrix}\right) \mid x_c \in \mathcal{E}_P \wedge y_c^2 \leq 1\}$ is included within $\mathcal{E}_{Q_\mu}$ (Figure 2.3).

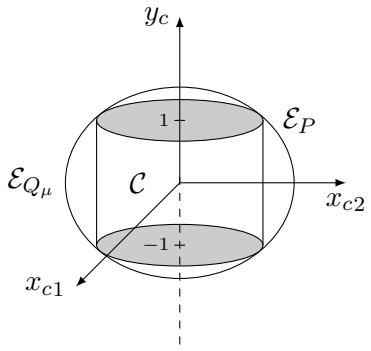


Figure 2.3: Inclusion of $\mathcal{E}_P$ within $\mathcal{E}_{Q_\mu}$

The next two invariants

```
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10    u = Cc*xc + Dc*yc;
      % (x_c; y_c) ∈ E_{Q_μ}, u^2 ≤ (C_c  D_c) · Q_μ^{-1} · (C_c  D_c)^{-1}
11    xc = Ac*xc + Bc*yc;
      % x_c ∈ E_R, R = [(A_c  B_c) · Q_μ^{-1} · (A_c  B_c)^T]^{-1},
      % u^2 ≤ (C_c  D_c) · Q_μ^{-1} · (C_c  D_c)^{-1}
```

also rely on similar algebraic arguments and theorems. Other invariants are trivial. Finally, only the very last loosening

```
      % x_c ∈ E_R
      skip;
      % x_c ∈ E_P
```

i.e. $\mathcal{E}_R \subset \mathcal{E}_P$, that "closes" the loop, is not purely algebraic since its validity relies on the numerical parameters A_c, B_c, C_c, D_c. This assertion needs to be checked to ensure the correctness of the proof statements. This can be done either numerically, or algebraically for at most two-dimensional systems like this one.

We have checked this proof using Mathematica 8 [Wol14]. Our proof notebook is available online[†1].

[†1] Mathematica source file is available at: `http://www.cri.mines-paristech.fr/people/maisonneuve/lyafloat/resources/lyafloat_stability.nb`, and the corresponding PDF file at: `http://www.cri.mines-paristech.fr/people/maisonneuve/lyafloat/resources/lyafloat_stability.pdf`.

Chapter 3

Proof Scheme with Limited-Precision Numbers

Nous voulons vérifier que la preuve de stabilité est toujours valide sur un dispositif de commande réaliste, en utilisant des nombres à précision limitée (par exemple, mais pas nécessairement, des nombres à virgule flottante). Avec une arithmétique en précision limitée, les valeurs des constantes sont légèrement modifiées et les calculs sont susceptibles de produire des erreurs d'arrondi.

D'un point de vue théorique, il est impossible de passer de nombres réels à des nombres de précision limitée sans affecter le comportement du contrôleur. Ainsi, la preuve de la stabilité ne peut pas être conservée dans le cas général. Par ailleurs, le programme peut encore être stable si les erreurs d'arrondi sont suffisamment petites et que l'inclusion finale $\mathcal{E}_R \subset \mathcal{E}_P$ leur laisse assez de place. Nous étudions comment les invariants de preuve peuvent être modifiés pour être appliqués à une sémantique en précision limitée. L'objectif est d'obtenir, à partir du schéma de preuve en nombres réels, un schéma de preuve qui convienne à une arithmétique en précision limitée et dont la correction, non garantie, pourra être vérifiée aussi aisément que dans la preuve d'origine.

* * *

We would like to check that the stability proof still holds on a realistic controller device, using limited-precision numbers (for instance, but not necessarily, floating-point numbers). When using limited-precision arithmetic, the values of constants are slightly altered and calculations are likely to produce rounding errors.

In absolute terms, it is impossible to switch from real to limited-precision numbers without affecting the behavior of the controller. Thus, the stability proof cannot be preserved in the general case. On the other hand, the program can still be stable if rounding errors are small enough and the final inclusion $\mathcal{E}_R \subset \mathcal{E}_P$ leaves enough room. We study how proof invariants can be tweaked so that they apply to a limited-precision semantic. Our goal is to derive from the proof scheme for real numbers a proof scheme suited for limited-precision arithmetic, whose correctness, although not guaranteed, can be checked as easily as in the original proof.

3.1 Formalism

In this section, an abstract formalism is defined to handle a program and its proof, with both real and limited-precision arithmetic. This formalism is implemented in Coq.

Let X be the (finite) set of variables in the program.

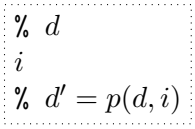


Figure 3.1: Abstract scheme of invariant propagation on the original program

3.1.1 Values

Let $\mathbb{F} \subset \mathbb{R}$ be the target set of limited-precision representations of real numbers (e.g., floating-point numbers). We assume that each real number $c \in \mathbb{R}$ has a finite-precision representation $\bar{c} \in \mathbb{F}$ and that Equation (3.1) is satisfied:

$$c \in \mathbb{F} \Longrightarrow \bar{c} = c. \tag{3.1}$$

Throughout the rest of the document, we will use the uniform notation $\mathbb{K}$ to refer to either the set $\mathbb{R}$ or $\mathbb{F}$.

3.1.2 Functions Symbols

We consider a set of function symbols $\mathcal{F}_{\mathbb{R}}$ on real numbers. Each real function symbol $f \in \mathcal{F}_{\mathbb{R}}$ is associated to a finite-precision counterpart $\bar{f} \in \mathcal{F}_{\mathbb{F}}$. Functions symbols in $\mathcal{F}_{\mathbb{R}}$ and $\mathcal{F}_{\mathbb{F}}$ will be simply referred as *functions* when there will be no ambiguity.

Each function symbol f in $\mathcal{F}_{\mathbb{R}}$ or $\mathcal{F}_{\mathbb{F}}$ has an *arity* $n \in \mathbb{N}$ and can be *evaluated* into a corresponding mathematical function:

$$f \in \mathcal{F}_{\mathbb{K}} \Longrightarrow \mathrm{eval}(f) \in \mathbb{K}^n \to \mathbb{K}.$$

Functions f and $\bar{f}$ have the same arity:

$$\mathrm{arity}\, f = \mathrm{arity}\, \bar{f}.$$

As for now, no other assumption is made as to the behavior of $\bar{f}$ compared to f. In particular, there is no guarantee that, applied to the same inputs, they will return the same value.

Notice that function symbols are not identified to their evaluations. The reason is that functions with similar evaluations on real arithmetic may behave differently on a limited-precision paradigm, e.g. function $f : x \mapsto 2^{100} + x - 2^{100}$ compared to the identity function $\mathrm{Id} : x \mapsto x$: $\mathrm{eval}(f) = \mathrm{eval}(\mathrm{Id})$ but $\mathrm{eval}(\bar{f}) \neq \mathrm{eval}(\overline{\mathrm{Id}})$.

3.1.3 Valuations

A *valuation* v on $\mathbb{K}$ is a function that maps variables in X to values in $\mathbb{K}$. The set of valuations on $\mathbb{K}$ is noted $\mathrm{Val}_{\mathbb{K}}$.

$$\mathrm{Val}_{\mathbb{K}} \ni v : X \to \mathbb{K}.$$

Notice that $\mathrm{Val}_{\mathbb{F}}$ is a subset of $\mathrm{Val}_{\mathbb{R}}$.

Each valuation v in $\mathrm{Val}_{\mathbb{R}}$ is associated to a valuation $\bar{v}$ in $\mathrm{Val}_{\mathbb{F}}$, defined as follows:

$$\bar{v} : x \in X \longmapsto \overline{v(x)}.$$

Using this definition, it can easily be shown using Equation (3.1) that for any valuation $v \in \mathrm{Val}_{\mathbb{R}}$,

$$v \in \mathrm{Val}_{\mathbb{F}} \Longrightarrow \bar{v} = v.$$

Let v_1 and v_2 be two valuations in $\mathrm{Val}_{\mathbb{R}}$. The *distance* dist between v_1 and v_2 is the valuation that maps a variable in X to the distance in absolute value between its associated values in v_1 and v_2:

$$\mathrm{Val}_{\mathbb{R}} \ni \mathrm{dist}(v_1, v_2) = x \in X \longmapsto |v_1(x) - v_2(x)|.$$

We say that v_1 is *lower* than v_2, and note $v_1 \leq v_2$, if

$$\forall x \in X, v_1(x) \leq v_2(x).$$

Finally, the *sum* of the two valuations, noted $v_1 + v_2$, is the valuation defined as follows:

$$v_1 + v_2 : x \in X \longmapsto v_1(x) + v_2(x).$$

3.1.4 Domains (Invariants)

Domains are used to represent Floyd's and Hoare's invariants seen in previous sections. A *domain* d on $\mathbb{K}$ is a set of valuations on $\mathbb{K}$, i.e. $d \subset \mathrm{Val}_{\mathbb{K}}$. The set of domains on $\mathbb{K}$ is noted $\mathcal{D}_{\mathbb{K}}$, with $\mathcal{D}_{\mathbb{F}} \subset \mathcal{D}_{\mathbb{R}}$.

A real domain $d \in \mathcal{D}_{\mathbb{R}}$ can be associated to a limited-precision domain $\bar{d} \in \mathcal{D}_{\mathbb{F}}$ defined by:

$$\bar{d} = \{\bar{v} \,|\, v \in d\}.$$

As for valuations in $\mathbb{F}$, it can be shown that

$$d \in \mathcal{D}_{\mathbb{F}} \Longrightarrow d = \bar{d}.$$

We also define an operation to overapproximate a domain with respect to a valuation (intuitively, an "error valuation"). The *extension* of a domain $d \in \mathcal{D}_{\mathbb{R}}$ using a positive valuation $v \in \mathrm{Val}_{\mathbb{R}}$, noted $d \oplus v$, is defined as follows:

$$d \oplus v = \{v_2 \in \mathrm{Val}_{\mathbb{R}} \,|\, v_1 \in d \wedge \mathrm{dist}(v_1, v_2) \leq v\}.$$

3.1.5 Expressions

We consider a very simple language with variables, constants and function calls. This language is expressive enough to represent most of control-command softwares, which corresponds to our needs. The *expressions* $\mathcal{E}_{\mathbb{K}}$ of this language are constructed as follows:

$$\begin{array}{rll} \mathcal{E}_{\mathbb{K}} \ni e ::= & x \in X & \text{variable} \\ | & c \in \mathbb{K} & \text{constant} \\ | & f(e_1, \ldots, e_n) & \text{function call} \\ & \quad \text{with } f \in \mathcal{F}_{\mathbb{K}}, \text{arity } f = n & \\ & \quad \text{and } \forall i \in [1, n], e_i \in \mathcal{E}_{\mathbb{K}} & \end{array}$$

An expression $e \in \mathbb{K}$ can be evaluated in the environment described by the valuation $v \in \mathrm{Val}_{\mathbb{K}}$. The result is a value in $\mathbb{K}$.

$$\mathrm{eval}(e, v) = \begin{cases} v(x) & \text{if } e = x \in X \\ c & \text{if } e = c \in \mathbb{K} \\ \mathrm{eval}(f)(c_1, \ldots, c_n) & \text{if } e = f(e_1, \ldots, e_n) \\ \quad \text{with } \forall i \in [1, n], c_i = \mathrm{eval}(e_i, v) & \end{cases}$$

where the notation eval is overloaded to handle expressions.

3.1.6 Instructions

An *instruction* is the affectation of an expression value to a variable. The set of instructions is noted $\mathcal{I}_{\mathbb{K}}$:

$$\mathcal{I}_{\mathbb{K}} = X \times \mathcal{E}_{\mathbb{K}}$$

We note $x := e$ the instruction $(x, e) \in \mathcal{I}_{\mathbb{K}}$.

As for expressions, an instruction $(x := e) \in \mathcal{I}_{\mathbb{K}}$ can be evaluated in an environment $v \in \mathrm{Val}_{\mathbb{K}}$. The result is a valuation in $\mathrm{Val}_{\mathbb{K}}$.

$$\mathrm{eval}(x := e, v) = \left(y \in X \longmapsto \begin{cases} \mathrm{eval}(e, v) & \text{if } x = y \\ v(y) & \text{if } x \neq y \end{cases} \right)$$

This definition can be extended to evaluate the image of a domain $d \in \mathcal{D}_{\mathbb{K}}$, the result being also a domain of $\mathcal{D}_{\mathbb{K}}$.

$$\mathrm{eval}(x := e, d) = \{\mathrm{eval}(x := e, v) \,|\, v \in d\}$$

3.1.7 Invariant Propagation

An invariant propagator, or simply *propagator*, is a function p that takes as input a domain in $\mathcal{D}_{\mathbb{K}}$, known as *precondition*, an instruction in $\mathcal{I}_{\mathbb{K}}$, and returns a domain in $\mathcal{D}_{\mathbb{K}}$ called *postcondition*.

$$p : \mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}} \longrightarrow \mathcal{D}_{\mathbb{K}}$$

A propagator is expected to return a valid postcondition for a given precondition and instruction. Formally, this can be described by the correctness condition

$$\forall (d, i) \in \mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}}, \forall v \in d, \mathrm{eval}(i, v) \subset p(d, i)$$

that can also be written:

$$\forall (d, i) \in \mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}}, \mathrm{eval}(i, d) \subset p(d, i). \tag{3.2}$$

The set of (correct) propagators is noted $\mathcal{P}_{\mathbb{K}}$.

Propagators are an abstract representation of the argument theorems used to propagate Floyd's and Hoare's invariants en route to prove a program. Unlike these arguments, propagators have to be defined on the whole domain $\mathcal{D}_{\mathbb{K}} \times \mathcal{I}_{\mathbb{K}}$, that is, for any precondition and instruction. It is well known that most mathematical theorems make assumptions on their inputs. To make propagators total functions, the universal postcondition can always be returned if the domain is outside the scope of the corresponding argument, for instance when the hypothesis of some Lyapunov theorem is not verified.

3.2 Translation Scheme

The general translation scheme that is implemented is the following (see Figure 3.2). Each instruction i in the abstract controller code with real numbers is turned into a corresponding instruction $\bar{\imath}$ on the concrete controller, where real constants and operators are replaced by their limited-precision counterparts. A post-condition $\bar{d}'$ on the limited-precision side is computed using the same proof argument p as in the original code, "enlarged" to include the error term bound v_{err} that may occur in limited-precision arithmetics in $\bar{\imath}$. The same propagator p is used because Floyd's and Hoare's invariants are still based on real numbers, even if the code is implemented with a limited-precision arithmetic. v_{err} is computed on an "intermediate" instruction form $\tilde{\imath}$, the meaning of which will be explained later. The precondition d might have been previously replaced by $\bar{d}$ as the postcondition of a preceding instruction.

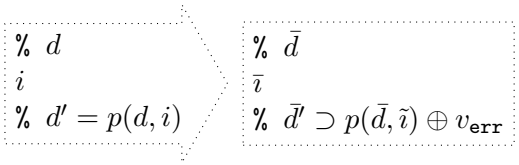


Figure 3.2: General translation scheme

The error term bound v_{err} depends on the operators of the instruction $\bar{\imath}$ and on the information on the program variables given by $\bar{d}$. The limited-precision arithmetic error is not necessarily bounded even when the variables are, e.g. consider division. In this case, we are not able to compute a suitable $\bar{d}'$ and the proof translation attempt fails.

Also, $\bar{d}'$ will be used as a precondition in the next instruction. An effort has to be made so that $\bar{d}'$ has a similar shape to d', in order to fit with the proof argument p' used in this instruction, if possible. In our case, we will strive to maintain the invariant under the form of an ellipsoid. Thus, $\bar{d}'$ is an overapproximation of $p(\bar{d}, \bar{\imath}) \oplus v_{\text{err}}$ in the domain of p'. In consequence, the automated analysis falls short if such a $\bar{d}'$ cannot be found.

3.3 Translation Steps

In this section, we consider an instruction $i = (x := e)$ in $\mathcal{I}_{\mathbb{R}}$, along with a precondition d and a postcondition d' in $\mathcal{D}_{\mathbb{R}}$. d, i and d' are linked through a propagator p in $\mathcal{P}_{\mathbb{R}}$:

$$d' = p(d, i).$$

As outlined in Section 3.2, we study how the instruction i can be translated into an equivalent instruction $\bar{\imath} \in \mathcal{I}_{\mathbb{F}}$ on limited-precision arithmetic, while using information on the precondition d and propagator p of i to compute an interesting postcondition $\bar{d}'$ for $\bar{\imath}$ (given a modification $\bar{d}$ of d). This is a two-step process:

1. First, the program constants $c \in \mathbb{R}$ are converted into their limited-precision representations $\bar{c} \in \mathbb{F}$ while keeping absolute-precision functions, giving an instruction $\tilde{\imath} \in \mathcal{I}_{\mathbb{R}}$. This induces a modified application of the propagator p.

2. Second, the functions $f \in \mathcal{F}_{\mathbb{R}}$ that appear in the code are changed into their limited-precision counterparts $\bar{f} \in \mathcal{F}_{\mathbb{F}}$. This induces the introduction of rounding errors that are added to the invariant obtained at step 1.

The resulting code relies on limited-precision arithmetic only, which is what is being sought. The translation scheme is described in Figure 3.3.

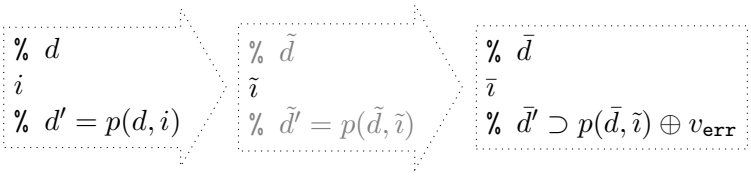


Figure 3.3: Refined translation scheme

3.3.1 Converting Constants

We construct an expression $\tilde{e} \in \mathcal{E}_{\mathbb{R}}$, obtained from e by converting its constants from $\mathbb{R}$ to $\mathbb{F}$. Formally, $\tilde{e}$ is defined recursively as follows:

$$\tilde{e} = \begin{cases} x & \text{if } e = x \in X \\ \bar{c} & \text{if } e = c \in \mathbb{R} \\ f(\tilde{e}_1, \ldots, \tilde{e}_n) & \text{if } e = f(e_1, \ldots, e_n) \end{cases} \tag{3.3}$$

We let $\tilde{\imath} = (x := \tilde{e})$.

Let $\bar{d} \in \mathcal{D}_{\mathbb{F}}$ be the precondition corresponding to d after converting both constants and functions in the preceding instructions of the program. Let $\tilde{d}' = p(\bar{d}, \tilde{\imath}) \in \mathcal{D}_{\mathbb{R}}$. According to Equation (3.2), $\tilde{d}'$ is a valid postcondition for precondition $\bar{d}$ and instruction $\tilde{\imath}$.

3.3.2 Converting Functions

The next step is to convert real functions f that appear in the expression into their limited-precision counterparts $\bar{f}$, as defined in Section 3.1.2. The resulting expression $\bar{e} \in \mathcal{E}_{\mathbb{F}}$ is obtained with:

$$\bar{e} = \begin{cases} x & \text{if } e = x \in X \\ \bar{c} & \text{if } e = c \in \mathbb{R} \\ \bar{f}(\bar{e}_1, \ldots, \bar{e}_n) & \text{if } e = f(e_1, \ldots, e_n) \end{cases} \tag{3.4}$$

Compared to the definition of $\tilde{e}$ in Equation (3.3), we just have turned functions symbols f into $\bar{f}$. As previously, we also let $\bar{\imath} = (x := \bar{e})$.

Unlike $\tilde{\imath}$, instruction $\bar{\imath}$ relies on different functions than i (taken from $\mathcal{F}_{\mathbb{F}}$ instead of $\mathcal{F}_{\mathbb{R}}$): the underlying invariant propagation argument in propagator p is very likely not to be applicable to it, as the mathematical theorems used in p typically rely on properties of real arithmetic operators. In this case, $p(\bar{d}, \bar{\imath})$ would fall back on the universal domain, which of course is valid but prevents us from doing any interesting proof further in the program. We want to obtain a more precise and still valid postcondition invariant $\bar{d}'$.

Functional Arithmetic Errors

To circumvent this issue, we propose when it is possible to "enlarge" the invariant $\tilde{d}' = p(\bar{d}, \tilde{\imath})$ found in Section 3.3.1 to take into account arithmetic errors introduced when converting functions from $\mathcal{F}_{\mathbb{R}}$ to $\mathcal{F}_{\mathbb{F}}$. Such errors are called functional arithmetic errors, but we will refer to them as *arithmetic errors*, or simply errors, later in this document.

Equation (3.5) defines the arithmetic error on an expression $\tilde{e} \in \mathcal{E}_{\mathbb{R}}$ modulo a valuation $v \in \mathrm{Val}_{\mathbb{F}}$. The result is a value in $\mathbb{R}$.

$$\mathrm{err}(\tilde{e}, v) = |\,\mathrm{eval}(\bar{e}, v) - \mathrm{eval}(\tilde{e}, v)| \tag{3.5}$$

This definition is extended to define the arithmetic error induced by an instruction $\tilde{\imath} = (x := \tilde{e})$ in $\mathcal{I}_{\mathbb{R}}$ on a valuation $v \in \mathrm{Val}_{\mathbb{F}}$. The result is a valuation in $\mathrm{Val}_{\mathbb{R}}$ defined as follows:

$$\mathrm{err}(x := \tilde{e}, v) = \left(y \in X \longmapsto \begin{cases} \mathrm{err}(\tilde{e}, v) & \text{if } x = y \\ 0 & \text{if } x \neq y \end{cases} \right).$$

Finally, a valuation $v_{\mathrm{err}} \in \mathrm{Val}_{\mathbb{F}}$ is an *upper bound* of the arithmetic error induced by instruction $\tilde{\imath}$ on a domain $d \in \mathcal{D}_{\mathbb{R}}$ if:

$$\forall v \in d, \mathrm{err}(\tilde{\imath}, v) \leq v_{\mathrm{err}}.$$

Now, let us consider an instruction $\tilde{\imath}$ in $\mathcal{I}_{\mathbb{R}}$ surrounded by a precondition $\bar{d}$ and a postcondition $\tilde{d'} = p(\bar{d}, \tilde{\imath})$ in $\mathcal{D}_{\mathbb{R}}$. We have:

$$\text{eval}(\tilde{\imath}, \bar{d}) \subset \tilde{d'}$$

Theorem 3.1. Let $v_{\text{err}} \in \text{Val}_{\mathbb{F}}$ be an upper bound of the arithmetic error induced by instruction $\tilde{\imath}$ on domain $\bar{d}$. Then, the implication

$$\forall \bar{d'} \in \mathcal{D}_{\mathbb{R}}, \tilde{d'} \oplus v_{\text{err}} \subset \bar{d'} \Longrightarrow \text{eval}(\bar{\imath}, \bar{d}) \subset \bar{d'}$$

holds.

In other words, if $\tilde{d'}$ is a postcondition for an instruction $\tilde{\imath}$ on real numbers, on a given precondition domain $\bar{d}$, and $\bar{d'}$ is a superset of $\tilde{d'}$ loose enough to support the arithmetic error v_{err}, then $\bar{d'}$ is a valid postcondition for $\bar{\imath}$, the limited-precision counterpart of instruction $\tilde{\imath}$.

A proof in Coq, limited to binary functions (which include all the functions that we need in this document), has been implemented and is available at `http://www.cri.mines-paristech.fr/people/maisonneuve/thesis/LyaFloat.v`.

Chapter 4

Translating Stability Proof Scheme to Floating-Point Numbers

Nous voulons vérifier si la preuve de stabilité dans le programme de Feron est toujours valable sur un contrôleur implémenté avec des nombres flottants, en utilisant l'approche présentée dans le chapitre précédent. Dans ce chapitre, on utilisera le standard IEEE pour l'arithmétique à virgule flottante [IEE08], codée sur 64 bits[†1]*, comme c'est le cas dans la majorité des unités de calcul à virgule flottante actuelles. Dans cette norme, l'addition et la multiplication sont correctement arrondies en fonction du mode actif d'arrondi, ce qui permet de borner l'erreur en fonction de la valeur des opérandes. Le cas des représentations numériques alternatives est discuté dans la section 4.5.*

* * *

We would like to check whether the stability proof in Feron's program still holds on a controller implemented with floating-point numbers using the approach of the previous chapter. In this chapter, we use the IEEE Standard for Floating-Point Arithmetic [IEE08] encoded on 64 bits[†2], as most of today's floating-point units do. In this standard, both addition and multiplication are correctly rounded depending on the active rounding mode, which allows to bound the rounding error depending on the values of operands. The case of alternative numeric representations is discussed in Section 4.5.

4.1 Automatic Translation

We have developed a program to automatically perform these computations, following the translation scheme framework described in the previous chapter. It is a module called LyaFloat, written in Python and built upon Python libraries SymPy (to handle symbolic mathematics) and Mpmath (for arbitrary-precision floating-point arithmetic).

Here is the listing of a script that automatically computes the floating-point output invariant ellipsoid $\overline{\mathcal{E}_R}$ (in variable `ERbar`) corresponding to $\mathcal{E}_R$ in the original proof, and tests whether $\overline{\mathcal{E}_R} \subset \mathcal{E}_P$. Two cases are possible:

- either $\overline{\mathcal{E}_R} \subset \mathcal{E}_P$, then the program is Lyapunov-stable on a floating-point architecture;
- or $\overline{\mathcal{E}_R} \not\subset \mathcal{E}_P$: as $\overline{\mathcal{E}_R}$ was obtained through overapproximations, we cannot conclude about the program behavior.

[†1] La procédure qui suit serait exactement la même avec des nombres flottants 32 bits, les résultats numériques seraient simplement différents. La précision numérique est en fait un paramètre du système.

[†2] The procedure that follows would be exactly the same with 32-bit floating-point numbers, only with different numerical results. In fact, arithmetic precision is a parameter of the system.

```
from lyafloat import *
# Parameters
setfloatify(constants=True, operators=True,
       precision=64)

# Definition of E_P
P = Rational("1e-3") * Matrix(rationals(
       ["0.6742 0.0428", "0.0428 2.4651"]))
EP = Ellipsoid(P)

# Definition of E_{Q_mu}
mu = Rational("0.9991")
Qmu = mu * P
Qmu = Qmu.col_insert(2, zeros(2, 1)).
       row_insert(2, zeros(1, 3))
Qmu[2,2] = 1 - mu
EQmu = Ellipsoid(Qmu)

# Symbols
xc1, xc2, yc = symbols("xc1 xc2 yc")
Xc = Matrix([[xc1], [xc2]])
Yc = Matrix([[yc]])
Zc = Matrix([[xc1], [xc2], [yc]])

# Constant matrices
Ac = Matrix(constants(
       ["0.4990 -0.0500", "0.0100 1.0000"]))
Bc = Matrix(constants(["1", "0"]))
Cc = Matrix(constants(["564.48 0"]))
Dc = Matrix(constants(["-1280"]))

# Definition and verification of E_R
AcBc = Ac.col_insert(Ac.cols, Bc)
R = (AcBc * Qmu.inv() * AcBc.T).inv()
ER = Ellipsoid(R)
print("ER included in EP :", ER <= EP)

# Computation and verification of overline(E_R)
i = Instruction({Xc: Ac * Xc + Bc * Yc},
       pre=[Zc in EQmu], post=[Xc in ER])
ERbar = i.post()[Xc]
print("ERbar =", ERbar)
print("ERbar included in EP :", ER <= EP)
```

Listing 4.1: Computation of $\overline{\mathcal{E}_R}$ using LyaFloat

In this open-loop case, our program LyaFloat is able to check the inclusion. Thus, the stability of the open-loop system with a 64-bit IEEE 754 compliant implementation is formally proven to hold using our proof translation scheme.

Notice that the precision of the floating-point arithmetic can be set in `setfloatify`. This allows us to check that the controller is still stable on a 32-bit only architecture, or to compute that the minimum required precision is 17 bits. This can be useful if, instead of adapting the

controller code and/or its proof to the hardware, we have to select a controller device that suits the proof.

In the remainder of this section, we describe the operations performed by our tool to compute $\overline{\mathcal{E}_R}$. It follows the two-step scheme described in Section 3.3. Alternatively to our program, an implementation of this proof using Mathematica is available online: see Footnote †1, page 27.

4.2 Handling Constants

Following Section 3.3.1, the first step when translating Feron's program is to convert real constants into floating-point numbers. The first lines of code in the program:

```
1 Ac = [0.4990, -0.0500; 0.0100, 1.0000];
2 Bc = [1; 0];
3 Cc = [564.48, 0];
4 Dc = -1280;
```

Listing 4.2: Original constant definitions in the program

become, assuming rounding to nearest value:

```
1 Ac = [0.4989999999999999991118215802998747676610946655273437 5,
       -0.0500000000000000027755575615628913510590791702270507812 5;
        0.0100000000000000000208166817117216851329430937767028808593 75,
        1.0000];
2 Bc = [1; 0];
3 Cc = [564.4800000000000181898940354585647583007812 5, 0];
4 Dc = -1280
```

Listing 4.3: Constant definitions, using floating-point values

Theses matrices $\overline{A_c}$, $\overline{B_c}$, $\overline{C_c}$ and $\overline{D_c}$ will be used instead of the original matrices A_c, B_c, C_c and D_c in the sequel of the proof.

Apart from constants, the proof scheme for the first part of the program is unchanged:

```
5 xc = zeros(2,1);
  % x_c ∈ E_P
6 receive(y, 2); receive(yd, 3);
  % x_c ∈ E_P
7 while (1)
      % x_c ∈ E_P
8     yc = max(min(y - yd, 1), -1);
      % x_c ∈ E_P, y_c^2 ≤ 1
9     skip;
      % (x_c; y_c) ∈ E_{Q_μ}, Q_μ = [μP 0_{2×1}; 0_{1×2} 1−μ], μ = 0.9991
10    u = Cc*xc + Dc*yc;
```

Listing 4.4: Beginning of the proof, unchanged

4.3 Handling Functions

The second step of our proof translation scheme is to convert function symbols from real to floating-point. We show how it is done instruction by instruction.

4.3.1 Invariant on u

The next instruction after Section 4.2 in the original proof scheme is:

```
% (xc yc) ∈ E_{Q_μ}, Q_μ = ( μP  0_{2×1} ; 0_{1×2}  1−μ ), μ = 0.9991
10  u = Cc*xc + Dc*yc;
% (xc yc) ∈ E_{Q_μ}, u² ≤ (C_c  D_c) · Q_μ^{-1} · (C_c  D_c)^{-1}
```

Listing 4.5: Computation of u, original code

First of all, matrices C_c and D_c must be replaced by their floating-point counterparts $\overline{C_c}$ and $\overline{D_c}$ both in the program instruction Line 10 and the ensuing invariant. This invariant relies only on algebraic arguments and does not depend on the values in the matrices, it still holds considering exact arithmetic operations. But this is not sufficient: indeed, this instruction is a sum of matrix multiplications, i.e. a set of additions and multiplications on floating-point numbers that yield rounding errors (see Section 3.3.2).

We can notice that entering this instruction, the values of matrices $\overline{C_c}$, $\overline{D_c}$ and $\mathcal{E}_{Q_\mu}$ are known, and the values of x_c and y_c are bounded by the precondition

$$\left(\begin{smallmatrix} x_c \\ y_c \end{smallmatrix}\right) \in \mathcal{E}_{Q_\mu},$$

that is

$$0.000673593x_{c1}^2 + 0.000085523x_{c1}x_{c2} + 0.00246288x_{c2}^2 + 0.9991y_c^2 \leq 1. \tag{4.1}$$

From Equation (4.1), we deduce:

$$\begin{cases} |x_{c1}| & \leq & 3 \cdot 10^5 \sqrt{\frac{13\,695}{829\,322\,227\,639}} & < & 38.5515 \\ |x_{c2}| & \leq & 10^5 \sqrt{\frac{33\,710}{829\,322\,227\,639}} & < & 20.1614 \\ |y_c| & \leq & \frac{100}{\sqrt{9991}} & < & 1.00046 \end{cases} \tag{4.2}$$

Here we are able to find algebraic solutions, but this may be impossible with ellipsoids of higher dimension. Still, we would be able to find bounds using numerical methods.

In floating-point arithmetic, rounding errors created by addition and multiplication operators can be bounded when the operands are known or bounded by Equation (4.2), provided that overflow, underflow, and denormalized numbers do not occur [Gol91, Hig02].

Here, we need to compute

$$\overline{C_c}x_c + \overline{D_c}y_c = \overline{C_{c(0,0)}}x_{c1} + \overline{D_c}y_c$$

where all values in the right-hand term are known or bounded. Thus, a constant ε can be computed that bounds the absolute rounding error created when computing u. We obtain:

$$\varepsilon = 5.90 \cdot 10^{-12}$$

This way, starting from the algebraic result obtained on real numbers

$$|u| \leq \sqrt{(C_c\ D_c) \cdot Q_\mu^{-1} \cdot (C_c\ D_c)^\intercal}$$

we can ensure that with floating-point numbers, the following inequality holds:

$$|u| \leq \bar{U} = \sqrt{(\overline{C_c}\ \overline{D_c}) \cdot Q_\mu^{-1} \cdot (\overline{C_c}\ \overline{D_c})^\intercal} + \varepsilon$$

which leads to the invariants:

```
% (xc yc) ∈ E_{Q_μ}, Q_μ = ( μP  0_{2×1} ; 0_{1×2}  1−μ ), μ = 0.9991
10  u = Cc*xc + Dc*yc;
% (xc yc) ∈ E_{Q_μ}, u² ≤ Ū²
```

Listing 4.6: Computation of u, modified

4.3.2 Invariant on x_c

The next instruction, considering changes of constants, is:

```
11    xc = Ac*xc + Bc*yc;
      % x_c ∈ ℰ̃_R, R̃ = [(Ā_c  B̄_c) · Q_μ^{-1} · (Ā_c  B̄_c)ᵀ]^{-1},
```

Listing 4.7: Computation of x_c, original code

where $\tilde{R}$ is defined the same way R is, using floating-point terms $\overline{A_c}$, $\overline{B_c}$ instead of the real-valued counterparts A_c, B_c, and $\widetilde{\mathcal{E}_R}$ is the ellipsoid built upon $\tilde{R}$. Again, this invariant holds independently of matrix values.

Here, we compute the values assigned to $x_c = \binom{x_{c1}}{x_{c2}}$:

$$\begin{cases} x_{c_1} := \overline{A_c}_{(0,0)} x_{c1} + \overline{A_c}_{(0,1)} x_{c2} + y_c \\ x_{c_2} := \overline{A_c}_{(1,0)} x_{c1} + x_{c2} \end{cases}$$

Using the same method as above, absolute rounding errors introduced by floating-point operations can be bounded on dimensions x_{c1} and x_{c2} by constants

$$\varepsilon_1 = 7.42 \cdot 10^{-15} \qquad \text{and} \qquad \varepsilon_2 = 3.62 \cdot 10^{-15}.$$

These constants must be taken into account in the postcondition. Then the postcondition can be replaced by

```
% x_c ∈ ℰ_R
```

where $\overline{\mathcal{E}_R}$ is an ellipse that includes $\widetilde{\mathcal{E}_R}$ plus the rounding error terms (see Figure 4.1). As mentioned in Section 3.2, replacing the ellipse in the postcondition by another ellipse has the advantage of introducing little change in the stability proof sketch (instead of using a different domain, which would involve different theorems), which can greatly facilitate tweaking the rest of the proof in longer codes. Formally, $\overline{\mathcal{E}_R}$ must satisfy:

$$\forall x_c \in \widetilde{\mathcal{E}_R}, \forall {x_c}' \in \mathbb{R}^2, |x_{c1}' - x_{c1}| \leq \varepsilon_1 \wedge |x_{c2}' - x_{c2}| \leq \varepsilon_2 \Longrightarrow {x_c}' \in \overline{\mathcal{E}_R} \tag{4.3}$$

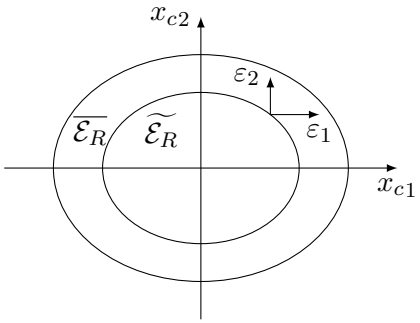


Figure 4.1: Relation between $\mathcal{E}_R$ and $\mathcal{E}_R$

At the end of the proof scheme, the system is stable with floating-point numbers if and only if the inclusion

$$\overline{\mathcal{E}_R} \subset \mathcal{E}_P$$

holds. To succeed, $\overline{\mathcal{E}_R}$ should be as narrow as possible with respect to Equation (4.3). This is not a clear criterion, as several shapes are possible for $\overline{\mathcal{E}_R}$ with no clear winner. Indeed, different criteria can be taken into consideration: area, bounding box, shape, or a combination of criteria… We propose to define $\overline{\mathcal{E}_R}$ as the smallest homothety of $\widetilde{\mathcal{E}_R}$ centered around 0 that

satisfies Equation (4.3). It can be computed rather easily, for any number of dimension; we give details for two dimensions.

Let a, b, c be the coefficients of $\widetilde{\mathcal{E}_R}$:

$$\widetilde{\mathcal{E}_R} = \{(x_{c1}, x_{c2}) \,|\, ax_{c1}^2 + bx_{c2}^2 + cx_{c1}x_{c2} \leq 1\}.$$

a, b and c are known, positive values. Then there exists $k \geq 0$ such that

$$\overline{\mathcal{E}_R} = \{(x_{c1}, x_{c2}) \,|\, ax_{c1}^2 + bx_{c2}^2 + cx_{c1}x_{c2} \leq k\}.$$

As $\overline{\mathcal{E}_R}$ is wider than $\widetilde{\mathcal{E}_R}$, $k \geq 1$. We need a condition on k that guarantees Equation (4.3).

We consider a point (x_{c1}, x_{c2}) located on the border of $\widetilde{\mathcal{E}_R}$:

$$ax_{c1}^2 + bx_{c2}^2 + cx_{c1}x_{c2} = 1. \tag{4.4}$$

By construction, for any values $e_1, e_2 \in \mathbb{R}$ such that $|e_1| \leq \varepsilon_1 \wedge |e_2| \leq \varepsilon_2$ the relation

$$(x_{c1} + e_1, x_{c2} + e_2) \in \overline{\mathcal{E}_R}$$

must hold, that is to say:

$$a(x_{c1} + e_1)^2 + b(x_{c2} + e_2)^2 + c(x_{c1} + e_1)(x_{c2} + e_2) \leq k.$$

It develops into

$$(ax_{c1}^2 + bx_{c2}^2 + cx_{c1}x_{c2}) + (2ae_1 + ce_2)x_{c1} + (2be_2 + ce_1)x_{c2} + (ae_1^2 + be_2^2 + ce_1e_2) \leq k,$$

that is

$$1 + (2ae_1 + ce_2)x_{c1} + (2be_2 + ce_1)x_{c2} + (ae_1^2 + be_2^2 + ce_1e_2) \leq k$$

due to Equation (4.4).

Greatest values for the left-hand term are reached with $|e_1| = \varepsilon_1 \wedge |e_2| = \varepsilon_2$, depending on the signs of x_{c1} and x_{c2}. As the ellipse $\widetilde{\mathcal{E}_R}$ is symmetric around the origin point $(0, 0)$, we can set $e_1 = \varepsilon_1$, which leaves only two cases to study. Finally, we numerically verify that greatest values of the term are reached when $e_2 = \varepsilon_2$. This is the only case we detail here.

We can write:

$$1 + \alpha x_{c1} + \beta x_{c2} + \gamma \leq k$$

with values $\alpha = (2a\varepsilon_1 + c\varepsilon_2)$, $\beta = (2b\varepsilon_2 + c\varepsilon_1)$ and $\gamma = (a\varepsilon_1^2 + b\varepsilon_2^2 + c\varepsilon_1\varepsilon_2)$.

We know that x_{c1} and x_{c2} are bounded, thus so is the term $\alpha x_{c1} + \beta x_{c2}$: we can compute a minimum bound δ such that $\alpha x_{c1} + \beta x_{c2} \leq \delta$. So it is sufficient that k satisfies:

$$k \geq 1 + \gamma + \delta.$$

Consequently, the smallest homothety of $\widetilde{\mathcal{E}_R}$ that satisfies Equation (4.3) is obtained with $k = 1 + \gamma + \delta$; we take it as our definition of $\overline{\mathcal{E}_R}$. The instruction becomes:

```
11    xc = Ac*xc + Bc*yc;
      % x_c ∈ Ē_R
```

Listing 4.8: Computation of x_c, modified

In our case, starting from the ellipse

$$\widetilde{\mathcal{E}_R} = \{(x_{c1}, x_{c2}) \,|\, 0.00269007x_{c1}^2 + 0.000341414x_{c1}x_{c2} + 0.00247323x_{c2}^2 \leq 1\}$$

we get the following values:

$$\alpha = 1.03246 \cdot 10^{-17}, \beta = 1.84829 \cdot 10^{-17}, \gamma = 7.17582 \cdot 10^{-32}.$$

Our program finds $\delta = 5.35754 \cdot 10^{-16} \gg \gamma$ and finally

$$k = 1 + 5.35754 \cdot 10^{-16}$$

that gives $\overline{\mathcal{E}_R}$.

4.3.3 End of Proof Scheme

Then, what remains of the stability proof scheme is only to handle inputs and outputs. No change is required and this becomes:

```
      % x_c ∈ overline(E_R)
12    send(u, 1);
      % x_c ∈ overline(E_R)
13    receive(y, 2);
      % x_c ∈ overline(E_R)
14    receive(yd, 3);
      % x_c ∈ overline(E_R)
15    skip;
      % x_c ∈ E_P
16 end
```

Listing 4.9: End of proof scheme

The only step to check carefully is at line 15. At this stage, the final assertion $\overline{\mathcal{E}_R} \subset \mathcal{E}_P$ holds and is checked successfully.

4.4 Closed-Loop Stability Proof Scheme with Floating-Point Numbers

We now show how the proof of state boundedness of the closed-loop system specifications can be migrated to the level of the controller code and executable model of the system. To be more precise, we exploit the invariance of the ellipsoid $\mathcal{E}_P$ to develop a proof of proper behavior, that is, stability and variable boundedness, for the computer program that implements the controller as it interacts with the physical system. Unlike the developments related to the open-loop controller, this proof necessarily involves the presence of a model of the physical system. In [Fer10], Feron chooses to represent the physical system and the computer program by two concurrent programs. The code for controller dynamics is unchanged, same as in Section 2.2. The pseudocode to represent the physical system dynamics is shown below.

```
1 Ap = [1.0000, 0.0100; -0.0100, 1.0000];
2 Bp = [0.00005; 0.01];
3 Cp = [1, 0];
4 while (1)
5     yp = Cp * xp;
6     send(yp, 2);
7     receive(up, 1);
8     xp = Ap * xp + Bp * up;
9 end
```

Listing 4.10: Plant pseudocode

In this scheme, the computer program representation of the physical system is to remain unchanged, since it only exists for modeling purposes and does not correspond to any actual program, whereas the controller code is allowed to evolve to reflect the various stages of its implementation.

Establishing proofs of stability of the closed-loop system at the code level is necessarily tied to understanding the joint behavior of the controller and the plant. The entire state space therefore consists of the direct sum of the state spaces of the controller and the physical system. The approach described in the previous sections is used to document the corresponding system of two

processes. One interesting aspect of these processes is their concurrency, which can complicate the structure of the state transitions. However, a close inspection of the programs reveals that the transition structure of the processes does not need to rely on the extensions of Hoare's logic to concurrent programs: one program at a time is running, through the blocking nature of the `receive` primitive.

Feron's stability proof with real numbers is much longer than for the open-loop system. We do not detail it, the interested reader is referred to [Fer10] for full information. To be noticed, the resulting invariants are not much more complex than those available from the study of the controller alone in the open-loop case. On the good side, as already mentioned, the Hoare formalism is not significantly affected by the concurrent structure of the closed-loop system.

A floating-point representation of the closed-loop system consists of keeping the listing corresponding to the physical system in its real-number settings, while replacing the controller part with the corresponding floating-point implementation, as we did in the first part of this chapter. Using similar techniques to the study of the controller alone, proof invariants can be tweaked to take into account constant changes and rounding errors resulting from the use of floating-point arithmetic in the controller and real arithmetic in the plant.

Unfortunately, these invariants are not sufficient to show that the stability condition holds at the end of the loop body in the case of 64-bit floating-point numbers. In this case, our tool cannot prove the system stability on a double-precision floating-point architecture: either the system is not stable with the floating-point based controller, and in this case the proof parameters ($\mathcal{E}_P$, μ, etc.) must be chosen more carefully by the controller designer, or stability holds but we lost it by overapproximating the errors. Tuning the precision of floating-point arithmetic, as described in Section 4.1, teaches us that the controller stability holds on a quadruple-precision platform (i.e., with 128 bits). Thus, an alternative solution might be to use a microcontroller device with superior precision.

4.5 Alternative Limited-Precision Arithmetics

Our general idea is to replace some of the invariants in the original proof scheme by wider ones that include rounding errors, with the hope that the stability condition is strong enough and still holds. This approach is made possible by the fact that the rounding errors introduced by the operations used in the code are bounded on bounded inputs and bounded state variables.

In previous sections, we mostly dealt with the case of a floating-point representation of real numbers. They are not available on all architectures, especially on microcontrollers that are commonly used to implement control systems. In this section, we quickly discuss alternative real-number representations.

- We can deal with fixed-point arithmetic the same way we do with floating-point, as long as we stand far enough from extremal values that can lead to overflows or extremely large error terms;

- Another way to represent real numbers is to use two integers, a numerator and a denominator. Considering that the input values are exact, the elementary operations do not introduce rounding errors but can easily lead to overflows, e.g. when computing

$$\frac{p_1}{q_1} + \frac{p_2}{q_2} = \frac{p_1 q_2 + p_2 q_1}{q_1 q_2}.$$

  A strategy must be used to prevent overflows by introducing approximations: in this case, the question is to quantify the errors introduced by these approximations.

In our example, we exclusively used additions and multiplications: divisions are not involved in linear control. Still, programs with divisions can also be analyzed, if the numerator can be shown to be far enough from zero: it is a supplementary constraint, but it is reasonable to assume that it should be respected on a realistic control system that uses divisions. Differentiable, periodic functions such as (sin) can be computed with a table of values (abacus) and an interpolation function, thus with bounded error. In the same way, functions not periodic, but restricted to finite domains, can also be approximated. Other functions, such as tangent or square root [Ner13], could raise more issues.

Part II

Linear Relation Analysis

Chapter 5

Models, Programs, Verification

Dans ce chapitre, nous donnons les principales définitions et notations relatives aux programmes, leur sémantique, et les propriétés que nous voulons vérifier les concernant. Tout d'abord, nous introduisons les systèmes de transition (section 5.1) qui constitueront le modèle sémantique standard des programmes ; nous définissons ensuite les propriétés de ces systèmes, et la problématique générale de la vérification (section 5.2). Les automates interprétés, un modèle de programme très général, sont introduits dans la section 5.3.

* * *

In this chapter, we give the main definitions and notations related to programs, their semantics, and the properties we want to check about them. First, we introduce transition systems (Section 5.1) that will constitute the standard semantic model of programs, then we define the properties of theses systems, and the general problematic of verification (Section 5.2). In Section 5.3, we introduce interpreted automata, a very general program model.

5.1 Transition Systems

Transition systems are the usual semantic models of programs. We give here definitions and notations related to transition systems.

Definition 5.1 (Transition System). A *transition system* is a triple $(Q, \rightarrow, Q_{\text{init}})$ where Q is a set of states, $Q_{\text{init}} \subseteq Q$ is a subset of states called *initial states*, and $\rightarrow \subseteq Q \times Q$ is a binary relation over Q called *transition relation*.

Considering two states $q, q' \in Q$, the relation $(q, q') \in \rightarrow$ is usually noted: $q \rightarrow q'$.

Definition 5.2 (Successor State, Predecessor State). A state q' is a *successor* of a state q if $q \rightarrow q'$. Likewise, q is a *predecessor* of q'.

Definition 5.3 (Path, Trace). A *path* from a state q to a state q' is a sequence $(q_0, \ldots, q_n)$ such as $q_0 = q$, $q_n = q'$ and: $\forall i < n, q_i \rightarrow q_{i+1}$. A *trace* is a path whose origin is a state of Q_{init}.

Definition 5.4 (Accessible State, Coaccessible State). A state q' is *accessible* from a state q (likewise, q is coaccessible from q') if there exists a path between q and q', that is if (q, q') belong to the transitive closure of $\rightarrow$.

5.2 Verification

We consider the framework of property verification (as opposed, for example, to program comparison [BRSV90, Fer90]). More specifically, we consider a linear semantics (as opposed, for

example, to arborescent semantics [Lam80]): a behavior of a transition system is a trace, a property is a set of traces. We are essentially interested in *safety* properties.

5.2.1 Properties

As said, we consider that a property of a transition system is a set of traces. In this paragraph, we assume that these traces are infinite. Let Q^ω be the set of infinite sequences of states in Q. Two important classes of properties are usually distinguished [Lam77, AS87], mainly because different techniques are used to verify them. Informally:

- A *safety* property expresses that "something bad" will never happen during the execution.
- A *liveness* property expresses that "something good" will necessarily happen at some point as the system is executing.

The violation of a safety property can be detected by a finite execution. Indeed, if a trace $\sigma = (q_0, q_1, \dots)$ violates the property, then there exists a state q_i in σ where "something bad" occurs and this is definitive since the property supposes that it never happens. So, any trace having $(q_0, q_1, \dots, q_i)$ as prefix (noted $\sigma[\dots i]$) violates the property. A trace satisfies the safety property P if and only if every finite prefix verify P or, more exactly, if every finite prefix can be extended into a trace that verifies P. Thus, we discuss the satisfaction of a safety property by a *finite* trace.

Definition 5.5 (Safety Property). $P \subseteq Q^\omega$ is a safety property if and only if, for any trace σ:

$$\sigma \in P \iff \forall i \geq 0, \exists \sigma' \in Q^\omega \text{ such as } \sigma[\dots i] \cdot \sigma' \in P \tag{5.1}$$

where $\sigma[\dots i]$ is the finite sequence of the i first terms of σ, and "$\cdot$" is the sequence concatenation operator.

Usual examples of safety properties include the sequential behavior of transition systems, mutual exclusion in products of transition systems, the absence of division by zero, the validity of array accesses or pointer dereference, or the absence of deadlocks.

If P is a liveness property, no finite execution is definitively bad, since "something good" can always happen in the future. Every finite execution can be prolonged in a trace that satisfies P:

Definition 5.6 (Liveness Property). $P \subseteq Q^\omega$ is a liveness property if and only if

$$\forall \sigma \in Q^*, \exists \sigma' \in Q^\omega \text{ such as } \sigma \cdot \sigma' \in P. \tag{5.2}$$

A usual example of liveness property is program termination.

Example. We consider a program with two threads. Each thread can be either waiting to be executed (0), working (1) or terminated (2). The state space of the program is $Q = \{0,1,2\} \times \{0,1,2\}$ and the unique initial state is $Q_{\text{init}} = \{(0,0)\}$.

The mutual exclusion property $P_{\text{mutex}} = \{\sigma \in Q^\omega \mid \forall q \in \sigma, q \neq (1,1)\}$ ensures that the two threads cannot be working at the same time. It is a safety property, as defined by Equation (5.1): indeed, given a trace $\sigma \in P$, each finite prefix $\sigma[\dots i]$ can be extended, for instance with the infinite prefix $((0,0),(0,0),\dots)$, into a trace that verify P_{mutex}.

The termination property $P_{\text{join}} = \{\sigma \in Q^\omega \mid \exists q \in \sigma, q = (2,2)\}$ expresses that the program terminates at some point during execution. It is a liveness property, according to Equation (5.2): given any set of states $\sigma \in Q^*$, the trace $\sigma \cdot ((2,2),(2,2),\dots)$ belongs to P_{join}. However, it is not

a safety property: the trace $\sigma = ((0,0),(0,0),\dots)$ does not belong to P_{join}, still any finite prefix $\sigma[\dots i]$ can be extended with $((2,2),(2,2),\dots)$ into a trace verifying P_{join}, which contradicts Equation (5.1).

In this dissertation, we are only interested in verifying safety properties.

5.2.2 Accessibility and Verification

As mentioned above, a safety property expresses that "something bad never happens". Accessibility properties constitute a subset of safety properties, expressing that a certain set of states — representing "something bad", and noted Q_{err} — cannot be reached. Their verification involves the computation of accessible states (from the initial states Q_{init}), or of coaccessible states (from the error states Q_{err}).

We introduce a few usual definitions relative to accessibility. Let X be a subset of Q.

Definition 5.7 (Postconditions, Preconditions). We note:

$$\text{Post}(X) = \{q' \in Q \mid \exists q \in X, q \to q'\} \qquad \text{and} \qquad \text{Pre}(X) = \{q \in Q \mid \exists q' \in X, q \to q'\}.$$

$\text{Post}(X)$ is the set of successor states of all states in X. Similarly, $\text{Pre}(X)$ is the set of all predecessors of states in X.

Definition 5.8 (Accessible States, Coaccessible States). We note $\text{acc}(X)$ (respectively $\text{coacc}(X)$) the set of all accessible states (respectively coaccessible states) from some state in X. As $\text{acc}(X)$ (respectively $\text{coacc}(X)$) is the smallest solution of the fixpoint equation $Y = X \cup \text{Post}(Y)$ (respectively $Y = X \cup \text{Pre}(Y)$), we have:

$$\text{acc}(X) = \bigcup_{n \geq 0} \text{Post}^n(X) \qquad \text{and} \qquad \text{coacc}(X) = \bigcup_{n \geq 0} \text{Pre}^n(X).$$

It appears that there are two ways to verify an accessibility property:

- Compute $\text{acc}(Q_{\text{init}})$ and verify that $\text{acc}(Q_{\text{init}}) \cap Q_{\text{err}} = \emptyset$.
- Compute $\text{coacc}(Q_{\text{err}})$ and verify that $Q_{\text{init}} \cap \text{coacc}(Q_{\text{err}}) = \emptyset$.

Therefore, the crux of verifying accessibility properties is fixpoint computation. In some cases, this compution can be done exactly: in these cases, we speak of *model checking* [QS82, CES86] (Section 5.2.3). But in the general case, only *approximations* of fixpoints can be computed automatically: then, we speak of *conservative verification* (Section 5.2.4).

5.2.3 Model Checking

Model checking mostly concerns *finite-state* systems. In this case, iterative fixpoint computations always converge and all properties are, theoretically, decidable. Efforts on model checking are about, on the one hand, the verification of very general properties — both of safety and of liveness, and with linear as well as arborescent semantics (temporal logic) — and, on the other hand, the optimization techniques allowing to work on huge sets of states. As the verification is exact, its conclusion consists either in a satisfaction verdict, or in an error verdict that can come with a counterexample.

Moreover, model checking is also used in the case of infinite state systems, when they can be abstracted (ignoring some aspects) into finite systems [GL93, CGL94]. In this case, we talk of *abstract* model checking. It should be noted that in case of failure, this kind of verification cannot conclude that the property is not satisfiable, as the failure can be due to the approximations that were made during abstractions.

5.2.4 Conservative Verification

If every type of property requires fixpoint computations, approximate computations can be enough in the case of accessibility properties: if we compute a superset Q' of $\mathrm{acc}(Q_{\mathrm{init}})$ (respectively, of $\mathrm{coacc}(Q_{\mathrm{init}})$) and if it does not intersect Q_{err} (respectively, Q_{init}), then the property is certainly satisfied (Figure 5.1). However, if the intersection is not empty, we do not know whether the property is violated or if the overapproximation used is too inaccurate. Conversely, if we compute a subset of $\mathrm{acc}(Q_{\mathrm{init}})$ (respectively, of $\mathrm{coacc}(Q_{\mathrm{err}})$) that intersects Q_{err} (respectively, Q_{init}), the property is certainly violated. In both cases, we speak of *conservative* verification: in case of success, the verification provides a guaranteed result, but in case of failure, it is inconclusive.

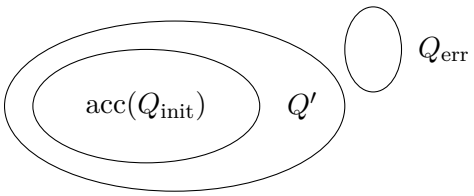


Figure 5.1: Conservative Verification

The approximation of fixpoints is the central theme of the theory of *abstract interpretation* that is presented in Chapter 6.

5.3 Program Models

Transition systems constitute a very low-level model. We need a higher-level model of programs, but, as we do not want to rely on a particular programming language, we choose a relatively abstract and general model: interpreted automata. We give in Section 5.3.2 a semantics of interpreted automata in terms of transition systems. Also, the translation of a safety property into an accessibility property is defined at the interpreted automaton level (Section 5.3.5).

5.3.1 Interpreted Automata

An interpreted automaton [bor95] is composed of a finite set of *control points* K, including one initial point. Each control point is mapped to a set of guarded commands, that act on variables and lead to other control points.

Let X be a finite set of typed variables (for example, integer and boolean variables). A *valuation* is a function mapping each variable to one value of its type. Let Val be the set of valuations on X.

Definition 5.9 (Guard, Action, Guarded Command).

- A *guard* g is a function $g : \mathrm{Val} \to \mathbb{B} = \{\top, \bot\}$, giving the value of a logic formula on a valuation V. In practice, the guard is assimilated to its truth set (the set it is a characteristic function of): $\{V \mid g(V) = \top\}$. The set of guards is noted Guards.
- An *action* a is a binary relation on valuations ($a \subseteq \mathrm{Val} \times \mathrm{Val}$). The set of actions is noted Act.
- A *command* is a quadruplet $c = (k, g, a, k') \in K \times \mathrm{Guards} \times \mathrm{Act} \times K$. Intuitively, when the automaton is at control point k, and if the current valuation V satisfies the guard g (i.e. $g(V) = \top$), the command may be executed, and when executed it moves to the control point k', with a valuation V' such that $(V, V') \in a$.

The actions are by default nondeterministic. In particular, a read action (noted "$x := ?$") maps a valuation V to any valuation V' that differs from V only by the value of x (in other words, x can take any value). Thus, the value of some variables (input variables) can be completely unspecified at some control points. At transition level, the choice of the next state can also be nondeterministic.

Definition 5.10 (Interpeted Automaton). An interpreted automaton is a triple $\alpha = (K, \mathrm{Com}, k_{\mathrm{init}})$ where K is a finite set of control points, Com is a finite set of commands, and $k_{\mathrm{init}} \in K$ is the initial control point.

Interpreted automata are a refinement of finite state machines: they have state and transitions, but also variables that are involved in transitions. Also, the terminology is different: transitions in finite state machines are called commands and states are called control points.

An example of interpreted automaton is given in Figure 5.2. It corresponds to the interpreted automaton $(K, \mathrm{Com}, k_{\mathrm{init}})$ with $K = \{k_{\mathrm{init}}, k_1\}$, $\mathrm{Com} = \{c_1, c_2, c_3\}$ and

$$\begin{aligned} c_1 &= \big(k_{\mathrm{init}}, (\lambda x, y : \cos y \leq x), \{(x, y : x + 1, y) \,|\, x, y \in \mathrm{Val}\}, k_1\big) \\ c_2 &= \big(k_1, (\lambda x, y : \top), \{(x, y, x, y + 2) \,|\, x, y \in \mathrm{Val}\}, k_1\big) \\ c_3 &= \big(k_1, (\lambda x, y : x \neq y), \{(x, y, x, y^x) \,|\, x, y \in \mathrm{Val}\}, k_{\mathrm{init}}\big) \end{aligned}$$

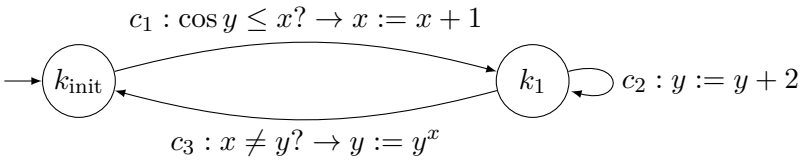


Figure 5.2: An Interpreted Automaton

5.3.2 Semantics in Terms of Transition Systems

We give the semantics of interpreted automata in terms of transition systems. Let $(K, \mathrm{Com}, k_{\mathrm{init}})$ be an interpreted automaton. It is associated to a transition system $(Q, \to, Q_{\mathrm{init}})$ defined as follows:

- The set of states is the set of pairs (k, V) where k is a control point and V is a valuation: $Q = K \times \mathrm{Val}$.
- Initial states have k_{init} as control point, and any valuation: $Q_{\mathrm{init}} = \{k_{\mathrm{init}}\} \times \mathrm{Val}$.
- The transition relation is derived from the command semantics:

$$(k, V) \to (k', V') \iff \exists (k, g, a, k') \in \mathrm{Com}, g(V) = \top \wedge (V, V') \in a.$$

5.3.3 Example

We illustrate the modeling of a program by an interpreted automaton on a very small example, from [Hal93]. We want to model a car

1. Whose speed is always lower or equal to 2 m/s.
2. That stops after 4 seconds.
3. And should not collide into a wall after 10 meters when leaving the track.

Our purpose is to show that, under assumptions (1) and (2), the car stops before hitting the wall.

The simulation program has two boolean inputs m and s (respectively true when a meter is covered and a second is elapsed, the events are supposed to be not simultaneous), read at every transition. The program counts the distance traveled (d, a counter of meters), the elapsed time (t, a counter of seconds) and the instant speed (v, a counter of meters by second, reset at each second), stops the car if it goes too fast, or hits the wall after 10 m are covered.

The corresponding automaton is represented in Figure 5.3. As in [Mer05], the automaton can be slightly modified to only take into account numerical aspects (Figure 5.4). We get rid of the inputs, and we can see that state k_{fast} is not accessible. It turns out that, as the system is finite-state, the corresponding transition system is also finite. On this transition system, it is easy to verify that the state k_{wall} is unreachable.

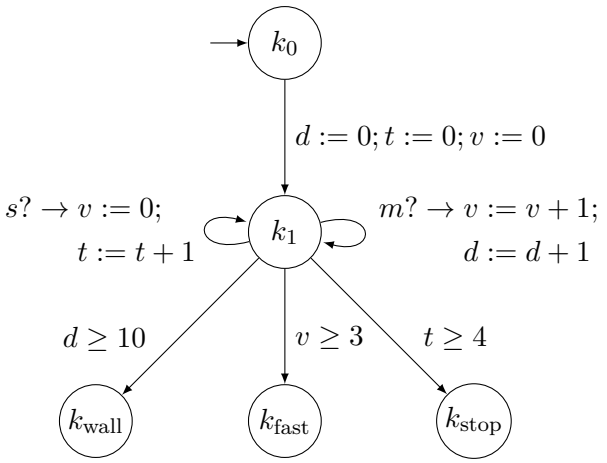


Figure 5.3: Interpreted automaton with boolean input simulating a car controller

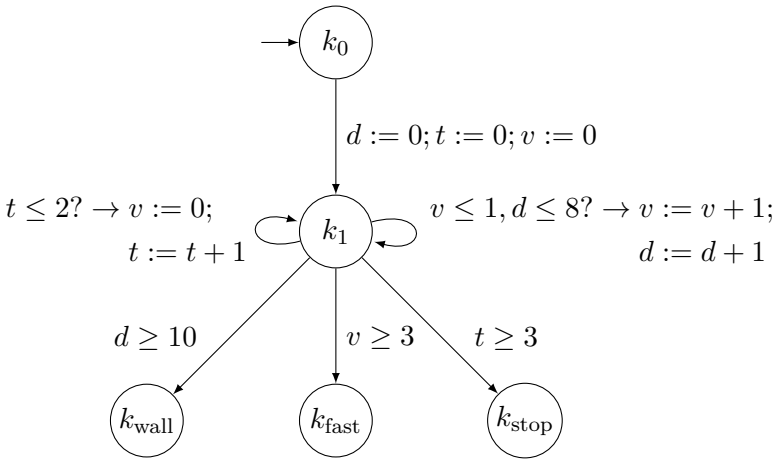


Figure 5.4: The Car Example

Remarks.

- As this example is finite-state, model checking can be used.
- If the constants appearing in the hypotheses (2 m/s, 4 s, 10 m) are increased, the number of states in the transition system tends to explode, making model checking difficult. If these constants are parameters, given symbolically, then model checking cannot be used.

5.3.4 Collecting Semantics

To verify an accessibility property in an interpreted automaton, it is necessary to compute, or approximate, either the set $\mathrm{acc}(\{k_{\mathrm{init}}\} \times \mathrm{Val})$ of accessible states in the associated transition system, either the set $\mathrm{coacc}(\{k_{\mathrm{err}}\} \times \mathrm{Val})$ of coaccessible states, assuming that error states are characterized by a control point k_{err} (in the car example, the control point k_{wall}).

To this end, instead of directly applying the semantics defined in Section 5.3.2, we can try to take advantage of the control structure of the automaton to break up the problem.

For every $k \in K$, the set of accessible valuations (respectively, coaccessible valuations) at point k is noted A_k (respectively, C_k):

$$A_k = \{V \mid (k, V) \in \mathrm{acc}(\{k_{\mathrm{init}}\} \times \mathrm{Val})\} \quad \text{and} \quad C_k = \{V \mid (k, V) \in \mathrm{coacc}(\{k_{\mathrm{err}}\} \times \mathrm{Val})\}.$$

Of course, we have:

$$\mathrm{acc}(\{k_{\mathrm{init}}\} \times \mathrm{Val}) = \bigcup_{k \in K} A_k \qquad \text{and} \qquad \mathrm{coacc}(\{k_{\mathrm{err}}\} \times \mathrm{Val}) = \bigcup_{k \in K} C_k.$$

Then, the sets $(A_k)_{k \in K}$ (respectively, $(C_k)_{k \in K}$) can be caracterized as the smallest solution of a *forward* (respectively, *backward*) *system of semantics equations*, defined as follows:

- Forward system:

$$A_{k_{\mathrm{init}}} = \mathrm{Val} \qquad \text{and} \qquad \forall k \neq k_{\mathrm{init}}, A_k = \bigcup_{(k', g, a, k) \in \mathrm{Com}} a(A_{k'} \cap g) \tag{5.3}$$

  where $a(X) = \{V' \mid \exists V \in X, (V, V') \in a\}$ (image of X under the relation a).

- Backward system:

$$C_{k_{\mathrm{err}}} = \mathrm{Val} \qquad \text{and} \qquad \forall k \neq k_{\mathrm{err}}, C_k = \bigcup_{(k, g, a, k') \in \mathrm{Com}} a^{-1}(C_{k'}) \cap g \tag{5.4}$$

  where a^{-1} is the inverse relation of a, and $a^{-1} = \{(V', V) \mid (V, V') \in a\}$.

For instance, in Figure 5.5, the equation associated to point k is the following:

$$A_k = a_1(A_{k_1} \cap g_1) \cup a_2(A_{k_2} \cap g_2) \cup a_3(A_k \cap g_3).$$

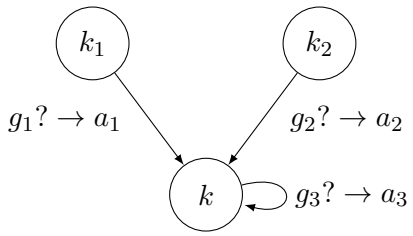


Figure 5.5: An Interpreted Automaton

Intuitively, in any non-initial point k, a valuation V accessible in k is obtained through the command (k', g, a, k) on a valuation V' accessible in k', satisfying g, and connected to V by a ($(V', V) \in a$). Similarly, in any point k different from the target k_{err}, a valuation V coaccessible in k must lead, through the command (k, g, a, k'), to a valuation V' coaccessible in k', that is to say that V must satisfy g, and that (V, V') must belong to a.

The advantage of this semantics (often called "collecting semantics") is that it gives an *equation system* of fixed points, instead of a single equation. We will show in Chapter 6 that this *partitioning* allows to greatly increase the accuracy of computations in the case of an approximated computation of the solution.

5.3.5 Composition of Interpreted Automata and Observers

The purpose of this section is to show how safety properties can be turned into accessibility properties, thanks to the notion of "synchronous observer" [HLR94]. The idea is to associate to a safety property an interpreted automaton called observer, that works as a recognizer for traces satisfying the property. Appropriatly composed with the automaton to verify (that is to say, in synchronous parallelism), the observer detects every violation of the property by joining an error control point. This way, the verification consists into making sure that the composition of the initial automaton with its observer cannot reach a state where the observer is in its error point, which is an accessibility property.

First, we define the parallel synchronous composition on interpreted automata: intuitively, two automata composed with parallel synchronisation will execute commands together, provided that the values of variables coincide on both sides — the idea being that a variable is either local to an automaton (and so, left unspecified in the other), or computed in one automaton and read (and so, left unspecified) in the other.

Definition 5.11 (Synchronous product). The synchronous product $\mathcal{A} \times \mathcal{A}'$ of two interpreted automata $\mathcal{A} = (K, \mathrm{Com}, k_{\mathrm{init}})$ and $\mathcal{A}' = (K', \mathrm{Com}', k'_{\mathrm{init}})$ with a common set of variables X is the automaton $(K \times K', \mathrm{Com}^{\times}, (k_{\mathrm{init}}, k'_{\mathrm{init}}))$ where

$$\mathrm{Com}^{\times} = \{((k_1, k'_1), g \wedge g', a \cap a', (k_2, k'_2)) \,|\, (k_1, g, a, k_2) \in \mathrm{Com}, (k'_1, g', a', k'_2) \in \mathrm{Com}'\}.$$

In other words, a command of the product from a composed state (k_1, k'_1) consists in the (synchronous) composition of a command for each component: the synchronous composition of two commands (g, a) and (g', a') on a valuation V is only possible if both guards are satisfied $((g \wedge g')(V))$, and its result V' is a common result of both actions $((V, V') \in (a \cap a'))$.

Let $\mathcal{A}$ be an interpreted automaton, working on a set of variables X. Let P be a safety property.

Definition 5.12 (Observer). An *observer* of P is an interpreted automaton $\mathcal{O}$ working on a partitioned set of variables $X \uplus X'$ and such that

- Variables in X are left unspecified in $\mathcal{O}$.
- $\mathcal{O}$ has a distinctive control point, noted k_{err}, that the automaton reaches (and never leaves) if and only if the evolution of variables in X violates P. Formally, consider $((k_0, V_0 \uplus V'_0), (k_1, V_1 \uplus V'_1), \ldots, (k_n, V_n \uplus V'_n))$ a trace in $\mathcal{O}$, then

$$k_n \neq k_{\mathrm{err}} \iff ((k_0, V_0), (k_1, V_1), \ldots, (k_n, V_n)) \in P.$$

Proposition 5.13. If $\mathcal{O}$ is an observer of P, then $\mathcal{A}$ satisfies P if and only if $\mathcal{A} \times \mathcal{O}$ never reaches a state in $K_{\mathrm{err}} = \{(k, k_{\mathrm{err}}) \,|\, k \in K\}$.

Thus, every safety property for which an observer can be provided, can be turned into an accessibility property.

Chapter 6

Abstract Interpretation

L'interprétation abstraite est une théorie d'approximation de la sémantique des programmes informatiques fondée sur les fonctions monotones pour ensembles ordonnés, en particulier les treillis. Elle peut être définie comme une exécution partielle d'un programme pour obtenir des informations sur sa sémantique (par exemple, sa structure de contrôle, son flot de données) sans avoir à en faire le traitement complet.

La principale fonction de l'interprétation abstraite est l'analyse statique, l'extraction automatique d'informations sur les exécutions possibles d'un programme. Ce chapitre définit les notions de base de l'interprétation abstraite, qui sont utilisées dans la suite du document.

* * *

Abstract interpretation is a theory to approximate computer program semantics, based on monotonic functions on ordered sets, in particular lattices. It can be defined as a partial program execution to obtain information on its semantic (for instance, control structure, data flow) without having to fully process it.

The main function of abstract interpretation is static analysis, automatic extraction of information about possible executions of a program. This chapter defines basic notions of abstract interpretation, that are used later in the document.

6.1 Introduction

In the previous chapter, we have seen the kind of systems and properties we want to verify. In particular, we have seen

- How a safety property can be translated into an accessibility property (Section 5.3.5).
- How verifying an accessibility property can boil down to computing a set of accessible states $\mathrm{acc}(Q_{\mathrm{init}})$ or coaccessible states $\mathrm{coacc}(Q_{\mathrm{err}})$ (Section 5.2.2).
- Finally, than these sets of states were solutions of fixed-point equations (Section 5.2.2)

$$\mathrm{acc}(Q_{\mathrm{init}}) = Q_{\mathrm{init}} \cup \mathrm{Post}(\mathrm{acc}(Q_{\mathrm{init}})) \qquad \mathrm{coacc}(Q_{\mathrm{err}}) = Q_{\mathrm{err}} \cup \mathrm{Pre}(\mathrm{coacc}(Q_{\mathrm{err}}))$$

  or of systems of fixed-point equations (Section 5.3.4)

$$A_{k_{\mathrm{init}}} = \mathrm{Val} \qquad \text{and} \qquad \forall k \neq k_{\mathrm{init}}, A_k = \bigcup_{(k',g,a,k)\in \mathrm{Com}} a(A_{k'} \cap g)$$

$$C_{k_{\mathrm{err}}} = \mathrm{Val} \qquad \text{and} \qquad \forall k \neq k_{\mathrm{err}}, C_k = \bigcup_{(k,g,a,k')\in \mathrm{Com}} a^{-1}(C_{k'}) \cap g.$$

Usual theorems ensure that the considered fixed-point equations have least solutions, which are well-characterized: according to Kleene's fixed-point theorem, an equation $X = F(X)$ — F being a continuous function on a complete lattice — has a least solution

$$\mathrm{lfp}(F) = \bigsqcup_{n \geq 0} F^n(\bot) \tag{6.1}$$

where $\sqcup$ is the least upper bound operator, and $\bot$ is the least element in the complete lattice. In the present case, the complete lattice is the set of parts of Q ordered by inclusion, the upper bound operator is the union of sets of states, and the least element is the empty set.

In general, solving Equation (6.1) is impossible. When the set of system states Q is infinite, computing the ascending Kleene chain $F(\bot), F(F(\bot)), F(F(F(\bot))), \ldots$ raises two kinds of issues:

- It requires to be able to represent arbitrary sets of states and to compute on them.
- The iterative computation may not terminate.

However, it was pointed out in Section 5.2.4 that accessibility properties could be proved in a conservative way, by computing an overapproximation of the solution. The approximate resolution of fixed-point equations is the main objective of *abstract interpretation*, a domain that was formalized by Patrick and Radhia Cousot in the late 1970s [CC77a], whose main principles will be recalled in the first part of this chapter (Section 6.2). The second part provides an overview of abstract interpretations dedicated to numerical properties (Sections 6.3, 6.4 and 6.5).

6.2 Principles of Abstract Interpretation

We have identified two obstacles to the exact resolution of fixed-point equations, that are:

- The impossibility to represent complex "values" — in general, infinite sets of states — and to compute on those values.
- The impossibility, in the general case, to compute the limit of infinite iterations.

Abstract interpretation proposes approximate solutions to these two problems. On the one hand, a notion of abstract domain is proposed, that allows to compute on "abstract values" simpler than sets of states. On the other hand, depending on the choice of these abstract values, the finiteness of iterations can be guaranteed; if not, techniques are provided to extrapolate and overapproximate the limit.

6.2.1 Abstract Domains

An abstract interpretation is first characterized by a domain of abstract values. These values are used to approximate, or abstract, concrete values (sets of states) that are too complex. The notion of approximation is formalized by an ordering relation: the abstract domain must be a complete lattice, as the concrete domain. The initial fixed-point equation on concrete values is then associated to an abstract fixed-point equation, to be solved in the abstract domain. Abstract values must be chosen so that their computer representation is possible, as well as the operations necessary to the iterative computation of the abstract equation.

We have seen that the concrete lattice is $(\mathcal{P}(Q), \subseteq, \cup, \cap, Q, \emptyset)$, that is the power set of Q, ordered by inclusion, with $\cup$ and $\cap$ as upper and lower bound operations, and Q and $\emptyset$ as greatest and least elements. As the theorical framework of abstract interpretation is more

general, we note $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ this concrete lattice. The lattice of abstract values is noted $(L^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \top^\sharp, \bot^\sharp)$.

The relation between concrete and abstract values may be handled by the notion of *Galois connection*: a Galois connection between L and $L^\sharp$ is a pair of functions

$$\alpha : L \to L^\sharp \text{ (abstraction)} \qquad \text{and} \qquad \gamma : L^\sharp \to L \text{ (concretization)}$$

such that (Figure 6.1):

$$\forall x \in L, \forall y \in L^\sharp, \quad \alpha(x) \sqsubseteq^\sharp y \iff x \sqsubseteq \gamma(y).$$

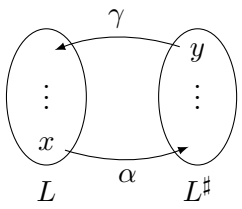


Figure 6.1: Galois Connection

As there exists a Galois connection between concrete and abstract lattices, the following theorem can be applied to the computation of fixed-point overapproximations:

Theorem 6.1 (Fixed-point overapproximation). If L and $L^\sharp$ are connected by a Galois connection (α, γ), if F is a continuous function in L [GHK+03] and G is a continuous function in $L^\sharp$, if

$$\forall x \in L, \alpha(F(x)) \sqsubseteq^\sharp G(\alpha(x)) \tag{6.2}$$

then

$$\mathrm{lfp}(F) \sqsubseteq \gamma(\mathrm{lfp}(G)).$$

This theorem means that, to compute an overapproximation of the least fixed-point of a concrete function F, we can choose or build an abstract function G satisfying Equation (6.2), compute the least fixed-point of G in the abstract lattice, and "concretize" the result.

6.2.2 Abstract Operations

It can be shown that the best choice for the abstract function G is $\alpha \circ F \circ \gamma$. However, this function is not computable in the general case, since it requires computations in the concrete domain. On the other hand, we aim to have an automatic method to compute G from F. We take advantage of the fact that the function F, in the case of program analysis, is constructed by composition of elementary functions on sets of states (the preconditions or postconditions of elementary instructions in the programming language), of unions and of intersections. Similarly, an abstract function G can be mechanically constructed from elementary abstract functions, associated to language instructions, and of upper and lower bounds operations ($\sqcup^\sharp$ and $\sqcap^\sharp$). Therefore, once the abstract lattice and the associated Galois connection are chosen, designing an abstract interpretation requires to define these abstract functions, and to provide effective algorithms.

Thus, in the case of interpreted automata, and for forward analysis (Equation (5.3), page 55), we must give

- An interpretation of guards in the abstract lattice, that is to say an effective way to associate to each guard g an abstract value $g^\sharp$ such that

$$\alpha(g) \sqsubseteq^\sharp g^\sharp \qquad (\text{or: } g \sqsubseteq \gamma(g^\sharp)).$$

- An interpretation of actions, that is to say an effective way to associate to each action a an abstract function $a^\sharp$ such that

$$\forall y \in L, \alpha(a(\gamma(y))) \sqsubseteq^\sharp a^\sharp(y) \qquad (\text{or: } \forall x \in L, a(x) \sqsubseteq \gamma(a^\sharp(x))).$$

So, the concrete equation (5.3)

$$A_k = \bigcup_{(k',g,a,k)\in \text{Com}} a(A_{k'} \cap g)$$

is associated to the abstract equation

$$A_k^\sharp = \bigsqcup_{(k',g,a,k)\in \text{Com}}^\sharp a^\sharp(A_{k'}^\sharp \sqcap^\sharp g^\sharp).$$

6.2.3 Widening and Decreasing Sequence

In many cases, the iterative fixed-point computation in the abstract lattice converges with a finite number of iterations: it is obviously the case when the abstract lattice is finite, but also when simply its depth is finite (i.e., every strictly increasing chain is finite). In these cases, the problem of infinite iterations is solved. However, it may be wise for accuracy purposes [CC92] to use more expressive lattices that do not satisfy these properties. In this case, the idea is to "guess" the limit of an infinite sequence using its first terms. To do this, the abstract interpretation comes with a "widening" operator, used to extrapolate the limit of the iterative sequence.

Definition 6.2 (Widening). A *widening* operator is a function $\nabla : L^\sharp \times L^\sharp \to L^\sharp$ that satisfies the following properties:

1. $\forall y_1, y_2 \in L^\sharp, (y_1 \sqcup^\sharp y_2) \sqsubseteq^\sharp (y_1 \nabla y_2)$.
2. For any increasing sequence $(x_n)_{n\geq 0}$ in $L^\sharp$, the sequence defined by $y_0 = x_0$, $y_{n+1} = y_n \nabla x_{n+1}$ converges in a finite number of iterations.

Theorem 6.3 (Extrapolation). If ∇ is a widening operator on $L^\sharp$ and G is a continuous function on $L^\sharp$, then the sequence $y_0 = \bot$, $y_{n+1} = y_n \nabla G(y_n)$ converges in a finite number of steps towards a limit $\hat{y}$ that is a post-fixpoint of G (i.e., $\hat{y} \sqsupseteq^\sharp G(\hat{y})$) and so, such that $\hat{y} \sqsupseteq^\sharp \text{lfp}(G)$.

A trivial example of widening operator is: $\forall x, y \in L^\sharp, x \nabla y = \top$.

The above theorem allows computing in every cases an approximation of the least fixed point of G. If the limit $\hat{y}$ is not a fixed point, the solution is improved by the following theorem:

Theorem 6.4 (Decreasing Sequence). If $\hat{y} \neq G(\hat{y})$, then the sequence $z_0 = \hat{y}$, $z_{n+1} = G(z_n)$ is decreasing, and all its terms are overapproximations of $\text{lfp}(G)$.

In short, to overapproximate the least fixed point of a function G, instead of computing the exact (and generally infinite) sequence

$$x_0 = \bot, x_{n+1} = G(x_n).$$

we first compute an extrapolation sequence

$$y_0 = \bot, y_{n+1} = y_n \nabla G(y_n)$$

that converges in a finite number of steps toward a limit $\hat{y}$, that is a correct overapproximation of $\text{lfp}(G)$. Furthermore, if $\hat{y} \neq G(\hat{y})$, the result can be improved by computing the first terms (up to any finite given precision) of the decreasing sequence

$$z_0 = \hat{y}, z_{n+1} = G(z_n)$$

that may be finite or infinite, but whose all terms are correct overapproximations of $\text{lfp}(G)$.

6.3 Abstract Interpretations of Interpreted Automata

We have seen, in Section 5.3.4, that the analysis of an interpreted automaton could lead to solving a system of fixed point equations, based on the control structure of the automaton, rather than a single equation. They are called *partitioned* systems. We focus on the "forward" system, the "backward" case being similar:

$$A_{k_{\text{init}}} = \text{Val}, \qquad \forall k \neq k_{\text{init}}, A_k = \bigcup_{(k',g,a,k)\in \text{Com}} a(A_{k'} \cap g)$$

This system is associated to the following abstract equation system:

$$A^\sharp_{k_{\text{init}}} = \top^\sharp, \qquad \forall k \neq k_{\text{init}}, A^\sharp_k = \bigsqcup^\sharp_{(k',g,a,k)\in \text{Com}} a^\sharp(A^\sharp_{k'} \sqcap^\sharp g^\sharp).$$

This system can be solved iteratively, considering the equations one by one in any order, until stabilization. The convergence speed in influenced by the order in which the equations are considered, and the application of widening can be optimized:

- It makes sense to consider equations taking into consideration the control structure: when $A^\sharp_k$ changes, the values $A^\sharp_{k'}$ associated to the control points k' that are successors of k are likely to change too. Moreover, there is no use to consider control points located downstream a strongly connected component in the control graph, before stabilizing the values associated to control points in the component.
- It is unnecessary to use widening (that leads to information loss) in every equations of the system to ensure convergence: it is sufficient to choose a set $K_\nabla \subseteq K$ of *widening control points* such that every cycle in the graph contains a point of K_∇, and to apply widening only in equations corresponding to widening points. As the choice of a minimal set of widening points is an NP-complete problem, we simply rely on an heuristic choice [Bou92].

6.4 Approximating Numeric Sets

0.5

We are interested by the verification of numerical programs, and we consider specifically the case where the state of the program is defined by a control point and a valuation of variables in a numerical set $\mathcal{N}$ ($\mathbb{Z}$, $\mathbb{Q}$ or $\mathbb{R}$). Using the semantics defined in Section 5.3.4, we need to represent subsets of $\mathcal{N}^n$ where n is the number of variables. Thus, the concrete lattice is:

$$\big(\mathcal{P}(\mathcal{N}^n), \subseteq, \cup, \cap, \mathcal{N}^n, \emptyset\big).$$

In this section, we provide a brief overview of the main abstract lattices that have been proposed in the literature to abstract this domain.

6.4.1 The Sign Lattice

A rough abstraction consists in associating to each variable an abstract value in the lattice represented in Figure 6.2. For instance, the set of points shown in Figure 6.3(a) would be abstracted by the first, strict quadrant $\{x, y \,|\, x > 0, y > 0\}$ (Figure 6.3(b)). In this lattice, abstract operations are straightforward, and, as the lattice is finite, there is no need for widening.

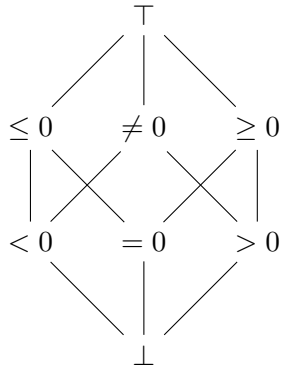
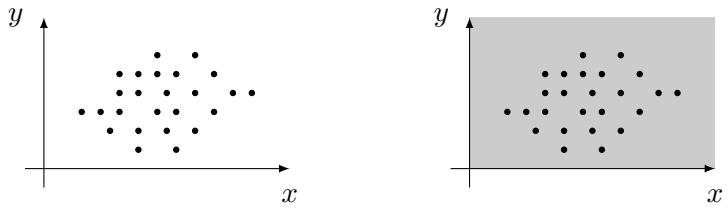


Figure 6.2: The Sign Lattice

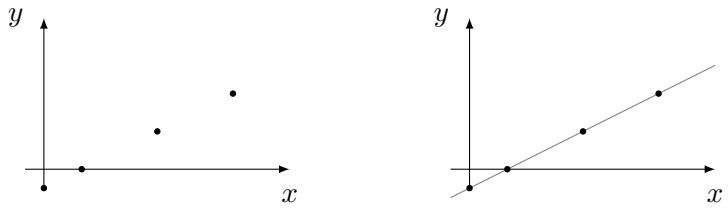


(a) (b)

Figure 6.3: Abstraction Using the Sign Lattice

6.4.2 The Lattice of Affine Equations

One of the first nontrivial abstractions, introduced by [Kar76], consists in synthesizing invariant linear equations. In this analysis, a set of points in $\mathcal{N}^n$ is abstracted by the smallest affine variety that contains it, that is to say by solution set of an affine system. Using this analysis, the set in Figure 6.3(a) would be abstracted into the whole space (no information, no equation), but the one in Figure 6.4(a) gives the equation $x - 2y = 1$ (Figure 6.4(b)). The lattice of affine equations is infinite, but its depth is finite (there are at most n linearly independent varieties in $\mathcal{N}^n$). Thus, there is no point in defining a widening operator.



(a) (b)

Figure 6.4: Abstraction Using Affine Equations

6.4.3 The Lattice of Linear Congruences

In his thesis [Gra91], Philippe Granger proposes, in particular, a lattice for abstracting parts of $\mathbb{Z}^n$ as linear congruences, in the form $a\vec{x} \equiv b \pmod{c}$ (that is, $\exists k \in \mathbb{Z}, a\vec{x} = kc+b$). For example, the set in Figure 6.5(a) as abstracted as shown in Figure 6.5(b), that is $x - 2y \equiv 2 \pmod{6}$. The lattice is infinite, with infinite depth, but the increasing chain condition is satisfied: there is no infinite sequence of strictly increasing abstract values. Once again, widening is useless.

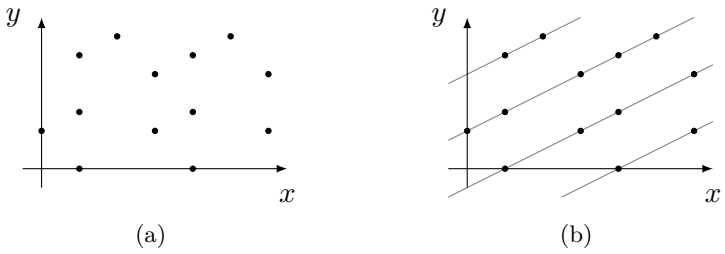


Figure 6.5: Abstraction Using Linear Congruences

6.4.4 The Lattice of Intervals

The interval lattice [CC77b] determines the variation interval of numerical variables. The abstraction of a set E consists to associate to each variable v the pair $[\min_v, \max_v]$ of minimal and maximal values (possibly $-\infty$, $+\infty$) of v in E. The abstraction by intervals of the set represented in Figure 6.6(a) is given in Figure 6.6(b). In the interval lattice, it is necessary to define a widening operator to ensure convergence. In [CC77b], the following widening operator is proposed:

$$[a,b]\nabla[c,d] = [\text{if } c < a \text{ then } -\infty \text{ else } a, \text{if } d > b \text{ then } +\infty \text{ else } b]$$

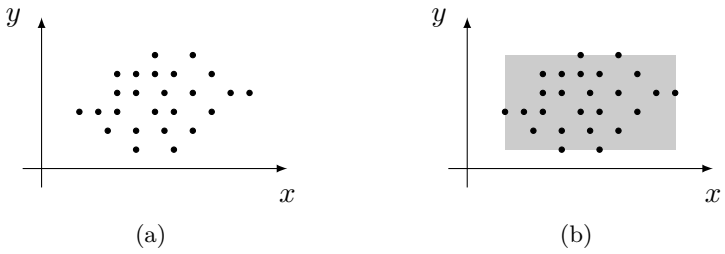


Figure 6.6: Abstraction Using Intervals

6.4.5 The Lattice of Convex Polyhedra

The lattice of convex polyhedra was first introduced by [CH78, Hal79] to analyse linear inequalities that are invariant amongst the variables of a program. The abstraction of a set of points in $\mathcal{N}^n$ is the smallest convex polyhedron that contains it (Figure 6.7). Let us notice that while this definition makes sense for a finite set of points, this is not the case of an infinite set whose convex hull is not necessarily polyhedral (for instance, a sphere). This difficulty arises from the fact that the polyhedra lattice is not complete (the upper bound of a infinite set of polyhedra is not necessarily of polyhedron). The actual, complete lattice is the lattice of convex sets in $\mathcal{N}^n$, but as the computations are always finite (thanks, in particular, to widening), the computed solutions are always polyhedra. This lattice is used in this dissertation and is discussed in greater details in Chapter 7.

6.4.6 Particular Polyhedra

As the worst-case complexity of operations on polyhedra is exponential, some subclasses have been identified for specific applications.

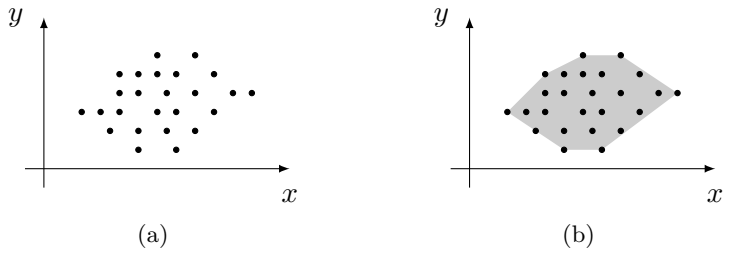


Figure 6.7: Abstraction Using Polyhedra

Areas

Areas are very particular cases of of polyhedra, whose constrains are only bounds on variables (as for intervals) or on differences of variables (Figure 6.8). In other words, all contraints are in the form $x \operatorname{op} c$ or $x - y \operatorname{op} c$ where op is an operator in $\{<, \leq, =, \geq, > \}$, and c is a constant. Areas are used to analyse timed automata [AD90, ACD93, HNSY92]. They can be represented as "difference-bound matrices", and normalized using graph algorithms.

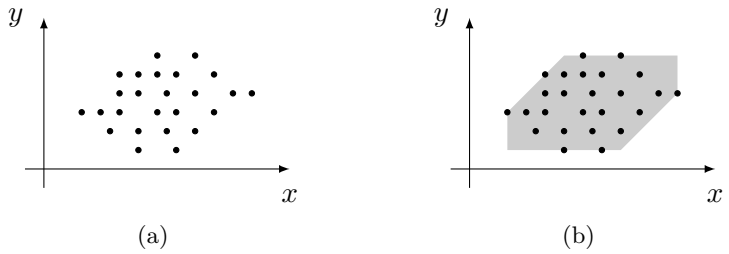


Figure 6.8: Abstraction with Areas

Octagons

Areas have been generalized, without increasing complexity, by Antoine Miné [Min06], that also allows constraints in the form $x + y \operatorname{op} c$. The resulting polyhedra are called "octagons" (Figure 6.9).

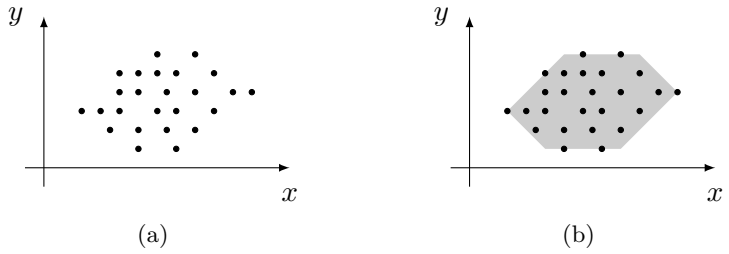


Figure 6.9: Abstraction with Octagons

Octahedra

Octagons have geen generalized too into "octahedra" [CC07], considering linear constraints whose coefficients are in $\{-1, 0, 1\}$ (so, that may involve more than two variables). However, polynomial complexity is not preserved.

6.4.7 Semi-Linear Sets and Presburger Arithmetic

We finish this short review of numerical set representation techniques by citing works less directly related to abstract interpretation, that aim to analyze in an exact way the possible values of integer variables during a program execution: some authors [BW94, BGP97, CJ98, FS00] use the decidability of Presburger arithmetic to represent more accurately infinite sets of integer vectors. As long as the sets of states to be represented are *semilinear*, algorithms to handle them are available. For some programs, the set of accessible states can be computed with *accelerations* [Gon07]. Acceleration is recognizing a fixed common pattern and exhibiting hardcoded solutions for this. It is used instead of widening in relevant cases and gives exact results.

6.5 Uses of Numerical Variables Analysis

There are various applications to the analysis of behavior of numerical variables. We classify them in two parts:

- Applications with a goal given *a priori*, in general a set of states we want to verify the inaccessibility of, and that can be taken into account by the analysis.
- Applications in which we simply want to get some information about, in general invariants, as precise as possible, to use it at a later stage.

6.5.1 Verifying Inaccessibility

We have seen in Section 5.3.5 that all safety properties can be expressed as the inaccessibility of some states. The same applies to verify the absence of errors at runtime (dynamic consistency), for which there is no need to specify the property, and that often rely on numerical aspects (divide by zero, out-of-bounds array access [NIAK01], arithmetic overflow).

In all these applications, the set Q_{err} of states we want to prove inaccessible is known, and we can proceed either "forward" — to prove $\text{acc}(Q_{\text{init}}) \cap Q_{\text{err}} = \emptyset$ — or "backward" — to prove $Q_{\text{init}} \cap \text{coacc}(Q_{\text{err}}) = \emptyset$. As it is an approximate analysis, we compute in fact either $\widehat{\text{acc}}(Q_{\text{init}})$ or $\widehat{\text{coacc}}(Q_{\text{err}})$, with

$$\text{acc}(Q_{\text{init}}) \subseteq \widehat{\text{acc}}(Q_{\text{init}}) \qquad \text{and} \qquad \text{coacc}(Q_{\text{err}}) \subseteq \widehat{\text{coacc}}(Q_{\text{err}}).$$

In terms of accuracy, it is beneficial to iterate forward and backward analyses, by computing successively

$$X_0 = \widehat{\text{acc}}(Q_{\text{init}}), \qquad Y_n = \widehat{\text{coacc}}(Q_{\text{err}} \cap X_n), \qquad X_{n+1} = \widehat{\text{acc}}(Q_{\text{init}} \cap Y_n)$$

that constitute a sequence increasingly precise, as long as the terms X_n or Y_n are not empty and do not stabilize.

6.5.2 Invariant Synthesis

A second class of applications is about invariant synthesis, with no preconceived objective, to be used later on. These applications are about:

- Compilation, with, for example, dead code elimination, finding expressions that can be computed at compile time or outside loops, the characterization of variation intervals of variables to optimize their memory implementation, the determination of the precise lifespan of array elements to improve memory management [LMC02]...
- Program parallelization [ACIK97] where an accurate analysis of inter-variable dependencies is required, especially between array elements.

- Interactive proof, where invariants found by analysis are used to make the human's work easier [BBC+00, BLO98].
- In some applications, it is necessary to add auxiliary counters to the program, and in the analysis of their behavior. For example, [Mer92] proposes a method of communication analysis in a multiprocess system by adding counters at each meeting point, analyzing the code of each process one at a time and then, combining results while equalizing counters of corresponding meetings. We can also try to show loop termination (a liveness property), or try to determine the number of iterations [PR04], by adding a loop counter and then bounding it.

Chapter 7

The Lattice of Polyhedra

Comme indiqué en introduction, une partie de ce travail est consacrée à l'analyse des relations linéaires, basée sur le treillis des polyèdres convexes. Ce chapitre présente les principaux algorithmes sur les polyèdres. Les algorithmes utilisés reposent sur deux représentations des polyèdres — par système de contraintes et par système générateur — présentée dans la section 7.1. La section 7.2 détaille le passage de l'une à l'autre des représentations, ce qui est l'algorithme clé pour effectuer toutes les autres opérations et surtout pour simplifier les représentations. La section 7.3 décrit les opérations usuelles sur les polyèdres convexes. Un exemple d'analyse est ensuite détaillé dans la section 7.4. Finalement, les principales bibliothèques polyédriques existantes sont présentées dans la section 7.5.

Les références générales de ce chapitre sont [Hal79], [Sch86] et [Jea00].

* * *

As mentioned in Introduction, part of this work is devoted to the analysis of linear relations. It is based on the lattice of convex polyhedra. This chapter presents the main algorithms relative to polyhedra. The algorithms we use are based on the two characterizations of polyhedra — by constraint systems and by generating systems — that are presented in Section 7.1. Section 7.2 details the translation from one representation to another, which is the key algorithm to perform all other operations, and most importantly to simplify representations. Section 7.3 describes the usual operations on convex polyhedra. An example of analysis is then detailed in Section 7.4. Finally, the main existing polyhedral libraries are presented in Section 7.5.

The general references of this chapter are [Hal79], [Sch86] and [Jea00].

We consider a numerical set $\mathcal{N}$ and the affine space $\mathcal{N}^n$. The scalar product of two vectors X and Y in $\mathcal{N}^n$, equals to $\sum_{i=1}^{n} X_i Y_i$, is noted $X \cdot Y$ or simply XY.

7.1 Two Representations

Definition 7.1 (Convex polyhedron). A *closed, convex polyhedron* (or simply polyhedron) is the intersection of a finite number m of closed half-spaces in $\mathcal{N}^n$.

7.1.1 Linear Constraints

Proposition 7.2 (Representation with affine constraints). Any convex polyhedron can be characterized as the solution set of a *linear constraint system* $AX \leq B$, where A is an $m \times n$ matrix and B is a vector in $\mathcal{N}^m$. We call *linear constraint* an inequation $aX \leq b$ or an equation $aX = b$, where $a \in \mathcal{N}^n$ is a line vector and $b \in \mathcal{N}$ is a scalar.

Example. Figure 7.1 represents the polyhedron corresponding to the constraint system

$$\begin{aligned} 2x_2 - x_1 &\leq 7 \\ -x_1 - x_2 &\leq -5 \\ -x_2 &\leq -1 \end{aligned}$$

in other terms, $AX \leq B$ with

$$A = \begin{bmatrix} -1 & 2 \\ -1 & -1 \\ 0 & -1 \end{bmatrix}, \quad X = \begin{bmatrix} x_1 \\ x_2 \end{bmatrix} \quad \text{and} \quad B = \begin{bmatrix} 7 \\ -5 \\ -1 \end{bmatrix}.$$

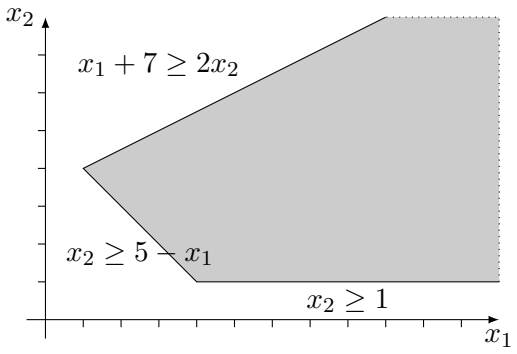


Figure 7.1: Representation of a polyhedron as a constraint system

In practice this is the representation we use, as it gives direct information on variables.

7.1.2 Non-Canonicity of Constraint Systems

In this paragraph, we focus on the non-canonicity of the representation by linear constraints. On the one hand, a constraint system may contain *redundant* constraints; on the other, when the polyhedron is contained within a strict vector subspace of $\mathcal{N}^n$ (i.e., when it satisfies linear equations), it may be characterized by several minimal constraint systems.

Definition 7.3 (Redundant constraints). A constraint $aX \leq b$ is said *redundant* in the system $AX \leq B$ if it can be removed while keeping unchanged the described polyhedron. In other words, the satisfaction of other constraints implies the satisfaction of $aX \leq b$. A constraint system with no redundant constraint is said to be *minimal*.

The proliferation of redundant constraints obviously penalizes the operations on constraint systems. It is therefore essential to be able to minimize constraint systems.

Definition 7.4 (Dimension of a polyhedron). The *dimension* of a polyhedron P is the dimension of the smallest affine subspace containing P. In other words, the dimension of P is equal to n, minus the number $\operatorname{codim} P$ of independent equations satisfied by P.

If the dimension of a polyhedron P is equal to n (i.e., $\operatorname{codim} P = 0$), and if its constraint system does not contain any redundant constraint, then this constraint system is unique, up to a multiplication factor (since constraints multiple of each other are equivalent).

Definition 7.5 (Mutually redundant constraints). Two constraints c_1 and c_2 are *mutually redundant* in a constraint system, if it it possible to remove either one of them — but not both — while leaving unchanged the described polyhedron. This situation may occur only if the polyhedron dimension is strictly inferior to n.

Example. In the system $x_2 = 0, 0 \leq x_1 \leq 1, x_1 + x_2 \geq 0$, the constraints $0 \leq x_1$ and $x_1 + x_2 \geq 0$ are mutually redundant (see Figure 7.2).

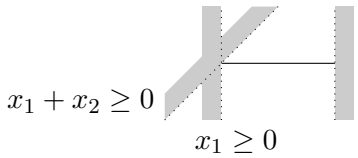


Figure 7.2: Mutually redundant constraints

As a consequence, when the dimension of a polyhedron if strictly inferior to n, it does not admit a unique, minimal representation as a constraint system.

If the origin is contained in the polyhedron P, it is possible to define a "normal" form using the scalar product. If P does not contain the origin, it can be reduced by translation. This representation will be used in Section 7.1.5 to allow the translation from one representation to another.

Definition 7.6 (Normal form of constraints). If the polyhedron P contains the origin, it is possible to obtain a normal form (A_1, A_2, A_3). These three sets separate the constraints according to their natures: an inequality with a positive constant, a zero constant or an equality to zero.

- The normal form of an inequation $aX \leq b$ with $b > 0$ is $\frac{a}{b}X \leq 1$ ($\frac{a}{b} \in A_1$).
- The normal form of an inequation $aX \leq b$ with $b = 0$ is unchanged ($a \in A_2$).
- The normal form of an equation $aX = 0$ is unchanged ($a \in A_3$).

7.1.3 Generators

Definition 7.7 (Generating system). Any convex polyhedron P can be characterized by a *generating system*, constituted by three finite sets V, R and L of vectors in $\mathcal{N}^n$, called sets of *vertices*, *rays* and *lines* of P, respectively[†1]. Then, P is the set of points that can be obtained as the sum of a *convex combination* of vertices, a *positive combination* of rays and a *linear combination* of lines:

$$P = \left\{ \sum_{v_i \in V} \lambda_i v_i + \sum_{r_j \in R} \mu_j r_j + \sum_{\ell_k \in L} \nu_k \ell_k \;\middle|\; \lambda_i \in [0,1], \sum_i \lambda_i = 1, \mu_j \geq 0, \nu_k \in \mathcal{N} \right\}.$$

Example. The polyhedron in Figure 7.1 can be represented in a dual way as a generating system (Figure 7.3), with

$$V = \left\{ v_1 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}, v_2 = \begin{bmatrix} 4 \\ 1 \end{bmatrix} \right\}, \qquad R = \left\{ r_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, r_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\}, \qquad D = \emptyset.$$

7.1.4 Non-Canonicity of Generator Systems

As for constraints, the generator representation is not canonical in the general case: it may contain redundant elements (called "non extremal" in this context), and, when the polyhedron contains a line, it admits an infinite number of minimal generating systems.

[†1] In the following, a straight line will be confounded with one of its direction vectors. Likewise, a ray will be confounded by one of its direction vectors.

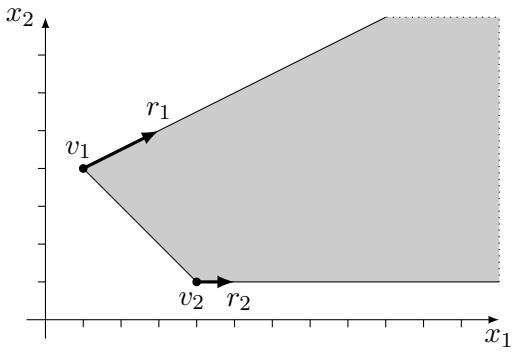


Figure 7.3: Representation as a generating system

Definition 7.8 (Extremal points and rays). A point (respectively, a ray) in a polyhedron is *extremal* if it cannot be obtained by a convex combination (respectively, positive combination) of other points in P (respectively, of other rays that are not collinear to it).

The vertices v_1 and v_2, the rays r_1 and r_2 of the example in Figure 7.3 are extremal. The point $(4, 4)$ belongs to the polyhedron, but is not extremal, the vector $(3, 1)$ is a non-extremal ray.

Definition 7.9 (Codimension of a polyhedron). The *codimension* of a polyhedron P is the dimension of the biggest affine subspace contained in P. In other words, the codimension of P is equal to the number $\dim P$ of independent lines contained in P.

A polyhedron of codimension zero (no line) admits a unique, minimal generating system (ignoring a multiplication factor for the rays), whose vertices are the extremal points of the polyhedron, and whose rays are its extremal rays (normalized).

If the polyhedron contains a line, it has no extremal point, and no unique, minimal generating system. In this case, a minimal generating system is constituted by a minimal set of lines (a basis of the biggest affine subspace included in the polyhedron), and points and extremal rays of a *section* of the polyhedron: a section of P is the intersection of P with any supplementary subspace[†2] of the biggest affine subspace included in the polyhedron.

In the example of Figure 7.4, the polyhedron contains the line ℓ of equation $x_1 = x_2$ and of direction vector $(1, 1)$. The line of equation $x_1 = 0$ is a supplementary subspace to ℓ, so the segment $-1 \leq x_2 \leq 0$ is a section of the polyhedron, with vertices $v_1 = (0, -1)$, $v_2 = (0, 0)$ and with no ray. Thus, $(\{v_1, v_2\}, \emptyset, \{\ell\})$ is a minimal generating system of the polyhedron. But we could have chosen another supplementary subspace (for instance $x_2 = 0$, or $x_1 + x_2 = 0$), so another section, and obtain another minimal generating system. If we have two lines or more, there is the choice for the basis.

7.1.5 Link between the Two Representations

Satisfaction of Constraints by the Generator Elements

Definition 7.10. Let P be a polyhedron defined by the constraint system $AX \leq B$. Then:

- Of course, a point v belongs to P if and only if $Av \leq B$.
- A vector r is a ray of P if and only if $Ar \leq 0$.
- A vector ℓ is a line of P if and only if $A\ell = 0$.

[†2]Two affine subspaces E and E' are supplementary if their intersection is reduced to a point, and if any point of the whole space is a linear combination of points in E and E'.

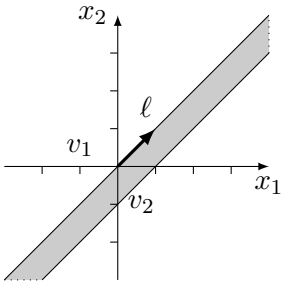


Figure 7.4: Polyhedron containing a line

Polar Polyhedron

Section 7.1.1 and Section 7.1.3 suggest a strong duality between the two representations. This duality is fully confirmed by the notion of polar polyhedron.

Definition 7.11 (Polar polyhedron). The *polar polyhedron* of P is the polyhedron P^+ defined by:

$$P^+ = \{X \in \mathcal{N}^n \,|\, \forall Y \in P, XY \leq 1\}.$$

Properties.

- P^+ always contains the origin.
- If P contains the origin, then $P^{++} = P$.
- If (V, R, L) is a generating system of P, then $X \in P^+$ if and only if:

$$\forall v \in V, v \cdot X \leq 1, \qquad \forall r \in R, r \cdot X \leq 0, \qquad \forall \ell \in L, \ell \cdot X = 0$$

- If P contains the origin and if (A_1, A_2, A_3) is the normal form of its constraint system (Definition 7.6), then (A_1, A_2, A_3) constitutes a generating system of P^+. The converse also holds.

Considering a polyhedron representation and a characterization or an algorithm on this representation, the polar polyhedron allows to define a dual property or algorithm on the other representation. For instance, with an algorithm computing the intersection of two polyhedra given as constraint systems, we can deduce an algorithm to compute the convex hull (dual operation) of two polyhedra passed as generating systems.

7.1.6 Minimization of Representations

In this section, we show how the knowledge of both representations of a polyhedron can be used to minimize these representations.

Definition 7.12 (Saturation). Let $aX \leq b$ be a linear inequation. Then a point v (respectively, a ray r) *saturates* the constraint, if and only if $av = b$ (respectively, $ar = 0$). Let $AX \leq B$ a constraint system and (V, R, L) a generating system of the same polyhedron P. Then

- For any inequation c, $\mathrm{Sat}(c)$ denotes the subset of $S \cup R$ of vertices and rays that saturate c.
- For any vertex $v \in V$ (respectively, for any ray $r \in R$), $\mathrm{Sat}(v)$ (respectively, $\mathrm{Sat}(r)$) denotes the set of inequations, in $AX \leq B$, saturated by v (respectively, by r).

Proposition 7.13 (Redundancy, extremality).

- A constraint c is redundant for P if and only if there exists another constraint c' in P such that $\mathrm{Sat}(c) \subseteq \mathrm{Sat}(c')$.

- A vertex v (respectively, a ray r) is not extremal in P if and only if there exists another vertex v' (respectively, another ray r') in P, such that $\mathrm{Sat}(v) \subseteq \mathrm{Sat}(v')$ (respectively, $\mathrm{Sat}(r) \subseteq \mathrm{Sat}(r')$).
- Two constraints c and c' in P are mutually redundant if and only if $\mathrm{Sat}(c) = \mathrm{Sat}(c')$.

The previous proposition allows to simplify the representations, and to identify mutually redundant constraints, at the cost of computing saturations (i.e., the product of the constraint matrix by the generator matrix). This computation can be complicated if there are many redundant elements, which increase the sizes of matrices. So we try to identify redundant elements earlier, when translating one representation into the other one (see Section 7.2).

In the next sections, we use the following definitions.

Definition 7.14 (Adjacency). Let P be a polyhedron of dimension $n - \operatorname{codim} P$ and codimension $\dim P$. Then

- Two vertices v and v' are *adjacent* if and only if they saturate $(n - \operatorname{codim} P - \dim P - 1)$ common inequations.
- A vertex v and a ray r are *adjacent* if and only if they saturate $(n - \operatorname{codim} P - \dim P - 1)$ common inequations.
- Two rays r and r' are *adjacent* if and only if they saturate $(n - \operatorname{codim} P - \dim P - 2)$ common inequations.

7.2 Computing the Dual Representation

Motzkin [MRTT53] has described an iterative algorithm that solves the problem of computing the dual form. A similar algorithm has been described by Chernikova [Che68] and rediscovered in [Hal79]. Thereafter, Le Verge [LV94] has proposed an optimization by improving the detection of redundant elements.

In practice, these algorithms do not manipulate polyhedra of dimension n but cones of dimension $n+1$, centered on the origin. A cone is represented by a vertex (the origin) and a set of rays and lines (seen as bidirectional rays). The polyhedron is then seen as the intersection of the cone with an hyperplane. This technical choice does not lessen the generality of the method. As for polyhedra, a cone can be described with constraints or with generators. Turning the polyhedron into a cone allows several simplifications in the process. First, by adding a dimension, vertices and rays are treated in the same way. Another advantage is that the cone always contains the origin (Section 7.1.5) and we can settle for finding the positive solutions of a system, in which case simplex-based methods can be used.

7.2.1 Principle of Motzkin's Algorithm

Motzkin's algorithm can be used to compute the generating system of a polyhedron from its constraint system, as well as the inverse operation, by duality. We consider the first case.

The algorithm starts with a double, minimal description (constraints and generators) of a polyhedron that includes the polyhedron of which we want a double description. In practice, the universal polyhedron is used, whose a generating system is constituted by the origin vertex, an empty set of rays and a basis of the space as the set of lines. Initially, the set of constraints taken into account is empty.

Informally, each step of the algorithm consists in taking into account a new constraint $c = (aX \leq b)$[†3]. Then:

[†3] We present the case of an inequation, the case of equations is easily deduced.

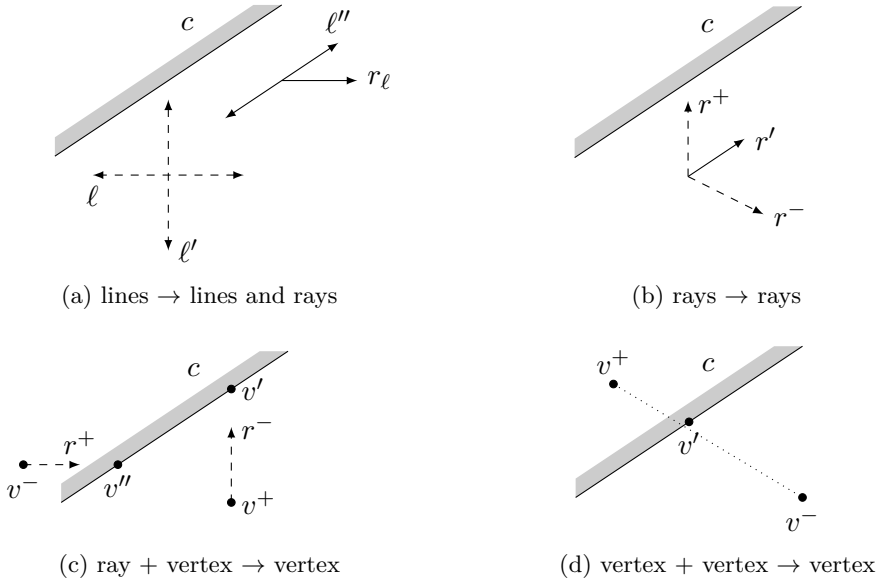
1. If all the lines of the current generating system satisfy the constraint, the set of lines is left unchanged. Otherwise, a line ℓ not satisfying the constraint (i.e., such as $a\ell \neq 0$, see Definition 7.10) is chosen: it is composed linearly with all the other lines ℓ' not satisfying the constraint, to create new lines ($\ell'' = (a\ell')\ell - (a\ell)\ell'$) that satisfy the constraint, and the vector ℓ or $-\ell$, depending on the sign of $a\ell$, becomes a new ray r_ℓ (Figure 7.5(a)).

2. For all pairs (r^-, r^+) of *adjacent* rays in the current generating system, such that r^- satisfies the constraint ($ar < 0$) and r^+ does not ($ar^+ > 0$), r^+ is replaced by $r^- = (ar^+)r^- - (ar^-)r^+$, a positive combination that saturates the constraint (Figure 7.5(b)).

3. For all pair (v^-, r^+) (respectively, (v^+, r^-)) formed by a vertex strictly satisfying (respectively, not satisfying) the constraint (i.e., $av^- < b$ and $av^+ > b$, respectively) and a ray that does not satisfy it (respectively, that strictly satisfies it), (v^-, r^+) (respectively, (v^+, r^-)) being *adjacent*, r^+ (respectively, v^+) is replaced by the vertex $v' = v^- + \frac{b-av^-}{ar^+} \cdot r^+$ (respectively, $v'' = v^+ + \frac{b-av^+}{ar^-} \cdot r^-$) that saturates the constraint (Figure 7.5(c)).

4. For all pair of vertices (v^-, v^+), adjacent and situated on either side of the constraint, v^+ is replaced by the vertex $v' = \lambda v^- + (1-\lambda)v^+$, with $\lambda = \frac{av^+-b}{av^+-av^-} \in]0, 1[$, that saturates the constraint (Figure 7.5(d)).

5. Other elements in the current generating system that do not satisfy the constraint are removed.



(a) lines → lines and rays

(b) rays → rays

(c) ray + vertex → vertex

(d) vertex + vertex → vertex

Figure 7.5: Steps of Motzkin's algorithm

An invariant of this algorithm is that at each step, the generating system is minimal, thanks to the fact that only *adjacent* generating elements are combined. Therefore, the resulting generating system is minimal. Constraints are processed one by one until all constraints in the initial system have been dealt with.

7.2.2 Optimizations

As mentioned above, Motzkin's algorithm is, in general, implemented on a representation of polyhedra based on cones. Unfortunately, determining adjacencies is expensive, and the main optimizations have eliminated these computations:

- Wilde [Wil93] and Le Verge [LV94] propose to combine all pairs of elements situated on either side of the constraint, without concern for adjacency. As the result of the combination is no longer necessarily extremal, its extremality is tested *a posteriori* with the following criterion:

  Let $n - \operatorname{codim} P$ be the dimension of a polyhedron. Then a vertex (respectively, a ray) that does not saturate at least $n - \operatorname{codim} P$ (respectively, $n - \operatorname{codim} P - 1$) inequations is not extremal.
- The extremality condition of Le Verge is only necessary, and so does not implies the minimality of the result. It has to be ensured by a complete analysis of saturations (see Proposition 7.13). The Parma Polyhedral Library [BRZH02] relies on the following necessary and sufficient extremality condition:

  Let $n - \operatorname{codim} P$ be the dimension of a polyhedron and $\dim P$ its codimension. Then a vertex (respectively, a ray) is extremal if and only if it saturates at least $n - \operatorname{codim} P - \dim P$ (respectively, $n - \operatorname{codim} P - \dim P - 1$) inequations.

Another optimization, used in the Linear library (Section 7.5.4), arises from the remark that, in practice, matrices involved in representations are often very sparse, and consists in using classic data structures for sparse matrices.

7.3 Operations on Polyhedra

An abstract interpretation of interpreted automata (see Section 6.3, page 61), based on the lattice of polyhedra, requires defining the following operations:

- Lattice operations:
  - Computation of the order relation (test for polyhedral inclusion).
  - Comparison to $\bot$, $\top$ (test for emptiness and fullness).
  - Lower bound, upper bound (intersection, convex hull).
- Effect of an action, forward and/or backward.
- Widening.

These operations rely sometimes on one representation, sometimes on the other, or even both of them, confirming the interest of having both representations.

7.3.1 Tests

We need to perform the following tests:

Emptiness Test

A polyhedron is empty if its constraints are not satisfiable. This test can easily be performed on the minimized generating system, since a polyhedron is empty if and only if its vertex set is empty.

Fullness Test

A polyhedron is equivalent to the universe polyhedron if its constraints are always satisfied, in other words if its minimized constraint system is empty.

Inclusion Test

Here, both representations are used, since

$$P \subseteq P' \iff \forall v \in V, A'v \leq B' \wedge \forall r \in R, A'r \leq 0 \wedge \forall \ell \in L, A'\ell = 0$$

where (V, R, L) is a generating system of P, and $A'X \leq B'$ is a constraint system of P' (see Definition 7.10).

7.3.2 Intersection

The intersection of two convex polyhedra is a convex polyhedron. The intersection satisfies the constraints of both polyhedra. Thus, trivially, the conjunction of constraint systems of both (or several) polyhedra is a constraint system of their intersection. This conjunction is not minimal in general and is often minimized immediately, to optimize future computations.

7.3.3 Convex Hull

The union of two convex polyhedra is not necessarily convex. The upper bound operation in the lattice of polyhedra is the *convex hull*, noted $\sqcup$, that maps two (or several) polyhedra to the smallest polyhedron containing them (Figure 7.6).

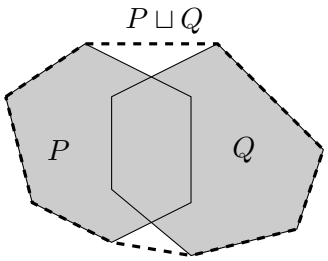


Figure 7.6: Convex hull of two polyhedra

This is the dual operation of intersection, and a generating system of $P_1 \sqcup P_2$ is formed by the union of vertices in P_1 and P_2, the union of their rays, and the union of their lines. The resulting generating system is not minimal in general.

7.3.4 Applying Actions, Projections and Affine Transformations

The action of an interpreted automaton, as defined in Section 5.3.1, page 52, is a relation between values of variables. Obviously, in all generality, it is impossible to define the effect of such an action on a polyhedron. Usually, it is simply defined in the case of *affine* relations, and roughly approximated in other cases, eliminating the involved variables (existential quantification).

Image by an Affine Relation

We only focus on the case of relations a in the form

$$a(X, X') \iff X' = FX + G$$

where F is an $n \times n$ square matrix, and G is a vector of dimension n[†4]. This case is interesting, from the point of view of the considered automata, since it corresponds to the actions defined by affine affectations to variables.

[†4]It would not be much more difficult to consider more general affine relations of the form $FX + F'X' \leq G$.

As in Section 5.3.4, we note: $a(P) = P[X := FX + G] = \{FX + G \,|\, X \in P\}$ and $a^{-1}(P) = P[X := FX + G]^{-1} = \{X \,|\, FX + G \in P\}$.

Then, if $AX \leq B$ and (S, R, D) are the two representations of P:

- $\big(\{Fv + G \,|\, v \in V\}, \{Fr \,|\, r \in R\}, \{F\ell \,|\, \ell \in L\}\big)$ is a generating system of $a(P)$.
- $AFX \leq B - AG$ is a constraint system of $a^{-1}(P)$.

Existential Quantification

It is often useful to eliminate certain variables (i.e., to lose all information about them), for instance to overapproximate the effect of a non-affine affectation to these variables. This operation corresponds to an existential quantification on a polyhedron: $\exists x, P$. This can be done, using the generating system of P, by simply adjoining it a line which is the base vector corresponding to x, or, using the constraint system of P, by applying a Fourier-Motzkin elimination [Sch86].

7.3.5 Widening

The choice of a widening operator heuristic is not an absolute choice but a compromise, that requires experimental validation. The standard operator has been proposed by [CH78]. The initial idea, considering two polyhedra P and Q, with $P \subseteq Q$, is to build the constraint system of $P \nabla Q$ by preserving only the inequalities $c = (aX \leq b)$ of P verified by every point X in Q (see Figure 7.7).

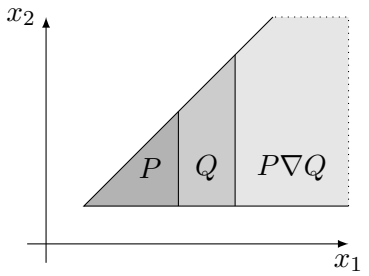


Figure 7.7: Standard widening for convex polyhedra

We should remember that widening is mainly used for fixed-point computation (Section 6.2.3), to "guess" the limit of an infinite sequence based on its first terms. Here, the idea is that a transformed inequality from P to Q (by a translation, a rotation…) may be transformed an infinite number of times and so, is removed. Polyhedra P and Q are included in $P \nabla Q$. As inequalities are removed from a finite system of inequalities, the algorithm stops in a finite number of iterations.

The problem with this definition is that it depends on the constraint system chosen for P: if this system is not canonical — and we stated in Section 7.1.2 that it happens when P satisfies equations — widening can unnecessarily lose information.

For instance, let us consider the following polyhedra (Figure 7.8):

- $P = \{x_1 = 0, 0 \leq x_2 \leq 1\}$.
- $Q = \{0 \leq x_1 \leq 1, 0 \leq x_2 \leq x_1 + 1\}$.

If the widening operator is directly applied, the result is a quadrant (Figure 7.8(a)):

$$P \nabla Q = \{0 \leq x_1, 0 \leq x_2\}.$$

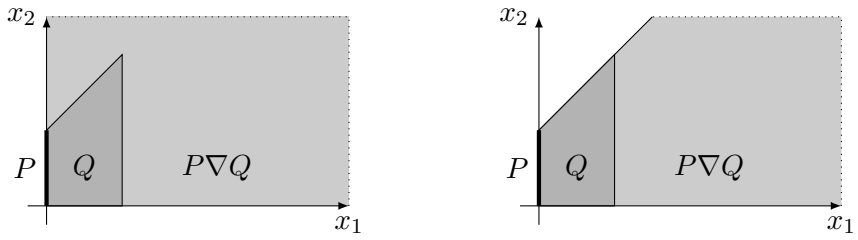


Figure 7.8: Widening a polyhedron satisfying an equation

However, if P is described by $P = \{x_1 = 0, 0 \leq x_2 \leq x_1 + 1\}$, we get (Figure 7.8(b)):

$$P\nabla Q = \{0 \leq x_1, 0 \leq x_2 \leq x_1 + 1\}.$$

To avoid this problem, the widening operator of [Hal79] uses the notion of mutually redundant constraints (see Definition 7.5): the constraint system of $P\nabla Q$ is formed by the constraints of Q that are mutually redundant, in P, with a constraint in P. In the previous example, the constraint $x_2 \leq x_1 + 1$ in Q is mutually redundant in P with the constraint $x_2 \leq 1$, and the widening gives the result shown in Figure 7.8(b), whatever the shape chosen for the constraints in P.

There were subsequent work [BHRZ05] to improve this widening operator, by distinguishing particular cases. But the standard operator, generally improved as follows, is the most commonly used.

7.3.6 Limited Widening

In [HPR97], a technique is proposed to improve the widening operator: when we know a set of constraints, $\mathcal{U}$, that is likely to be invariant in a widening state, we can choose to keep for the widened polyhedron the constraints in $\mathcal{U}$ that are satisfied by all points in both widening operands. As $\mathcal{U}$ is finite, the fundamental properties of widening are not changed. However, this optimization often immediately obtains the results that the decreasing sequence would provide, and also gives more accurate results.

7.4 Analysis Example

Let us consider the car example first introduced in Section 5.3.3, page 53, slightly simplified to show only numerical aspects. Each control point of the automaton is associated to a polyhedron (see Figure 7.9), as indicated in Section 6.3.

The abstract system of equations associated to this automaton is the following:

$$\begin{aligned}
P_{\text{init}} &= \top \\
P &= [d := 0][t := 0][v := 0]P_{\text{init}} \sqcup \\
&\quad [v := v + 1][d := d + 1](P \cap (v \leq 2 \wedge d \leq 9)) \sqcup \\
&\quad [t := t + 1][v := 0](P \cap (t \leq 3)) \\
P_{\text{wall}} &= P \cap (d \geq 10) \\
P_{\text{fast}} &= P \cap (v \geq 3) \\
P_{\text{stop}} &= P \cap (t \geq 4)
\end{aligned}$$

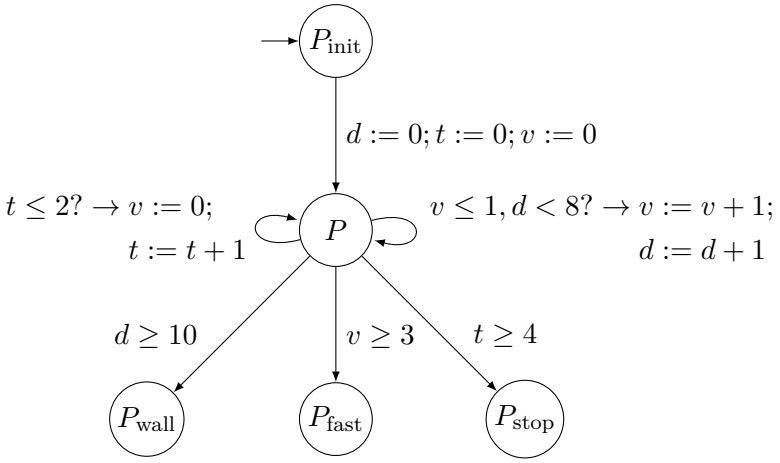


Figure 7.9: Car example with a polyhedron associated to each control point

and P is chosen as widening point, which will be limited by the set of distinguished constraints $\{v \leq 2, d \leq 8, t \leq 3\}$ (see Section 7.3.6). Indeed, the control point P is attached to is exited (definitively) as soon as one of these constraints is violated, so they are *a priori* likely to be invariant in this control point.

The iterative resolution (Section 6.3) is the following. We only give the successive values of P until convergence, then the final value of P will allow us to compute P_{wall}, P_{fast} and P_{stop}.

1. All polyhedra are initialized to the value $\bot$. So, the first iteration gives:

$$\begin{aligned} P^{(1)} &= [d := 0][t := 0][v := 0]\top \sqcup \bot \sqcup \bot \\ &= (d = 0 \wedge t = 0 \wedge v = 0). \end{aligned}$$

2. The second iteration produces:

$$\begin{aligned} P^{(2)} &= P^{(1)} \nabla \big([d := 0][t := 0][v := 0]\top \sqcup \\ &\qquad [v := v + 1][d := d + 1](P^{(1)} \cap (v \leq 2 \wedge d \leq 9)) \sqcup \\ &\qquad [t := t + 1][v := 0](P^{(1)} \cap (t \leq 3)) \big) \\ &= P^{(1)} \nabla \big((d = 0 \wedge t = 0 \wedge v = 0) \sqcup \\ &\qquad (d = 1 \wedge t = 0 \wedge v = 1) \sqcup \\ &\qquad (d = 0 \wedge t = 1 \wedge v = 0) \big) \\ &= P^{(1)} \nabla (0 \leq d \leq 1 \wedge 0 \leq t \leq 1 \wedge t + d \leq 1 \wedge v = d). \end{aligned}$$

   The standard widening would give $P^{(2)} = (v = d \wedge 0 \leq t \wedge 0 \leq d)$, but as the distinguished constraints are satisfied by both widening operands, the limited widening gives:

$$P^{(2)} = (v = d \wedge 0 \leq t \leq 3 \wedge 0 \leq d \leq 2).$$

3. The third iteration of P is:

$$\begin{aligned} P^{(3)} &= P^{(2)} \nabla \big((d = 0 \wedge t = 0 \wedge v = 0) \sqcup (v = d \wedge 0 \leq t \leq 3 \wedge 1 \leq d \leq 2) \sqcup \\ &\qquad (v = 0 \wedge 1 \leq t \leq 3 \wedge 0 \leq d \leq 2) \big) \\ &= P^{(2)} \nabla (0 \leq v \leq d \leq 2t + v \wedge t \leq 3 \wedge d \leq 2) \\ &= (0 \leq v \leq d \leq 2t + v \wedge t \leq 3 \wedge d \leq 2). \end{aligned}$$

4. The result for the fourth iteration is:

$$\begin{aligned} P^{(4)} &= P^{(3)} \nabla (0 \leq v \leq d \leq 2t + v \wedge d - 2 \leq v \leq 2 \wedge t \leq 3 \wedge d \leq 2) \\ &= (0 \leq v \leq d \leq 2t + v \wedge v \leq 2 \wedge t \leq 3). \end{aligned}$$

5. Finally, we find

$$\begin{aligned} P^{(5)} &= P^{(4)} \nabla P^{(4)} \\ &= P^{(4)}. \end{aligned}$$

   The computation has converged towards a fixed point, that cannot be improved by a decreasing sequence.

Now, we can compute the remaining polyhedra:

$$\begin{aligned} P_{\text{wall}} &= P^{(4)} \cap (d \geq 10) \\ &= \bot \\ P_{\text{fast}} &= P^{(4)} \cap (v \geq 3) \\ &= (v = 3 \wedge 3 \leq d \leq 2t + 3 \wedge t \leq 3) \\ P_{\text{stop}} &= P^{(4)} \cap (t \geq 4) \\ &= (t = 4 \wedge v = 0 \wedge 0 \leq d \leq 8) \end{aligned}$$

and the result $P_{\text{wall}} = \bot$ shows that in the model, the car stops before the wall!

7.5 Existing Libraries

Several libraries, providing most of the polyhedral operators, are available; we present the main ones in this section. All libraries mentioned below operate with rational numbers (with limited or arbitrary precision), for the sake of computational accuracy.

7.5.1 PolyLib

Initiated at IRISA [Wil93] in the early 1990s, the development of the PolyLib[†5] has benefited from the cooperation of several teams, working on program parallelization and the design of systolic architectures.

The provided operations are the usual polyhedral operations, along with the possibility to manipulate explicit finite unions of polyhedra which, unlike convex hulls, do not lose information. The PolyLib also offers sophisticated algorithms (for instance [NR00]) allowing to study integer solutions of linear systems.

The PolyLib is implemented in C.

7.5.2 NewPolka

The NBAC tool [Jea00] uses the NewPolka library[†6], that relies on a preliminary version of the PolyLib and on optimizations proposed in [FP96]. NewPolka is now integrated in the APRON library[†7], which offers access to several abstract domains, and also a higher-level interface. The main differences with PolyLib are the following:

[†5] See `http://www.irisa.fr/polylib/`.

[†6] See `http://pop-art.inrialpes.fr/~bjeannet/newpolka/index.html`.

[†7] See `http://apron.cri.ensmp.fr/library/`.

Representation of coefficients Computing with rational numbers, with reduction to a same denominator of constraint coefficients and vector components (reduction whose impact on algorithm efficiency is identified by [Avi98]) implies an increase of coefficient sizes, that, as soon as the dimension increases, causes arithmetic overflows. To solve this problem, NewPolka allows choosing between three integer representations: 32 bits, 64 bits and multiprecision integers using GMP[†8].

Lazy evaluation The library keeps in general only one representation of the polyhedron. The conversion towards the other one as well as the minimization are delayed and performed only when necessary. When both representations are present, then they are both in a minimal form.

Sorting constraints and generators Matrix lines are sorted (lazily) by lexicographic order for two reasons. In general, this sorting leads to an acceleration into the conversion algorithm. And, more specifically, it becomes easier to eliminate identical constraints, for instance in an intersection.

Strict inequalities NewPolka allows to take into account strict inequalities, by adding an extra dimension [HPR97].

7.5.3 PPL

The University of Parma has also developed a polyhedral library, the *Parma Polyhedral Library* (or PPL) that is available on its website[†9]. PPL is quite close to the library realized for NBAC. The main difference is the programming language, C++ instead of C. Other differences derive more or less directly from the choice of C++ as programming language.

PPL is fully dynamic, no maximal size is specified during initiation.

The treatment of strict inequalities [BRZH02] is different from that of NewPolka. The PPL also provides alternative widening operators [BHRZ05].

7.5.4 PIPS/Linear

Linear is the linear library used in the PIPS compilation framework[†10] [IJT91] developed at MINES ParisTech. This library uses sparse data structures to store the vectors, constraints and constraint systems, and dense data structures for matrices.

The C data type for coefficients can be set to 32 or 64-bit integers, but is incompatible with multiprecision numbers. However, intermediate computations may generate polyhedra with huge coefficients, leading to arithmetic overflows even with 64-bit integers, because constraint constants are transformed into coefficients by the convex hull operator. Eventually, when an overflow occurs, some constraints may be dropped, and the resulting invariant is less accurate than it should.

To address this problem, we added GMP support to some of the Linear polyhedral operators (Section 10.3.2). This seemingly purely pratical implementation decision allows for a drastical simplification of the polyhedral algorithms because overflow exceptions no longer have to be handled.

[†8] See `http://gmplib.org/`.

[†9] See `http://www.cs.unipr.it/ppl/`.

[†10] See `http://pips4u.org/`.

7.5.5 isl

The *Integer Set Library*[†11] (or isl) [Ver10] is a thread-safe C library for manipulating sets and relations of integer points bounded by affine constraints:

$$\begin{aligned} S(s) &= \left\{ x \in \mathbb{Z}^d \,\middle|\, \exists z \in \mathbb{Z}^e : Ax + Bs + Dz \geq c \right\} \\ R(s) &= \left\{ x_1 \to x_2 \in \mathbb{Z}^{d_1} \times \mathbb{Z}^{d_2} \,\middle|\, \exists z \in \mathbb{Z}^e : A_1 x_1 + A_2 x_2 + Bs + Dz \geq c \right\}. \end{aligned} \tag{7.1}$$

The descriptions of the sets and relations may involve both parameters (s in Equation (7.1)) and existentially quantified variables (z in Equation (7.1)). All computations are performed in exact integer arithmetic using GMP. The isl library offers functionalities that are similar to that offered by the Omega+ library[†12], but the underlying algorithms are in most cases completely different.

[†11] See `http://isl.gforge.inria.fr/`.

[†12] See `http://www.cs.umd.edu/projects/omega/`.

Chapter 8

ALICe: A Framework to Improve Affine Loop Invariant Computation

Comme cela a déjà été souligné au chapitre 6, un point crucial de l'analyse de programme est le calcul des invariants de boucle. Il est nécessaire d'avoir des invariants précis pour prouver des propriétés sur un programme, mais de tels invariants sont difficiles à calculer. Des recherches approfondies ont été effectuées mais, à notre connaissance, aucun benchmark n'a été développé jusqu'à présent pour comparer les algorithmes et les outils.

C'est pourquoi nous avons développé ALICe, un ensemble d'outils pour comparer des techniques de calcul automatique d'invariants affines de boucle. ALICe s'accompagne d'un benchmark, que nous avons construit à partir de 102 cas de test trouvés dans la littérature sur les invariants de boucles, et s'interface avec trois programmes d'analyse, reposant sur différentes techniques : Aspic, iscc et PIPS (voir la section 8.2.5). Des outils de conversion sont fournis pour manipuler les différents formats utilisés par ces programmes.

* * *

As already underlined in Chapter 6, a crucial point in program analysis is the computation of loop invariants. Accurate invariants are required to prove properties on a program but they are difficult to compute. Extensive research has been carried out, but, to the best of our knowledge, no benchmark has ever been developed to compare algorithms and tools.

This is why we have developed ALICe, a toolset to compare automatic computation techniques suited for affine loop scalar invariants. It comes with a benchmark, that we built using 102 test cases found in the loop invariant bibliography, and interfaces three analysis programs, that rely on different techniques: Aspic, iscc and PIPS (see Section 8.2.5). Conversion tools are provided with ALICe to handle the various formats used by these programs.

8.1 Invariant Computation

The standard state-based model checking problem is to characterize the set of all reachable states of a transition system modeling some program. This information is generally used to check safety properties on the system, ensuring that "bad" configurations cannot be reached. The accuracy of computed invariants is very important, as it plays an essential role in the success of the program analysis.

Dealing with potentially infinite-state models requires to overapproximate invariants into a mathematical model (or *abstract domain*) such that invariant representation is finite and that allows to make the required computations. Many such domains exist in the literature, but we focus

on the domain of *affine* invariants (Chapter 7), first introduced by N. Halbwachs [CH78, Hal79], as it offers a good compromise between invariant accuracy and computational complexity.

Most of the usual analysis techniques consist in starting from a set of supposed predicates about a particular control position in the transition system, and then propagating it to other positions by evaluating the effect of each transition on the predicates. This is pretty straightforward, except in the case of loops, which requires a special treatment. This lead to intensive research, with many approaches based either on abstract interpretation [Jea00, Gon07, Mer05], that is, using widening operations, or on direct computation [ACI10]. The case of concurrent loops, i.e. when there are different possible transition cycles on the same control point, is particularly challenging.

As of today, several tools for linear relation analysis use a vast array of algorithms and heuristics to handle loops, aiming to maximum accuracy. We propose a toolset that compares these tools on a common set of small-scale, previously published test cases and to test the sensibility of these tools to different encoding schemes.

8.2 The ALICe Framework

The ALICe framework project aims to provide tools and a standardized set of test cases to compare as fairly as possible different polyhedral analysis techniques and programs. ALICe is a free software distributed under GPLv3, available at `http://alice.cri.mines-paristech.fr/`.

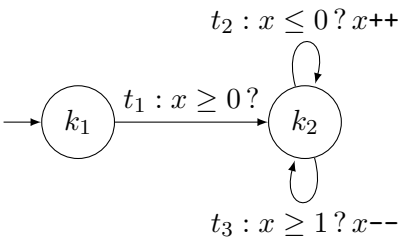
There are several motivations behind the ALICe project. First, we want to use it as a tool to compare polyhedral invariant computation tools on a common ground, using a set of published test cases issued from various sources, instead of evaluating their performances on *ad hoc* examples only, in the spirit of the TPTP library [Sut09] and the CASC competition [SS06, PSS02]. ALICe also gives the opportunity to evaluate the benefits of model-to-model restructurations prior to analysis, and/or the sensitivity to encoding of the tools (Chapter 9).

8.2.1 Program Model

Test cases in ALICe — also called *models* — are interpreted automata as described in Section 5.3: they are transition systems constituted by a set K of control points (nodes), connected by guarded commands acting on integer variables (edges), and by a set of initial states Q_{init}. Each model also comes with a set of *error states* Q_{err}.

Definition 8.1 (Correctness). A model m is *correct* if no error state is reachable from its initial states.

All models provided with ALICe are correct. An example of (correct) model is shown in Figure 8.1.



Initial states: $Q_{\text{init}} = \{(k_1, x) \mid x \in \text{Val}\}$

Error states: $Q_{\text{err}} = \{(k_2, x) \mid x < 0\}$

Figure 8.1: An example of model

Challenging a Tool

The purpose of error states is to challenge analysis tools. A tool successfully analyses a model if and only if the computed invariant, an overapproximation of the reachable states, does not intersect with the set of error states Q_{err}.

More precisely, it is not possible in general to compute exactly the set of accessible states $\text{Acc}(Q_{\text{init}})$. Instead, analysis tools tested by ALICe compute supersets Q' of $\text{Acc}(Q_{\text{init}})$: if a given Q' does not intersect with Q_{err}, then the tool that generated this Q' has verified that states in Q_{err} are unreachable (Figure 8.2), thus the test case is considered a success for that tool. However, if the intersection $Q' \cap Q_{\text{err}}$ is not empty, the tool user cannot conclude whether the property is violated or if the overapproximation is too inaccurate: this corresponds to a failure for the tool.

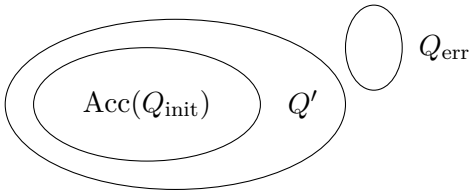


Figure 8.2: Model checking

Note that we could use *backward analysis* (see Section 5.3.4) instead: starting from error states Q_{err}, computing iteratively the set of coaccessible states and testing the intersection with Q_{init}. But, as all analysis tools used within ALICe rely on forward analysis (see Section 8.2.5), this is not explored further.

8.2.2 Test Cases

The benchmark suite itself consists in 102 previously published test cases, including work by L. Gonnord [Gon07, ADFG10], S. Gulwani [CCG+08, Gul09, GJK09, GMC08, GZ10], N. Halbwachs [Hal93, HPR97, Hal10, Hal79], B. Jeannet [Jea00] *et al.* The comprehensive list of model sources can be found in the bibliography [Bar05, Lee02, BHMR07, BGP97, CCG+08, CS01, CS02, CPR06, Cou05, GR06, GR07, GHK+06, Hen11, HJMS02, KEW+85, Mer05, PR04, PC07], as well as on the ALICe website. They come mostly from works on loop invariant computation, loop bound analysis, loop termination and, to a lesser extent, protocol verification. The list of models is given in Table 8.1.

Test cases are usually relatively small, with typically 1 to 10 states and variables, and 2 to 15 transitions. Histograms showing distributions of test cases according to their sizes are shown in Figure 8.3.

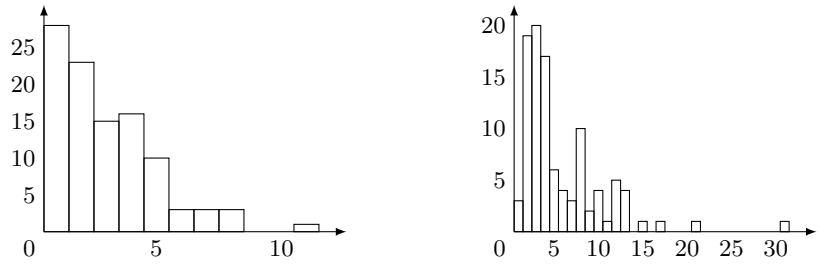
These test cases come in many forms and had to be encoded to a common format, described in Section 8.2.4. The encoding was either done manually, from the model description, or automatically from C code using the C2fsm utility developed by P. Feautrier [Fea10, FG10].

8.2.3 Supported Tools

When dealing with a state transition relation, a transitive closure is useful to compute invariants [BGP97, ACI10], to check loop termination or derive loop complexities [GZ10], to build dependency constraints or to move convex array regions from a control point to another [Kha13]. When dealing with a dependence relation, a transitive closure is useful to optimize or synthesize code [Won01, VCB11, BKK13].

aaron2	gulwani1	microsoftex6
ackerman	gulwani1_alt	microsoftex7
ax	gulwani2	multcounters1
bakery	gulwani2_alt	multcounters2
bardin	gulwani3	multcounters3
berkeley	gulwani4	multcounters4
car	halbwachs1	nd_loop
car_simple	halbwachs2	ndecr
counterex1	halbwachs3	nested
counters1	halbwachs4	perfect
counters2	halbwachs5	popeea
counters3	halbwachs6	random1d
counters4	halbwachs7	random2d
cousot9	halbwachs8	realbubble
determinant	halbwachs9	realheapsort
disj	henry	realheapsort_step1
disjbnd1	insertsort	realheapsort_step2
disjbnd2	interleaving1	realselect
easy1	interleaving2	realshellsort
easy2	interleaving3	relation1
exmini	interleaving4	rsd
forward	jeannet1	sipmabubble
gasburner	jeannet2	slam
gasburner_alt	jeannet3	slam_bad
gcd	jeannet4	speedometer
gonnord1	jeannet5	subway
gonnord2	jeannet6	synergy_bad
gonnord3	leeyannakis_bad	terminate
gonnord4	loops	terminator
gonnord5	maccarthy91	ticket
gonnord6	metro	wcet1
gopan_reps	microsoftex2	wcet2
gopan_reps_alt1	microsoftex4	while2
gopan_reps_alt2	microsoftex5	wise

Table 8.1: List of models currently provided with ALICe



(a) Number of control states (b) Number of transitions

Figure 8.3: Distribution of test cases

It is difficult to compare the algorithms and heuristics used by different tools designed for one of these goals because they require different inputs and produce incompatible outputs. Also, the encoding of the input often impacts the analysis.

As of now, ALICe is interfaced with three invariant computation tools: Aspic, iscc and PIPS.

- Aspic[†1] [Gon13], a polyhedral invariant generator developed by L. Gonnord. Aspic relies on classic linear relation analysis, improved with *abstract accelerations*, identifying classes of loops in the abstract polyhedral domain whose effect can be computed directly instead of using widening operations, thus granting better accuracy.
- iscc is an interactive interface to the barvinok counting library[†2] [VSB+07, VWBC05]. barvinok is a library for counting the number of integer points in parametric and non-parametric polytopes based on isl (Section 7.5.5) and PolyLib (Section 7.5.1). For a parametric polyhedron, the number of points is represented by either a piecewise quasi-polynomial or an Ehrhart series.
- PIPS[†3] [Cen15], an interprocedural source-to-source compiler framework for C and Fortran programs, initiated at MINES ParisTech, that relies on a polyhedral abstraction of program behavior. Unlike other tools, PIPS performs a two-step analysis. First, the program is abstracted: each program command instruction is associated to an affine transformer representing its underlying transfer function. This is a bottom-up procedure, starting from elementary instructions, then working on compound statements and up to function definitions. Second, polyhedral invariants are propagated along instructions, using transformers previously computed. PIPS relies on the Linear library (Section 7.5.4).

We do not consider the Omega+ [PRK+14] library, as it appears to be superseded by isl [Ver10]. It would be very interesting to use more tools, such as FASTer [Lab06, BFL04], NBAC [Jea10, Jea03] or PAGAI [HMM12a, HMM12b], but adding support is a time-consuming task, as explained below, and we have not been able to do it up to now.

8.2.4 Heterogeneity of Tools

ALICe test cases are written in the fsm format. This is a simple language that directly represents models, originally used by FAST [Lab06] and then by Aspic. An example of fsm program for the model in Figure 8.1 is given in Listing 8.1.

As we wish to analyze these test cases with different tools and compare results, we have to convert input and output formats. Basically, each tool uses different input and output formats:

- Aspic uses the fsm format as input and fsm expressions as output.
- iscc uses a custom format to describe both the input model as a relation on states with conditions on variables, and the output invariant, given as a map from states to polyhedral domains. An iscc relation corresponding to the model in Listing 8.1 is implemented in Listing 8.3, page 92.
- PIPS processes a structured C program according to a script written in a custom scripting language, called tpips, while resulting invariants are given as inserted before statements in the output C code. An example of PIPS output is given is Listing 10.3, page 103.

To check whether an analyzer works successfully on a test case, we follow these four steps:

1. If necessary, convert the model (originally in fsm) into the analyzer's input format;

[†1] See `http://laure.gonnord.org/pro/aspic/aspic.html`.

[†2] See `http://barvinok.gforge.inria.fr/`.

[†3] See `http://pips4u.org/`.

```
model m {
    var x;
    states k1, k2;
    transition t1 {
        from := k1;
        to := k2;
        guard := x >= 0;
        action := ;
    }
    transition t2 {
        from := k2;
        to := k2;
        guard := x <= 0;
        action := x' = x + 1;
    }
    transition t3 {
        from := k2;
        to := k2;
        guard := x >= 1;
        action := x' = x - 1;
    }
}
strategy s {
    Region init := {state = k1};
    Region bad := {x < 0};
}
```

Listing 8.1: Source code in fsm format

2. Run the analyzer and get the computed model invariant;
3. If necessary, convert the model invariant into isl format;
4. Use iscc to check whether the intersection of the model error region and the invariant computed by the analyzer is empty, i.e. whether the analyzer is able to solve the test case.

Those steps are illustrated in Figure 8.4.

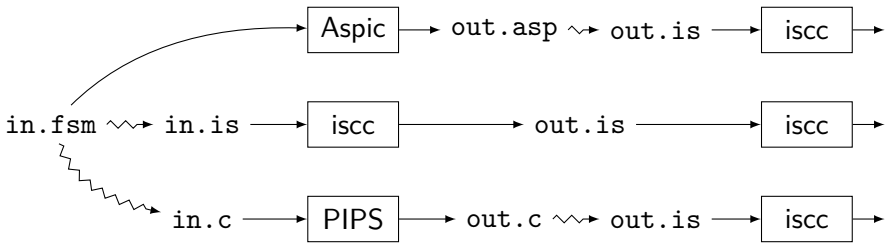


Figure 8.4: Analysis steps within ALICe

8.2.5 Translation Tools

Implementing these steps required to develop several translation programs, from and to Aspic, isl and PIPS formats. In particular, the tools fsm2c and fsm2iscc, presented below, respectively

convert a fsm model into C code or isl relation. There is also an export tool for fsm in dot format that allows visualization, fsm2dot. They are available within ALICe.

Conversely, it is possible to use ALICe to analyze simple models written in C by first translating them into fsm automata, using the C2fsm utility.

fsm2c

fsm2c is a conversion tool to generate C code from a fsm model. It is written in Python and distributed with ALICe. fsm2c works as follows:

1. The fsm code is parsed, using the PLY parsing library, and a finite state machine structure is generated.
2. A regular expression describing the possible paths in the automaton structure (regardless of transition guards) is generated, each character corresponding to a command. This regular expression represents the structure of the produced C code: unions are translated into conditional branches (`if (rand()) { ... }`), stars into loop statements (`while (rand()) { ... }`). Thus, it is important to generate a simple regular expression to avoid producing an unnecessarily complicated C code. This is done using Brzozowski's algorithm [Brz64], followed by a simplification step including the elimination of the ε-terms (the partial or final nature of an execution path is not relevant for invariant generation).
3. The C code is generated iteratively from the regular expression, as stated above. Each command $g\,?\,a$ is translated into a conditional statement with condition g and body executing the action in a. Temporary variables are introduced if needed, to handle concurrent modifications of variables in a, for instance in swaps.

Variables are renamed, to avoid any clash with C keywords. Dummy string instructions are also injected, to keep track of the variable names and control points of the original model in the produced C code. This allows to retrieve invariants at the model level when the C code is analyzed.

The C code produced from the fsm model in Listing 8.1 is shown in Listing 8.2.

fsm2iscc

fsm2iscc is a much simpler program. An iscc relation is directly produced from the fsm model, with no need for restructuring, renaming or control-state tracking. The iscc code produced from the fsm model in Listing 8.1 is shown in Listing 8.3.

8.3 Results for the Raw Test Cases

In this section, we present experimental results obtained with ALICe v 1.0. Benchmarks were run on a computer with a quad-core Intel i7-2600 processor clocked at 3.40 GHz with 16 GB of memory, using the following versions of analyzers (latest versions at the time of writing):

- Aspic version 3.3.
- iscc from barvinok version 0.37.
- PIPS revision 22 241 (September 2014).

The test suite was run 5 times and the average 3 runs were retained for each test case. Results obtained by these three tools are displayed in Table 8.2, along with the corresponding execution times. Aspic stands out as the winner in this comparison, with iscc coming second and PIPS,

which is not primarily targeted at invariant analysis, being placed last, both in terms of success rate and of execution time.

	Aspic	**iscc**	**PIPS**
Successes	75	63	43
Time (s.)	10.9	35.5	43.3

Table 8.2: Benchmark results

This ranking is not a total order because no tool is *strictly better* than another: for each tool, there is at least one model that is successfully analyzed only by this tool, as shown in Figure 8.5; and it appears in Table 8.3 that there is no clear trend as for the *quality* of generated invariants, in terms of invariant inclusion.

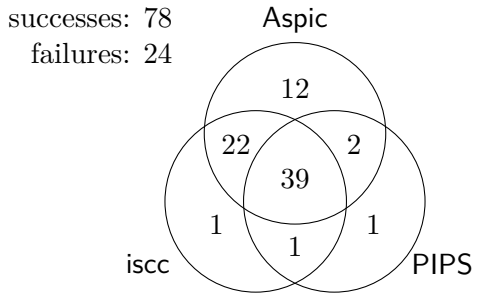


Figure 8.5: Venn diagram of successes for each tool

⊄	**Aspic**	**iscc**	**PIPS**
Aspic	–	21	23
iscc	49	–	54
PIPS	32	23	–

Table 8.3: Invariant inclusions, in terms of sets

A closer analysis shows that iscc performs comparatively well on test cases encoded with *concurrent loops* (several loops on a single control point, similar to what is shown Figure 9.3, page 98), unlike PIPS whose raw results appear to be the poorest. On the other hand, iscc can be quite slow on test cases that display a large, intricate control structure. Finally, despite its successes, Aspic has greater difficulty to deal with transitions featuring complex formulas, that it is not able to accelerate. This leads us to wonder if the tools are sensitive to the encoding, since a problem can be presented under many guises. This is the topic of Chapter 9.

PIPS comparative slowness is primarily due to the fact that it uses C as input format, which is much more universal and thus, difficult to handle, than Aspic or iscc specific formats. This issue is revisited in Section 10.2.1.

```c
#include <stdlib.h>
#include <stdbool.h>

#define {alice__abort}() {{while (1) {{}}}}
#define {alice__fail}() exit(1)
#define {alice__assume}(e) if (!(e)) {{{abort}();}}
#define {alice__rand}() (rand() - rand())

void model(main) {
    int var_0;
    "@alice variables: x";
    if (var_0 >= 0) {
        "@alice transition t1";
        "@alice state k1";
        while (alice__rand()) {
            if (alice__rand()) {
                "@alice transition t2";
                "@alice state k2";
                if (var_0 <= 0) {
                    var_0 = var_0 + 1;
                }
                else {
                    alice__abort;
                }
                "@alice state k2";
            }
            else {
                "@alice transition t3";
                "@alice state k2";
                if (var_0 >= 1) {
                    var_0 = var_0 - 1;
                }
                else {
                    alice__abort;
                }
                "@alice state k2";
            }
        }
        "@alice state k2";
    }
    else {
        alice__abort;
    }
}
```

Listing 8.2: C code produced by fsm2c for the example of Listing 8.1

```
model := {
    k1[x] -> k2[x]: x >= 0;
    k2[x] -> k2[x + 1]: x <= 0;
    k2[x] -> k2[x - 1]: x >= 1;
};
model + model^+;
```

Listing 8.3: iscc code produced by fsm2iscc for the example of Listing 8.1

Chapter 9

Model-to-Model Restructuring Transformations: Sensitivity to Encoding

On mentionne au chapitre 8 la possibilité d'utiliser ALICe pour évaluer l'efficacité et la pertinence des restructurations de modèles. La motivation initiale était d'améliorer des résultats obtenus avec PIPS en jouant sur la structure de contrôle des modèles.

À des fins d'analyse, un modèle que l'on ne parvient pas à vérifier peut être restructuré en un autre modèle, que l'on espère plus facile à analyser. Dans ALICe, nous avons implémenté deux restructurations sur les états des modèles, pour tester les outils d'analyse sur un plus grand échantillon de modèles et sur des schémas particuliers de modèles, et explorer l'impact de l'encodage des modèles. Les restructurations de modèles sont effectuées au tout début de l'exécution d'ALICe, lors d'une étape préliminaire additionnelle, représentée sur la figure 9.1.

La sûreté de ces deux restructurations a été prouvée en Coq, en utilisant un raisonnement par équivalence de traces.

* * *

We mention in Chapter 8 the possibility to use ALICe to test the efficiency and relevance of model-to-model restructuring transformations. An initial motivation was to improve results in PIPS by playing on the model control structure.

For analysis purposes, a model that fails to be checked can be restructured into another one, hopefully easier to analyze. Within ALICe, we have implemented two restructuring transformations on model states, to test analysis tools on a wider range of models and on specific model schemes, and explore the impact of model encoding. Model restructuring transformations are performed at the very beginning of ALICe execution, as an additional, preliminary stage, as shown in Figure 9.1.

Both restructuring transformations were proved sound in Coq, using a trace-equivalence scheme.

Definition 9.1 (Model restructuring). A *model restructuring* is a sound abstraction on models, i.e. a function $\rho : \mathcal{M} \to \mathcal{M}$ that maps a model m_1 to a model m_2 such that: if m_2 is correct (i.e., its error region is not reachable), then m_1 is also correct:

$$\forall m_1, m_2 \in \mathcal{M}, m_2 = \rho(m_1) \wedge \text{correct}(m_2) \Longrightarrow \text{correct}(m_1). \tag{9.1}$$

Thus, given a model m_1 and a restructuring ρ, it is sufficient to prove the correctness of $m_2 = \rho(m_1)$ to deduce that m_1 is also correct. In addition, the restructuring ρ can be equivalent

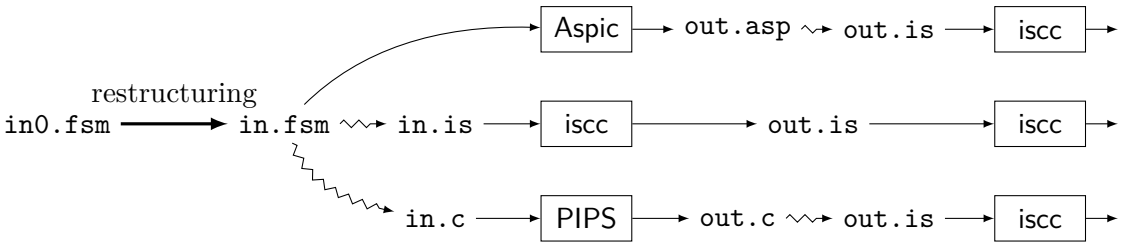


Figure 9.1: Analysis steps within ALICe, using restructuring transformations

(m_1 is correct if and only if m_2 is correct), although we are not interested in proving such properties in our toolchain. So, for instance, the restructuring that to any model associates the same trivial, inconsistent model is sound (but not very interesting).

Considering an arbitrary, possible state trace with transitions

$$\theta_1 = (k_0, v_0) \xrightarrow{t_1} (k_1, v_1) \xrightarrow{t_2} (k_2, v_2) \xrightarrow{t_3} \cdots$$

in the original model m_1, we show that for any corresponding trace θ_2 in the transformed model m_2 (corresponding to the same behavior, once the model has been transformed), then:

$$(\text{for all state } q_2 \text{ in } \theta_2, q_2 \notin Q_{\text{err}2}) \Longrightarrow (\text{for all state } q_1 \text{ in } \theta_1, q_1 \notin Q_{\text{err}1}),$$

thus ensuring Equation (9.1).

9.1 Control-Point Splitting Heuristic

The two main sources of imprecision that appear in Linear Relation Analysis are

- The computation of branches, performed with the convex union of invariants or transformers of each path (Sections 6.2.2 and 7.3.3).
- The computation of loops, whether using widening (Sections 6.2.3 and 7.3.5) or alternative techniques, as in PIPS (see Section 10.1.1).

These imprecisions are accumulated if there are structures in the automaton with two or more loops involved in the same control point. That is the case, for instance, of control k_2 in the model of Figure 8.1, page 84, that involves two self-loops t_2 and t_3.

If such a structure is met during analysis, two ideas are used to improve accuracy. The first one is to refine invariants or transformers involved in loops, so that both the transitive closure approximation and the convex union might be more precise. The second is to reduce the number of parallel loops. Both can be achieved through a control restructuring of the analyzed automaton.

The first restructuring we use is a heuristic to split control nodes that "contain" several cycles. The global idea is to get rid of such nodes, which are usually the most difficult to automatically analyze, by splitting them with respect to the guards of the transitions, and adjusting the initial and error regions accordingly, as shown in Figure 9.2. This heuristic was initially designed for PIPS and is presented in details in [Mai12].

9.1.1 Algorithm

The control-point splitting algorithm presented here is a restructuring aimed at achieving these two goals.

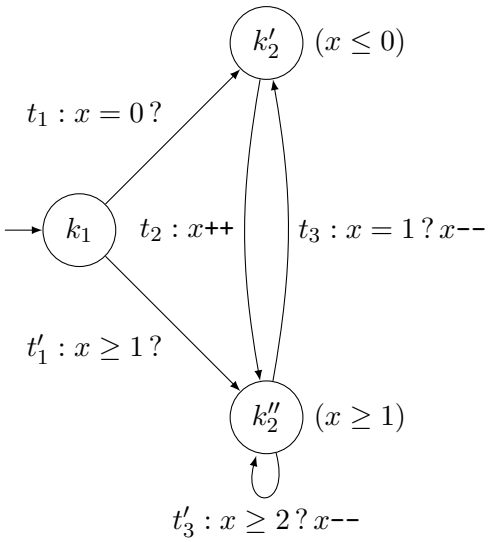


Figure 9.2: Model of Figure 8.1 transformed by the splitting heuristic

Let $m = (X, K, \text{Trans}, Q_{\text{init}}, Q_{\text{err}})$ be a model, $k \in K \setminus \{k_{\text{init}} \,|\, \exists v \in \text{Val}, (k_{\text{init}}, v) \in Q_{\text{init}}\}$ a non-initial control of m and $\mathcal{P} = P_1 \uplus \cdots \uplus P_m$ a partition of the domain of valuations Val. The model automaton $m_{\mathcal{P}/k}$ is obtained from m by performing the following steps:

1. Delete control k.
   Add fresh controls $k_{P_1}, \ldots, k_{P_m}$.

2. Delete each transition (k, g, a, k') that leaves k $(k' \neq k)$.
   For all $i \in [1, m]$, add the transition (k_{P_i}, g_i, a, k') with g_i such that:
   $$\forall v \in \text{Val}, g_i(v) \iff g(v) \wedge v \in P_i. \tag{9.2}$$

3. Delete each transition (k', g, a, k) that enters in k $(k' \neq k)$.
   For all $j \in [1, m]$, add the transition (k', g, a_j, k_{P_j}) with a_j such that:
   $$\forall v, v' \in \text{Val}, a_j(v, v') \iff a(v, v') \wedge v' \in P_j. \tag{9.3}$$

4. Delete each transition (k, g, a, k) that loops on k.
   For all $i, j \in [1, m]$, add the transition $(k_{P_i}, g_i, a_j, k_{P_j})$ with g_i and a_j such that:
   $$\forall v \in \text{Val}, \quad g_i(v) \iff g(v) \wedge v \in P_i. \tag{9.4}$$
   $$\forall v, v' \in \text{Val}, a_j(v, v') \iff a(v, v') \wedge v' \in P_j. \tag{9.5}$$

Controls that are not accessible nor coaccessible are not created, neither are the related transitions. Transitions whose guards are not satisfiable are not created either.

Interestingly, if guards g and actions a are polyhedral, and if every partition element P_i of $\mathcal{P}$ is a polyhedron, then guards and actions in the resulting transitions are also polyhedral.

9.1.2 Correctness Theorems

We consider a general automaton $m = (X, K, \text{Trans}, Q_{\text{init}}, Q_{\text{err}})$ and its image $m_{\mathcal{P}/k}$ obtained by splitting the control node $k \in K$ of m through the control-point splitting algorithm.

Theorem 9.2. For all $j \in [1, m]$, for all $v \in \text{Val}$, if $q = (k_{P_j}, v')$ is a reachable state of $m_{\mathcal{P}/k}$, then $v' \in P_j$.

Proof. As the initial states of $m_{\mathcal{P}/k}$ are the same as in m, k_{P_j} is not initial. As k_{P_j} is reachable, there exists a transition t of $m_{\mathcal{P}/k}$ that enters in k_{P_j}. Let $v, v' \in \mathrm{Val}$ the valuations before and after entering in k_{P_j} through t. According to the algorithm above, two cases are possible:

- Either $t = (k', g, a_j, k_{P_j})$, as described in Step 3.
- Either $t = (k_{P_i}, g_i, a_j, k_{P_j})$, as described in Step 4.

Then $a_j(v, v')$ holds and hence, according to Equations (9.3) and (9.5), $v' \in P_j$. □

In other words, we have a guarantee that in every control k_{P_i} of $m_{\mathcal{P}/k}$, the invariant given by P_i holds.

Given two controls k_1 in m, k_2 in $m_{\mathcal{P}/k}$, we introduce a "state equivalence" relation $\sim_{\text{St.}}$ between two states of m and $m_{\mathcal{P}/k}$, defined by: $(k_1, v) \sim_{\text{St.}} (k_2, v)$ if and only if

- Either $k_1 = k$ and $k_2 = k_{P_i}$ where $i \in [1, m]$ satisfies: $v \in P_i$.
- Either $k_1 = k_2 \neq k$.

We also define a relation $\sim_{\text{Tr.}}$ between traces of m and $m_{\mathcal{P}/k}$: $T_1 \sim_{\text{Tr.}} T_2$ if and only if states in T_1 and T_2 pairwise satisfy $\sim_{\text{St.}}$. Both relations $\sim_{\text{St.}}$ and $\sim_{\text{Tr.}}$ are bisimulations. They are one-to-one mappings, so notions of image and inverse image by $\sim_{\text{St.}}$ and $\sim_{\text{Tr.}}$ are defined.

Then, the following two theorems hold:

Theorem 9.3. For all trace T_1 of m, there exists a trace T_2 of $m_{\mathcal{P}/k}$ such that $T_1 \sim_{\text{Tr.}} T_2$.

Theorem 9.4. For all trace T_2 of $m_{\mathcal{P}/k}$, there exists a trace T_1 of m such that $T_1 \sim_{\text{Tr.}} T_2$.

These two theorems show that control-point splitting preserves reachable states:

Theorem 9.5. Let q_1 a state of m and q_2 a state in $m_{\mathcal{P}/k}$ such that $q_1 \sim_{\text{St.}} q_2$. q_1 is reachable in m if and only if q_2 is reachable in $m_{\mathcal{P}/k}$.

Equivalence results 9.3, 9.4 and 9.5 give correspondences between automata m and $m_{\mathcal{P}/k}$ in terms of traces and reachable states. Thus, safety and liveness properties on m can be translated into equivalent properties on $m_{\mathcal{P}/k}$ and conversely. This allows to use the automaton $m_{\mathcal{P}/k}$ instead of m to prove correctness of models, if it turns out to be easier.

9.1.3 Partition Choice

It is well known that the choice of a control structure affects result accuracy. Therefore, when dealing with a given model automaton m, an important question is to determine the control points that should be split as well as the partitions they should be split along, in order to make the analysis more accurate.

We have seen previously that most of the accuracy losses arise from the analysis of control points with parallel cycles. These control points are candidates to be split. Concerning the partition choices, a trade-off should be found between different criteria.

First, the automaton structure should be kept as simple as possible, in terms of control and, even more, of transition number. In the general case, partitioning a control k within m components not only adds m control states to the automaton, but also up to m^2 transitions between the newly created controls — from any control to any control —; the larger size and more complex structure will increase the analysis complexity. To avoid these issues, the number m of partition components must be bounded and the partition carefully chosen, so that some of the created states are not reachable or not co-reachable, or that some of the created transitions are not

satisfiable, as they will not be present in $m_{\mathcal{P}/k}$. If possible, the priority is to eliminate transitions involved in loops or in cycles, as explained above in Section 9.1.

Also, the resulting transition guards should be as precise as possible, especially the ones involved in loops or parallel paths, to limit accuracy losses due to approximations.

Choosing the partition can be done manually, considering the system behavior. Since we want to use automatic invariant generation techniques, we also propose below an automatic heuristic technique.

9.1.4 Guard-Based Control Partitioning

Let m be an affine transformer automaton. On every control k with "parallel" cycles, let $g_1, \ldots, g_p$ be the guards of transitions looping on k.

Let $G_i = \{v \mid g_i(v)\}$ and $\overline{G}_i = \mathrm{Val} \setminus G_i$. If G_i is a convex polyhedron, then $\overline{G}_i$ can be partitioned into a finite set of convex polyhedrons $\overline{G}_{i,1}, \ldots, \overline{G}_{i,j_i}$ [Cha84]. So $P_i = \{G_i, \overline{G}_{i,1}, \ldots, \overline{G}_{i,j_i}\}$ is a partition of Val. The partition taken on control point k is:

$$\mathcal{P}_k = P_1 \otimes \cdots \otimes P_p$$

where $\otimes$ is the "product partition" operation defined by:

$$\forall E_1, \ldots, E_n, \forall F_1, \ldots, F_m, \{E_1, \ldots, E_n\} \otimes \{F_1, \ldots, F_m\} = \{E_i \cap F_j \mid i \in [1, n], j \in [1, m]\}.$$

In a nutshell, controls are created to explicitly allow or disallow each transition. The key idea behind this heuristic is that most of "parallel" loops have at least partly disjuncts guards. In this case, there are seldom controls with many parallel loops in the resulting model $m_{\mathcal{P}/k}$, if at all, and the memory state for these controls is well known (Theorem 9.2). This assumption proved reasonable in most of our test cases. Despite this, the main drawback of this technique is the important number of created controls and transitions, which limits its applicability to transition systems with a small number of transitions per control node.

In Figure 9.2, we show the effect of this heuristic applied on the model of Figure 8.1, where the control point k_2 is split into two components with respect to the guards of transitions t_2 and t_3. The initial state set is unchanged. Proving that the new error state set $\{k_2', k_2'\} \times (x < 0)$ cannot be reached is easier than in the original model: it is true by construction for the control point k_2'', and can be easily deduced by looking at guards entering transitions for control point k_2'.

9.2 Reduction to a Unique Control Point

The other model restructuring reduces the set of controls to a unique control point ℓ, with all transitions turned into loops on ℓ, adding an extra boolean variable x_k for each original control point k in both guards and actions (assuming symbols ℓ and $\{x_k \mid k \in [1, \ldots, n]\}$ are not bound in the original model), to represent the corresponding control point of the original model. We ensure that, in every state of the system, exactly one x_k is set to 1. Formally, a transition t between control states k_i, k_j, with guard g and action a

$$t : k_i \xrightarrow{g \,?\, a} k_j$$

is turned into the transition

$$t' : \ell \xrightarrow{g \wedge x_i = 1 \wedge \bigwedge_{k \neq i} x_k = 0 \,?\, a \wedge x_j' = 1 \wedge \bigwedge_{k \neq j} x_k' = 0} \ell.$$

Again, the initial and error set states are adjusted accordingly.

A more direct approach is to encode control information on a unique integer variable instead of several boolean variables, by numbering the control points in the initial model. We did not adopt it because it introduces encoding issues. Indeed, some relations on control points might or might not be expressed in terms of affine constraints, depending on the control-point encoding. For instance, being in (k_1 or k'_2) in the model of Figure 8.1 cannot be expressed if the third control point k_2 is encoded with a value between those of k_1 and k'_2. This issue is avoided with our boolean variable scheme.

The resulting model of this restructuring applied on the initial model of Figure 8.1 is shown in Figure 9.3. The corresponding initial state set is: $Q_{\text{init}} = \{\ell\} \times \{b_1 = 1 \wedge b_2 = 0\}$. The error state set is: $Q_{\text{err}} = \{\ell\} \times \{b_1 = 0 \wedge b_2 = 1 \wedge x < 0\}$.

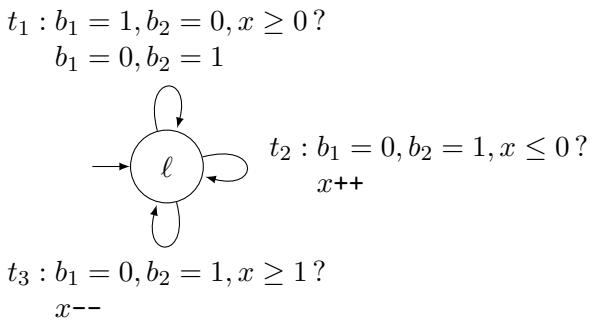


Figure 9.3: Model of Figure 8.1 reduced to a unique control point

This restructuring has three purposes. First, it stresses the tool with more difficult test cases. It also reduces bias factors related to encoding choices. Finally, if used before the control-point splitting heuristic, merging increases the effect of splitting by generating the type of control-point structure to which it applies.

9.3 Combining Restructuring Transformations

These model restructuring transformations can be used independently or together: first the model is reduced to a unique control point, then this control point is split, widening the scope of the splitting heuristic. Therefore, ALICe works with four versions of each initial model:

- The original version, with no restructuring (noted "direct" in the tables below);
- With all control points merged into a unique one, as described in Section 9.2 ("merged");
- Using the control-point splitting heuristic presented in Section 9.1 ("split");
- Combining both approaches ("merged-split").

9.4 Impact of Restructuring Transformations on Experimental Results

The results obtained using these restructuring schemes are displayed in Table 9.1.

We notice that the control state splitting heuristic leads to improved results for all tools, as shown by the comparison of Tables 9.1(a) and 9.1(c) with Tables 9.1(b) and 9.1(d), respectively. These results also confirm that iscc is significantly better in the treatment of concurrent loops, as previously noticed in Section 8.3: it outperforms the other tools on merged models

	Aspic	iscc	PIPS
Successes	75	63	43
Time (s.)	10.9	35.5	43.3

(a) Direct

	Aspic	iscc	PIPS
Successes	79	72	50
Time (s.)	12.8	43.0	60.6

(b) Split

	Aspic	iscc	PIPS
Successes	59	70	40
Time (s.)	16.7	26.2	43.5

(c) Merged

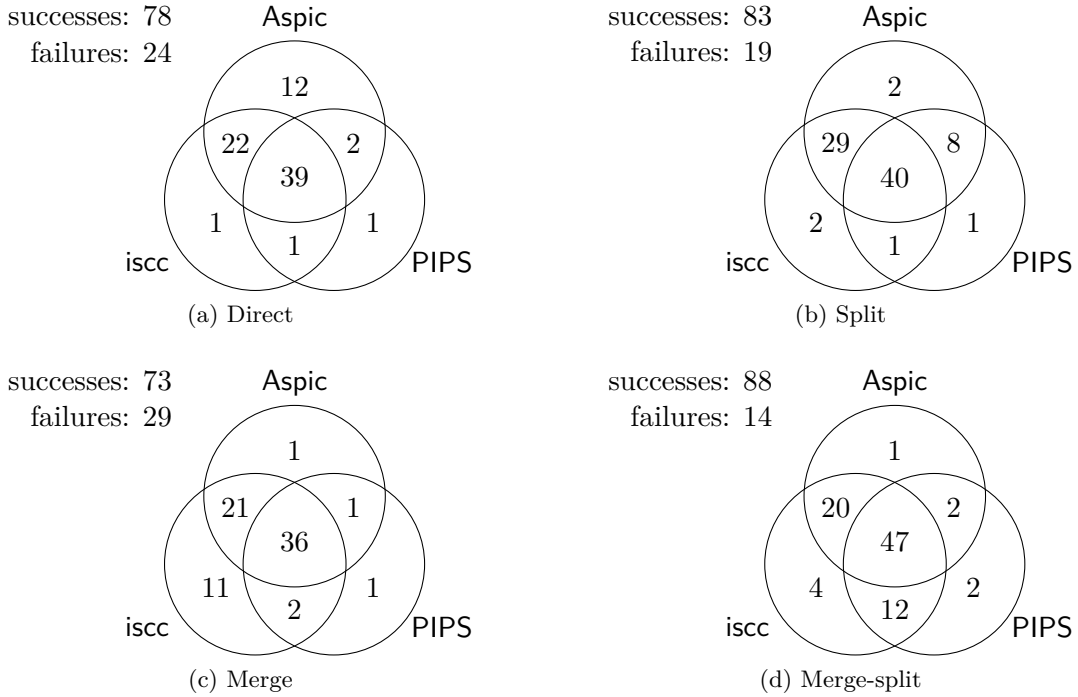
	Aspic	iscc	PIPS
Successes	70	83	63
Time (s.)	11.3	40.8	54.6

(d) Merged-split

Table 9.1: Benchmark results with different encodings

(Table 9.1(c)). Aspic has lower scores than iscc on split models, with or without merging (Tables 9.1(b) and 9.1(d)), because it cannot accelerate transitions that are generated by control node merging. PIPS is the worst performer in all cases, but we managed to increase its success rate by about 50 % (from Table 9.1(a) to Table 9.1(d)). The merge-splitting strategy gives the best results for iscc and PIPS.

Once again, detailed results are more contrasted. There is still no inclusion relation between successful test cases for different tools, regardless of the restructuring scheme (Figure 9.4). The same holds for invariant sharpness (Table 9.2).



(a) Direct (b) Split (c) Merge (d) Merge-split

Figure 9.4: Venn diagrams of successes for each tool

Globally, the merge-split restructuring leads to the best results with 88 out of 102 test cases correctly solved.

↝	Aspic	iscc	PIPS
Aspic	–	21	23
iscc	49	–	54
PIPS	32	23	–

(a) Direct

↝	Aspic	iscc	PIPS
Aspic	–	21	27
iscc	54	–	48
PIPS	51	36	–

(b) Split

↝	Aspic	iscc	PIPS
Aspic	–	7	8
iscc	68	–	54
PIPS	58	19	–

(c) Merge

↝	Aspic	iscc	PIPS
Aspic	–	21	22
iscc	69	–	58
PIPS	68	33	–

(d) Merge-split

Table 9.2: Invariant inclusions

Chapter 10

Computing Invariants with Transformers: Experimental Scalability and Accuracy

Dans l'analyse classique des relations linéaires, les invariants sont normalement calculés par propagation des préconditions le long des chemins de contrôle du programme, jusqu'à obtenir une précondition stable sur chaque point de contrôle, comme nous l'avons vu à la section 7.4. Pour éviter des propagations infinies, des opérateurs spéciaux d'approximation, comme l'élargissement, sont utilisés pour garantir la convergence en un nombre fini d'étapes.

Au lieu de cela, PIPS s'appuie sur une approche alternative, modulaire, en utilisant des transformeurs affines *[IJT91], c'est-à-dire des polyèdres représentant des fonctions de transfert. Dans la section 10.1, nous présentons comment PIPS calcule les transformeurs puis les invariants. La section 10.2 donne un aperçu de la modularité et de la précision de l'approche par transformeurs. Dans les sections 10.3 et 10.4, plusieurs améliorations sont apportées aux algorithmes de calcul des transformeurs afin d'augmenter la précision des invariants ; ces améliorations sont évaluées expérimentalement dans la section 10.5.*

* * *

In classic Linear Relation Analysis, invariants are usually computed by propagating preconditions along the control paths of a program until stable preconditions are obtained for each control node, as seen in Section 7.4. To avoid infinite propagations, special approximation operators, e.g. widening operators, are used to guarantee the convergence within a finite number of steps.

Instead, PIPS relies on a modular alternative approach using *affine transformers* [IJT91], i.e., polyhedra representing transfer functions. In Section 10.1, we present how transformers then invariants are generated in PIPS. Section 10.2 gives some insight on the modularity and accuracy of the transformer-based approach. In sections 10.3 and 10.4, several refinements in transformer computation algorithms are introduced to improve invariant accuracy; they are experimentally evaluated in Section 10.5.

10.1 Generation of Invariants with Transformers

Transformers and preconditions are shown in Listings 10.2 and 10.3 for the example `counter` used by Halbwachs *et al.* [HH12], as comments just above the related statement. Transformers for elementary statements contain many trivial equations of the form $x_{\text{init}} = x$ because usually few variables are modified. These equations are kept implicit and instead the modified variables

are listed as arguments. Finally, a `#init` suffix is used to distinguish the old value from the new value.

```
void foo(float x) {
    int n = 0;
    while (1) {
        if (x) {
            if (n<60)
                n++;
            else
                n = 0;
        }
    }
 }
```

Listing 10.1: Statements for `counter`

```
// T() {0==-1}
void foo(float x) {
// T(n) {n==0}
    int n = 0;
// T(n) {n#init==0}
    while (1) {
// T(n) {n<=n#init+1}
        if (x) {
// T(n) {n<=60, n<=n#init+1}
            if (n<60)
// T(n) {n==n#init+1, n<=60}
                n++;
            else
// T(n) {n==0, 60<=n#init}
                n = 0;
        }
    }
}
```

Listing 10.2: Transformers for `counter`

10.1.1 Generation of Transformers by PIPS

The algorithms used in PIPS assume no cycles in the call graph and proceeds as follows. First, each program command S, elementary or compound statement or procedure call, is over-approximated by an affine transformer $\mathcal{T}(S, P)$, possibly using information about a precondition P of S. This is a bottom-up procedure, detailed in the next paragraphs, because a default value can be used for the precondition P when no information is available. Each function is analyzed once and its transformer is reused at each call site. Then, preconditions are propagated from the program starting point using the transformers.

This approach can also be used with unstructured programs: the control flow graphs are either turned into equivalent structured graphs [Amm92], or approximated and simplified by adding transitions, or analyzed using a decomposition into cycles [Bou92].

```
void foo(float x) {
// P() {}
   int n = 0;
// P(n) {n==0}
   while (1)
// P(n) {0<=n, n<=60}
      if (x)
// P(n) {0<=n, n<=60}
         if (n<60)
// P(n) {0<=n, n<=59}
            n++;
         else
// P(n) {n==60}
            n = 0;
}
```

Listing 10.3: Preconditions for `counter`

The two-stage approach used by PIPS is the following. We suppose that elementary instructions have been turned into transformers, and show how control structures are handled.

Sequence

Let x_i, x_i' and x_i'' denote values of variable x_i at different states. A sequence of affine transformers "T_1 followed by T_2" is overapproximated by the union of constraints in T_1 (on values $x_1, \ldots, x_n, x_1', \ldots, x_n'$) with constraints in T_2 (on values $x_1', \ldots, x_n', x_1'', \ldots, x_n''$), then projected on $x_1, \ldots, x_n, x_1'', \ldots, x_n''$ to eliminate the "intermediate" values $x_1', \ldots, x_n'$. We note this operation $T_2 \circ T_1$.

Choice

The effect of a choice "T_1 or T_2" is the transformer $T_1 \cup T_2$, which is not affine in the general case (the union of two convex polyhedra is not a convex polyhedron). The best convex approximation is the convex union $T_1 \sqcup T_2$. This is a lossy operation in general.

Loop

Given an affine transformer T for a loop body, the affine transformer T^* represents the effect of any number of iterations of T. It is computed by the *Affine Derivative Closure algorithm* [ACI10].

As noticed in Section 9.1, the two main sources of imprecision that appear are the abstractions of loops ($*$) and parallel paths ($\sqcup$). Their impact is cumulated when multiple control paths appear within loops and nested loops.

These imprecisions are accumulated if there are structures in the automaton with two or more loops involved in the same control. The state splitting heuristic described in Section 9.1 was developed with regards to this issue. Two alternative possibilities are explored in this chapter: transformer lists and iterative analysis.

Note that the pure bottom-up approach may be used or not. Since transformers are not computed concurrently but by traversing the AST, information gathered previously can be used right away. The range of a transformer or the condition of a test can be used as a

precondition for the next statement to improve the accuracy of $\mathcal{T}$. This explains, for instance, why condition $n \leq 60$ appears in the transformer of statement `n++;` in Listing 10.2.

10.1.2 Generation of Invariants

Invariants, also known as *preconditions*, are forward propagated from the initial state of the program using the transformers computed during the previous phase. However, accuracy is improved when some preconditions are recomputed directly for compound statements. For instance, the postcondition of a conditional `if (`c`)` $T_\top$ `else` $T_\bot$ can be obtained either as the convex hull of the postconditions of the two branches:

$$\text{Post} = (T_\top \circ T_c)\,(\text{Pre}) \sqcup (T_\bot \circ T_{\neg c})\,(\text{Pre}),$$

or as the precondition transformed by the conditional transformer:

$$\text{Post} = (T_\top \circ T_c \sqcup T_\bot \circ T_{\neg c})\,(\text{Pre}).$$

The first equation provides more accurate results at little cost, because the branch postconditions are computed anyway.

10.2 Modularity and Accuracy: Experimental Results

PIPS has been developed as a compilation framework, able to process large applications interprocedurally [NI05]. It is important to check that the modularity provided by transformers results in the expected speed improvement and to measure the extent of its negative impact on accuracy. We show firstly that PIPS obtains accurate results in a small amount of time with respect to three other tools, Aspic, iscc and PAGAI, when dealing with loop nests and procedure calls. We then recall previous experimental results showing a lack of accuracy and large execution times when dealing with the test cases shown in Chapter 9 to illustrate invariant generation algorithms.

10.2.1 Tools Used

As in Section 8.2.5, we choose to compare PIPS with Aspic and iscc using the ALICe framework. Aspic and iscc's transitive closure functions use as input a state machine format, fsm, or a proprietary relational format. Unlike PIPS, these tools deal neither with the intricacies of C nor with their automatic abstraction. These differences in formats play an important role in the execution times obtained in Tables 8.2 and 9.1, as will be seen below.

For comparison purposes, we also use another polyhedral analyzer for C programs, PAGAI[†1], by Henry *et al.* [HMM12a, HMM12b]. PAGAI — "Path Analysis for invariant Generation by Abstract Interpretation" — is a static analyzer working over the LLVM compiler infrastructure, which computes inductive invariants on the numerical variables of the analyzed program. PAGAI implements various state-of-the-art algorithms combining abstract interpretation and decision procedures (SMT-solving), focusing on distinction of paths inside the control flow graph while avoiding systematic exponential enumerations.

Since the tools have very different structures and execution times, we report either directly the sum of the User and IO times reported by the `time` command for Aspic, iscc and PAGAI, or the times obtained using `LOG_TIMINGS` for the sole transformer and precondition passes of PIPS. In this way, the C parsing part of PIPS is eliminated, as it is for Aspic and iscc who use internal formats and for PAGAI who uses Clang for parsing, and we compare the execution time of the

[†1] See `http://pagai.forge.imag.fr/`.

passes of PIPS that might be replaced by new passes based upon the other tools. Furthermore, the evolution of the execution times for each tool is fully relevant and interesting.

Benchmarks were run on a computer powered by a quad-core Intel i7-2600 processor clocked at 3.40 GHz with 16 GB of memory, using Aspic version 3.3, iscc version 0.12.2, PAGAI version 14-04-07 and PIPS revision 22 241.

10.2.2 Impact of Cycle Nesting on Convergence

Nested loops are common in scientific codes and easy to analyze with transformers. However, a 2D loop nest is used by Halbwachs in [HH12] to show improvements in widening techniques. Let us consider the code in Listing 10.4, a matrix multiplication. To check that all array accesses are safe, invariants on `i`, `j` and `k` are needed, which requires the analysis of a 3D loop nest.

```
void mm(int l, int n, int m, float A[l][m], float B[l][n], float C[n][m]) {
    int i, j, k;
    for (i=0; i<l; i++) {
        for (j=0; j<m; j++) {
            A[i][j] = 0.;
            for (k=0; k<n; k++) {
                A[i][j] += B[i][k]*C[k][j];
            }
        }
    }
}
```

Listing 10.4: Matrix multiplication: $A = B \times C$

```
void mp(int n, int p, float A[n][n], float B[n][n]) {
    int i, j, k;
    for (i=0; i<n; i++) {
        for (j=0; j<n; j++)
            A[i][j] = B[i][j];
    }
    for (k=1; k<p; k++) {
        float T[n][n];
        for (i=0; i<n; i++) {
            for (j=0; j<n; j++) {
                T[i][j] = A[i][j];
            }
        }
        mm(n, n, n, A, T, B);
    }
}
```

Listing 10.5: Matrix exponentiation: $A = B^p$

To understand time behaviors of the tools, we measured the execution times of the transformer and precondition passes for nest depths of one to nine. Each loop has a zero lower bound and a symbolic upper bound, `for (i_k=0; i_k<b_k; i_k++)`, and the loop body is empty, `;`. No information about the upper loop bounds `b_k` is available: each loop can be either entered or skipped. The results are shown in Table 10.1. The execution time of tools using transformers or acceleration is not an affine function of the depth because the number of variables increases

```
void mp(int n, int p, float A[n][n], float B[n][n])
{
    int i, j, k;
    for (i = 0; i <= n-1; i += 1)
        for( j = 0; j <= n-1; j += 1)
            A[i][j] = B[i][j];
    for (k = 1; k <= p-1; k += 1) {
        float T[n][n];
        for (i = 0; i <= n-1; i += 1)
            for (j = 0; j <= n-1; j += 1)
                T[i][j] = A[i][j];
        {
            int i, j, k;
            for (i = 0; i <= n-1; i += 1)
                for (j = 0; j <= n-1; j += 1) {
                    A[i][j] = 0.;
                    for (k = 0; k <= n-1; k += 1)
                        A[i][j] += T[i][k]*B[k][j];
                }
        }
    }
}
```

Listing 10.6: Matrix exponentiation, inlined with PIPS

twice as fast as the depth and because the polyhedral operators are known for their exponential worst case complexities. However, it does not increase very fast, especially for the nesting depths usual in functions when inlining is not used.

Depth	**1**	**2**	**3**	**4**	**5**	**6**	**7**	**8**	**9**
Aspic	0.037	0.043	0.040	0.053	0.047	0.063	0.067	0.087	0.100
iscc	0.000	0.010	0.037	0.083	0.370	0.853	1.197	7.927	5.713
PAGAI	0.067	0.187	0.420	0.797	1.373	2.260	3.620	5.780	9.643
PIPS	0.004	0.009	0.015	0.021	0.030	0.039	0.053	0.071	0.090

Table 10.1: Times in seconds for invariant analyses of empty nested `for` loops with varying depths

10.2.3 Interprocedural Analysis or Inlining

The code in Listing 10.5, a matrix exponentiation, `mp`, contains a call to a matrix multiplication, `mm`, within a loop. Function `mp` is called from a `main` function that reads a matrix and an exponent, and prints the resulting matrix. This code can be analyzed interprocedurally or intraprocedurally after inlining, as show in Listing 10.6. Table 10.2 contains execution time measurements for main programs calling `mp` from one to five times. Each measurement was performed 10 times and the median time is displayed. The code is either analyzed interprocedurally or intraprocedurally after the callees are inlined, which reduces the number of call sites to zero. Modularity becomes more and more useful when multiple call sites of the same function are present in the analyzed program. Also, inlining increases the loop nest depths, which is bad for the execution time as seen in the section above.

	main-1		main-2		main-3		main-4		main-5	
	2 calls	inlined	3 calls	inlined	4 calls	inlined	5 calls	inlined	6 calls	inlined
Aspic	–	0.043	–	0.061	–	0.087	–	0.108	–	0.149
iscc	–	261.810	–	274.580	–	370.960	–	413.300	–	456.360
PAGAI	0.980	1.417	1.383	5.680	2.030	14.677	2.990	30.007	4.467	53.247
PIPS	0.048	0.043	0.049	0.063	0.048	0.084	0.050	0.108	0.051	0.127

Table 10.2: Times in seconds for interprocedural analyses and intraprocedural analyses of inlined versions

10.3 New Improvements in Transformer Computation

We present two new improvements for transformer-based analyses. The first one deals with concurrent loops, and the second one with integer overflows.

10.3.1 Concurrent Loops

We are dealing with structured code. A control path set is built for each loop body. When a test or a loop is found, each pre-existing control path is duplicated to take into account the true and false branches, or the loop entrance and skip. This is not performed recursively down the branches or the inner loop bodies. Thus the total number of control path is at most 2^k, where k is the number of (possibly compound) statements in the loop body.

Control-Path Transformers in Loops

Let us assume that a loop contains several control paths, each defined by its transformer T_i. Such loops are called *concurrent* loops above. The loop precondition, P^*, can be decomposed into a set of preconditions, each obtained after a variable number of iterations and merged together by convex hull operators, $\sqcup$: We propose to decompose the loop precondition P^* as:

$$P^* = P_0 \sqcup P_2 \sqcup P_+ \sqcup P_{3+} \tag{10.1}$$

P_0 is the precondition holding the first time the loop is entered. P_2 is obtained after two iterations with different control paths. P_+ corresponds to the case when only one control path is used for all iterations; it includes P_1, the precondition of the second iteration. Finally, P_{3+} is the loop precondition corresponding to all the longer control paths with at least three iterations and two different control paths.

$$P_2 = \bigsqcup_i \bigsqcup_{j \neq i} T_i(T_j(P_0)) \qquad P_+ = \bigsqcup_i T_i^+(P_0) \qquad P_{3+} = \bigsqcup_i \bigsqcup_{j \neq i} T_i^+(T_j(C^*(P_1)))$$

C^* is an over-approximation of the transitive closure of the convex hull C of the transitions, $C = \bigsqcup_i T_i$ and P_1 is the precondition after one iteration exactly, $\bigsqcup_i T_i(P_0)$.

When the default formula in PIPS, $P^* = P_0 \sqcup C^+(P_0)$, is used, the convex hull operations are mostly performed before the transitive closure, in the transformer space, when computing C^+. With Equation (10.1), they are executed later in the precondition space and each elementary transition T_i is applied as last transition. Hence, the information brought by idempotent transformations is preserved.

The formula used to compute the loop invariant has been unrolled to take into account explicitly up to three iterations. It is possible to generalize this to k steps, with an exponential increase in the number of terms. But we are lacking experimental cases justifying such a development.

Halbwachs *et al.* present in [HH12] the example in Listing 10.1. The reset to 0 is abstracted by a transformer that cannot be merged accurately with a transformer abstracting a conditional

increment and no information is obtained with the equations presented in [ACI10]. However, Equation (10.1) provides an accurate loop invariant, $0 \leq n \leq 60$, by combining them, their transitive closures and the initial loop precondition.

Motivations for Equation (10.1)

Equation (10.1), combined with the definitions of P_2, P_+ and P_{3+}, was developped:

1. To transfer as much as possible convex hulls performed in the transformer space into convex hulls performed in the invariant (pre- and postcondition) space; for instance, the merge of an incrementation and a decrementation results in no information, regardless of the conditions used to control their executions.
2. C^*, the transitive closure of C, the convex hull of the transformers related to each control path, is often imprecise. Information, especially idempotent components, can be restored using each control path transformer T_i as last step for the iterations. This remark can be generalized to T_i^+.
3. In some cases, control path i can follow control path j, but j cannot follow i ($T_j \circ T_i = \emptyset$). By considering the two last different kinds of iterations, some impossible multi-iteration paths are eliminated.

10.3.2 Arbitrary-Precision Numbers

With a few test cases, another issue appears with PIPS. Intermediate computations may generate polyhedra with huge coefficients, leading to arithmetic overflows even with 64-bit integers, because constraint constants are transformed into coefficients by the convex hull operator. Eventually, when an overflow occurs, some constraints must be dropped, and the resulting invariant is less accurate than it should. To address this problem, we added GMP support to some of the PIPS polyhedral operators. This seemingly purely practical implementation decision allows for a drastical simplification of the polyhedral algorithms because overflow exceptions no longer have to be handled. It turns out that this positively impacts both the execution time (about 6 times faster now) and the comparison between transformer- and precondition-based analyses.

10.4 Other Improvements in Transformer Computation

We recall several improvements for transformer-based analyses. They can either be applied directly to the source codes, or to existing analyses. They have already been published in [ACI10, Mai12] but they all still required implementation and experimentation.

10.4.1 Iterative Analysis

It is sometimes possible to improve the loop preconditions P^* by recomputing the transformers a second time, using the first set of preconditions as input to limit their domains and ranges.

As explained in [ACI10], the iterative relationship between transformers and preconditions is formalized by the two equations below where B stands for the loop body statement and the continuation condition, $\mathcal{T}$ for the function that converts a statement and its precondition into a convex transformer, P_0 represents the initial precondition entering the loop, and n is positive:

$$T_{n+1} = \mathcal{T}(B, P_n^*) \cap P_n^*, \qquad P_0^* = \top, \qquad P_{n+1}^* = P_0 \sqcup T_{n+1}\big(T_{n+1}^*(P_0)\big)$$

Note that the n-th loop precondition P_n^*, which can be computed more precisely using Equation (10.1), impacts the $(n+1)$-th transformer T_{n+1} in two different ways. The affine abstrac-

tion function $\mathcal{T}$ is sharpened and the domain of the resulting transformer also is restricted by the previous precondition, P_n^*.

The iterative refinement process does not always converge and it may even lead to a precision loss, due to magnitude overflows. These can be addressed with arbitrary precision numbers (Section 10.3.2).

```
void foo(int flag, float x) {
    int i, j = 1, a = 0, b = 0;
    if (flag) {
        i = 0;
    }
    else {
        i = 1;
    }
    while (x > 0.) {
        a++;
        b += j - i;
        i += 2;
        if (i % 2 == 0) {
            j += 2;
        }
        else {
            j++;
        }
    }
    if (flag) {
        assert(a == b);
    }
}
```

Listing 10.7: Example by Dilig *et al.*

Iteration 1:

```
...
while (x > 0.) {
// P(a,b,i,j) {2a==i, j<=2a+1, a+1<=j}
...
```

Iteration 2:

```
...
while (x > 0.) {
// P(a,b,i,j) {2a==i, 2a==j-1, 0<=a, b<=a}
...
```

Iteration 3:

```
...
while (x > 0.) {
// P(a,b,i,j) {a==b, 2a==i, 2a==j-1, 0<=a}
...
```

Listing 10.8: Loop invariants for Listing 10.7 obtained iteratively when `flag != 0`

The example in Listing 10.7 was published in [DDLM13]. Function `foo` has two different behaviors depending on the value of `flag` and it uses a non-affine condition in the loop, `i % 2 == 0`. Different behaviors that depend on a formal parameter can be separated by partitioning the function and values returned by the modulo operator can be analyzed precisely when information about the parity of `i` is known. When `flag` is not zero, it is possible to derive that `a` equals `b`. To obtain three equations in the function postcondition, transformers must be computed three times because each time a precondition makes the abstraction of a statement more precise (see Listing 10.8).

10.4.2 Periodicity

Periodic behaviors occur in scientific computing, when, for instance, two sub-arrays of a matrix `A` are swapped, to avoid copying new values in the locations of old values at each time step `x`: `A[new][*] = f(A[old][*])`, at the cost of a mere swapping of the indices. See Listing 10.9 where `new` and `old` are two variables used in practice to index an array where new values are computed at each iteration as a function of old values contained in the same array. To parallelize such programs, the compiler must find the invariant `new + old == 1`, which is then used in data dependence testing together with the conflict equation, `new == old`, to show that `A[new][*]` and `A[old][*]` refer to different sets of locations.

```
int main() {
    int x = 0, new = 0, old = 1, y = 0, z = 0;
    while (x < 10) {
        if (new == 0) {
            y++;
        }
        else {
            z++;
        }
        new = 1 - new;
        old = 1 - old;
        x++;
    }
    if (new == 1 && old == 0 || new == 0 && old == 1) {
        printf("property verified\n");
    }
    else {
        printf("property not found\n");
    }
}
```

Listing 10.9: Periodic behavior

As pointed out in [ACI10], there are different ways to encode the swap, and the above invariant may be more or less easy to generate. However, regardless of the encoding, the behavior is always periodic. More information is preserved if the transitive closure of the loop transformer is computed as a function of one of its powers [ACI10]. For instance, the square is useful for idempotent functions and functions with a period of 2:

$$T^* = \left(T^2\right)^* \sqcup T \circ \left(T^2\right)^*, \qquad T^+ = T \sqcup \left(T^2\right)^+ \sqcup T \circ \left(T^2\right)^+.$$

This can be generalized to any power of T. As no application has yet required a larger periodicity, our current implementation in PIPS is limited to T^2.

Iteration 1:

```
...
while (x < 10) {
// P(new,old,x,y,z) {new+old==1, y+z==x, 0<=new, new<=1, new<=x, x <= 9,
// y<=x, 0<=y}
    ...
}
// P(new,old,x,y,z) {new+old==1, x == 10, y+z==10, 0<=new, new<=1,
// new<=y, y<=new+9}
...
```

Iteration 2:

```
...
while (x < 10) {
// P(new,old,x,y,z) {new+old==1, new+x == 2y, new+z==y, 0<=new, new<=1,
// new<=y, y<=new+4}
    ...
    }
// P(new,old,x,y,z) {new==0, old==1, x == 10, y==5, z==5}
...
```

Listing 10.10: Loop invariants and postconditions for Listing 10.9

10.4.3 While-If to While-While Conversion

Note finally that an iterative analysis (see Section 10.4.1) improves the invariant for the periodic function: the relationship between `y` and `z` is found (see Listing 10.10). As for the Dilig example in Listing 10.7, and unlike what is claimed in [ACI10], the iterations are not linked only to non-polyhedral invariants, but also to polyhedral invariants and they may converge.

This optimization, the conversion of tests into while loops inside a while loop, was also used in [ACI10] and proved correct in the extended version, but it has not been implemented, neither explicitly with a program transformation nor implicitly when computing loop invariants. In fact, it may have a detrimental effect with respect to using the convex hull to obtain a unique loop transformer, unless control path transformers are used. The example in Listing 10.11 [Mas14] shows how easier it is to prove the loop termination once the internal test has been changed into a pair of while loops.

10.5 Experimental Evaluation of the Improvements

Although PIPS has not been designed to analyze the small test cases used in academic papers dealing with loop invariants and termination and discussed in the previous chapters, it is yet interesting to use them to measure the impact of the improvements presented above and to compare PIPS to other tools able to compute invariants. Also, test cases left unsolved are good starting points for designing new improvements in invariant generation.

10.5.1 Impact of Improvements for PIPS

Table 10.3 provides the numbers of successes and the total execution times obtained with the four different encodings of ALICe benchmark (Chapter 9) by Aspic, iscc, the legacy version of PIPS and PIPS with some of the improvements defined in Section 10.3. Accuracy figures for

```
void masse_vmcai_2014_14(int x) {
    while (x != 0) {
        if (x > 0) {
            x--;
        }
        else {
            x++;
        }
    }
}

void masse_vmcai_2014_14_transformed(int x) {
    while (x != 0) {
        while (x != 0 && x > 0) {
            x--;
        }
        while (x != 0 && x <= 0) {
            x++;
        }
    }
}
```

Listing 10.11: While-if to while-while conversion: Example by Massé *et al.*

Aspic, iscc and the default version of PIPS are reproduced from Section 9.4 for comparison purposes.

		Aspic	**iscc**	**PIPS**				
				Default	CP	IA	CP-IA	CP-IA-MP
Direct	Successes	75	63	43	69	45	72	73
	Time (s.)	10.9	35.5	6.1	7.8	18.5	19.6	151.4
Split	Successes	79	72	50	72	56	75	77
	Time (s.)	12.8	43.0	5.7	6.8	14.4	19.0	113.3
Merged	Successes	59	70	40	66	44	67	68
	Time (s.)	16.7	26.2	6.2	8.0	18.5	19.7	225.9
Merged	Successes	70	83	63	79	65	80	82
& Split	Time (s.)	11.3	40.8	6.6	9.6	16.8	23.0	222.2

Table 10.3: Successes and execution times for ALICe test cases with different options (Control Path transformers, Iterative Analysis, MultiPrecision).

As expected, the execution times increase and the benefit is sometimes quite small for the test cases in ALICe. However, using sets of control path transformers and a favorable encoding provides excellent results in the number of test cases solved per second, and future benchmarks for invariant generation may have different profiles. Note that periodic analysis (Section 10.4.2), not shown in Table 10.3, does not improve any of the test cases currently in ALICe.

Sizes in number of lines are indicated in Table 10.4 for FAST files and for C files generated by fsm2c. C file sizes are reduced by eliminating files greater than 1 KLOC and by a first PIPS pass (control simplification).

Note also that the execution times of PIPS in Table 10.3 differ widely from those shown in

	fsm	C	
Direct	5497	Raw	15482
		Simplified	4323
Split	9348	Raw	757222
		Simplified	4199
Merged	5579	Raw	15380
		Simplified	5187
Merged & Split	6206	Raw	38941
		Simplified	4549

Table 10.4: Size of fsm and C files in number of lines

Chapters 8 and 9. In previous chapters, the execution time is:

$$t_{\text{naive}} = \sum_{i \in \mathsf{ALICe}} \text{time}\Big(\mathsf{PIPS}\big(\mathsf{fsm2c}(case_i)\big)\Big)$$

that is the sum of the PIPS processing time for all test cases in ALICe converted into C by fsm2c. This is highly detrimental to PIPS because of its large startup time, because number of passes, such as parsing, are not performed by other tools that are input with a simpler native format and because fsm2c has also negative impacts discussed in the next paragraph.

The C encoding chosen for fsm2c implies the parsing of the `stdlib` header, an interprocedural analysis by PIPS, lots of unreachable code and some exponential blow-ups in the generated code sizes. In order to reduce the overheads incurred by PIPS, we eliminate test cases which cannot be regenerated in less than 1000 lines, we replace the functional encoding of aleas by uninterpreted expressions, we use PIPS to eliminate unreachable statements and we process all test cases in one call to PIPS. Formally, the execution time is now:

$$t = \text{time}\left(\mathsf{PIPS}\left(\mathsf{PIPS}\left(\text{substitute}\Big(\bigcup_{\substack{i \in \mathsf{ALICe} \\ \text{Csize}(i) < 1000}} \mathsf{fsm2c}(case_i)\Big)\right)\right)\right)$$

which reduces the PIPS execution time by a factor of 7 to 10.

10.5.2 Analysis of PIPS Failures

The ALICe benchmark contains 102 test cases, listed in Table 8.1, page 86. Out of them, there are 9 test cases that involve non-Presburger invariants and PIPS finds invariant for 82. This leaves 11 cases to investigate further, namely:

- `halbwachs7`.
- `henry`.
- `metro` and `subway`.
- `microsoftex2`.
- `microsoftex5`.
- `popeea`.
- `realheapsort` and `realheapsort_step2`.
- `synergy_bad`.
- `ticket`.

The next paragraphs discuss the reasons of the failures, and the improvements that can be introduced to solve them.

ALICe is based on the FAST format and the heuristics fsm2c used to generate structured C from FAST control flow graphs may blow up exponentially, up to 300 KLOC (`metro`, `realheapsort`, `realheapsort_step2` and `subway` at the very least).

When some test cases are written in C, using while loops as they were published, the invariants for `henry`, `halbwachs7` and `synergy_bad` [GHK+06] are found by PIPS.

Case `microsoftex2` is analyzed in [GZ10] to detect loop termination. The required information is not an invariant but a transformer, and PIPS computes the required transformer. Case `microsoftex5` is also analyzed in [GZ10], using non-convex transformers. It contains two nested while loops, and the internal loop may or may not be the identity function. The information about the identity behavior is not preserved by PIPS because control paths are not built recursively going down loops to keep their number small. This test case requires a different algorithm to construct control paths in PIPS.

Case `popeea` [PC07, GZ10] has a non-convex invariant, which can only be found by a convex tool if new control points are added. The heuristics used by ALICe, `fsmnodesplit`, probably fails to discover a proper node splitting. Case `ticket` [BGP97] is interesting because it is easy to make the invariant convex. However, the number of control paths is large and the convex hulls used by PIPS lead to overflows when the while loop is unrolled. The algorithm is described as a transition system and the way it may be coded in C is critical to its analysis by PIPS.

*

Conclusion

(English version follows.)

Un nombre croissant de systèmes critiques, dont les dysfonctionnement peuvent avoir des conséquences désastreuses, sont aujourd'hui basés sur des composants informatiques. C'est le cas par exemple des commandes de vol des avions de ligne, des appareils médicaux ou des centrales nucléaires. À cause de la complexité toujours plus élevée des logiciels, la probabilité d'un vice de conception ou d'une erreur de programmation augmente. Aussi, les méthodes formelles sont de plus en plus souvent utilisées en complément des bancs de test et des normes de certification de code, pour fournir des garanties mathématiques de la correction des programmes, et ainsi atteindre une meilleure fiabilité. Ces méthodes proposent différents niveaux de précision et d'automatisation.

La vérification automatique repose en général sur le model checking et l'interprétation abstraite, une famille de méthodes efficaces pour générer automatiquement des invariants de programme, c'est-à-dire des propriétés mathématiques qui sont toujours vraies lors de l'exécution du programme, à différents endroits dans le code. Ces invariants sont ensuite utilisés pour assurer que le programme respecte sa spécification. Ainsi, il est crucial de générer des invariants précis, afin d'avoir des garanties sur les comportements complexes du programme. Les contributions présentées par ce manuscrit s'inscrivent dans ce cadre.

Contributions

Systèmes linéaires

La première partie de cette thèse concerne les sytèmes de contrôle, des dispositifs qui régulent le comportement d'autres systèmes. Les systèmes de contrôle sont largement utilisés dans l'industrie, et interviennent dans un grande variété d'applications. Un sujet important en théorie du contrôle est de garantir la stabilité d'un système de contrôle, c'est-à-dire que le système démarré près d'un point d'équilibre restera dans un voisinage de ce point tout au long de l'exécution. La théorie du contrôle énonce qu'un système de contrôle linéaire est stable s'il existe un invariant caractéristique, appelé invariant de Lyapunov, et de nombreux systèmes sont conçus et prouvés en utilisant ce théorème. Mais les preuves de stabilité de ces systèmes reposent sur l'arithmétique des nombres réels, et ne s'appliquent pas nécessairement lorsque le système est implémenté sur une architecture avec une arithmétique machine.

Nous introduisons un nouveau framework pour traduire les invariants de preuves en nombres réels vers des invariants en nombres machine, en prenant en compte la représentation des nombres et les problèmes d'arrondis. Les invariants traduits sont exacts, mais ne garantissent pas nécessairement la stabilité du système. Le schéma de traduction générique a été vérifié à l'aide de Coq, et est implémenté pour l'arithmétique à virgule flottante, la plus répandue des arithmétiques machine. Cette implémentation, LyaFloat, peut vérifier automatiquement la stabilité d'un système sur des architectures à virgule flottante, et calculer le nombre de bits nécessaires à la fois pour la stabilité en boucle ouverte et en boucle fermée.

Analyse des relations linéaires

La deuxième partie de ce manuscrit concerne l'analyse des relations linéaires, une interprétation abstraite courante, basée sur la surapproximation des états numériques accessibles dans un programme par des polyèdres convexes. Il existe aujourd'hui plusieurs outils pour l'analyse des relations linéaires. Ils disposent d'un grand nombre d'algorithmes et d'heuristiques pour traiter les boucles, en essayant d'obtenir la meilleure précision possible. Nous proposons un nouveau framework, ALICe, pour comparer aussi équitablement que possible trois de ces outils : Aspic, iscc et PIPS, sur un jeu commun de 102 petits cas de test, venant de publications traitant de problèmes d'invariants ou de terminaison de boucles. Les expériences effectuées ont permi d'aboutir à deux résultats originaux.

Premièrement, le choix de l'encodage est crucial pour réussir à analyser un modèle. Ce point est exploré au moyen de deux algorithmes de restructuration de modèles, que nous avons développés et prouvés en Coq. Le premier réduit les biais d'encodage en fusionnant le modèle en un unique point de contrôle. L'autre partitionne les points de contrôle qui ont des boucles concurrentes, qui sont difficiles à analyser. Utilisés séparément ou ensemble, ils permettent des améliorations significatives des résultats d'analyse et aide à identifier les forces et les faiblesses de chaque outil d'analyse.

La seconde observation expérimentale est que les algorithmes utilisés par défaut dans PIPS manquent de précision dans le cas de boucles concurrentes. Contrairement à Aspic et iscc, le calcul des invariants dans PIPS repose sur une première étape pour calculer les transformeurs, qui sont ensuite utilisés pour propager les invariants. La meilleure scalabilité de cette approche est utile pour analyser de gros volumes de code, au prix d'approximations supplémentaires. Nous présentons de nouvelles améliorations dans le calcul des transformeurs qui augmentent la précision des invariants dans PIPS. En les combinant avec d'autres techniques existantes, il est possible d'obtenir avec PIPS des performances comparables à celles d'Aspic et iscc.

Perspectives

Systèmes linéaires

Concernant les perspectives, plusieurs améliorations peuvent être apportées dans l'outil de traduction d'invariants LyaFloat. Actuellement, seule l'arithmétique à virgule flottante, la plus répandue, est supportée ; il serait relativement facile de supporter l'arithmétique à virgule fixe, au moins en théorie (l'implémentation est rendue difficile par l'absence de bibliothèque de calcul en virgule-fixe et précision paramétrique en Python). Dans la même veine, il serait intéressant de supporter davantage de fonctions et d'arguments de propagation, afin d'étendre la portée de l'outil.

Dans sa forme actuelle, LyaFloat génère des invariants pour du code en arithmétique machine en suivant un modèle formellement prouvé, mais il n'y a pas de garantie formelle que LyaFloat lui-même n'est pas défectueux. On gagnerait un niveau d'assurance s'il était possible de vérifier par un autre moyen les invariants générés. Une solution serait de réécrire et de formaliser LyaFloat en Coq plutôt que d'utiliser Python. Cela nécessiterait de prouver (ou au moins de formaliser) les arguments de propagation d'invariants requis dans les preuves de stabilité, ce qui représenterait un travail conséquent. Une autre idée serait de générer avec les invariants des scripts Coq qui feraient office de certificat de correction pour la propagation d'invariants, ce qui serait suffisant pour vérifier facilement les invariants générés. Cela nécessiterait de développer une bibliothèque Coq de théorèmes autour de la théorie de la stabilité.

D'un point de vue général, de nombreuses étapes sont nécessaires pour traduire un modèle de contrôleur linéaire idéal et prouvé vers le code C de bas niveau qui l'implémente, et changer la représentation arithmétique n'est que l'une de ces étapes. Parmi les autres étapes, on trouve la

discrétisation d'un contrôleur continu, ce qui pose le problème de garantir des bornes de temps de calcul dans la boucle de contrôle, ou l'acquisition des données, qui est implémentée sur la plupart des contrôleurs sous la forme de routines de traitement d'interruptions.

Analyse des relations linéaires

Dans ALICe, la manipulation de code source en C s'est avérée problématique. Actuellement, tous les cas de test sont implémentés sous forme de machines à états finis, dans le format fsm, et sont compilés en C avant d'être analysés par PIPS. La transformation de fsm en C n'est pas simple et les codes C générés tendent à être excessivement longs et complexes et donc difficiles à analyser, ce qui désavantage PIPS en comparaison avec Aspic et iscc. Même si le problème peut être en partie contourné avec une phase intermédiaire de simplification, nous pensons qu'il faudrait employer une meilleure stratégie pour les codes C, que ce soit en enregistrant une version C de chaque modèle en plus de la version fsm (ce qui amène d'autres problème : choix des représentations, expressivité, cohérence, encodage), ou bien en améliorant considérablement l'outil fsm2c afin de résoudre ces problèmes.

Le domaine d'application d'ALICe pourrait également être étendu, d'une part en ajoutant de nouveaux modèles à la base de cas de test, et d'autre part en supportant davantage d'outils d'analyse. Cette dernière tâche demande du temps puisqu'à cause du manque de standardisation des formats utilisés par les outils d'analyse, il faut écrire des routines de conversion de formats pour chaque outil ajouté. Il faudrait aussi vérifier que les outils d'analyse génèrent des invariants réalistes, par exemple en associant à chque cas de test un invariant minimal. Un outil d'analyse ne serait considéré correct que si les invariants qu'il génère sont des sur-ensembles des invariants minimaux. Cela éviterait qu'un outil trivialement faux puisse bouleverser les résultats.

Finalement, même si les améliorations apportées au calcul d'invariants par transformeurs ont permis d'améliorer significativement les résultats dans PIPS, les coûts de calcul les rendent impraticables pour l'analyse de gros programmes. Des algorithmes précis et de complexité raisonnable sont encore à trouver.

* * *

Computers are involved in an ever-increasing number of critical systems, whose failures may have a disastrous impact. Examples include flight commands, medical devices or nuclear power plants. Due to the growing complexity of software, the probability of a misconception or programming error is even increasing. Hence, formal methods are involved more and more often in addition to extensive testing and code certification, to provide mathematical insurances of program correctness, and thus achieve greater reliability. These methods offer various levels of accuracy and automation.

Automatic verification generally relies on model checking and abstract interpretation, an efficient family of methods to generate automatically program invariants, i.e. mathematical properties that always hold at various locations in the code during the program execution. These invariants are then used to ensure that the program meets its specifications. Thus, it is crucial to obtain accurate invariants in order to have guarantees about complex program behaviors; this manuscript has contributed to this general subject.

Contributions

Linear Systems

The first part of this thesis is about control systems, devices that regulates the behavior of other systems. Control systems are broadly utilized in industry, in a wide variety of applications. A usual topic in control theory is ensuring the stability of a control system, that is, the system

started near an equilibrium point will stay forever in its neighborhood. Control theory states that a linear control system is stable if a distinctly shaped invariant, called Lyapunov invariant, holds, and many systems are designed and proved stable accordingly. But the stability proofs on these systems rely on exact arithmetic on real numbers, and do not necessarily apply when implementing the system on a device with machine-precision arithmetic.

We introduce a new framework to translate real-arithmetic proof invariants into invariants suited for machine arithmetic, taking into account number representation and rounding error issues. The generated invariants are guaranteed to be correct, but do not necessarily ensure the system stability. This generic translation scheme is checked using Coq, and is also implemented in the case of floating-point arithmetic, the most common standard for machine arithmetic. This implementation, LyaFloat, can check automatically system stability on floating-point platforms, and compute the number of bits necessary to ensure it, for both open-loop and closed-loop stabilities.

Linear Relation Analysis

The second part of this manuscript is about linear relation analysis, a classical abstract interpretation based on an over-approximation of reachable numerical states of a program by convex polyhedra. As of today, several tools for linear relation analysis exist. They rely on a large set of algorithms and heuristics to handle loops, aiming to maximum accuracy. We propose a new toolset, ALICe, to compare as fairly as possible three of these tools: Aspic, iscc and PIPS, on a common set of 102 small-scale, test cases previously published in the loop invariant and termination bibliography. Two main original results arise from the experimental measurements.

First, the choice of encoding is critical to the success of a model analysis. This issue is explored with two model-to-model restructuring algorithms that we have developed and proved correct with Coq. The first one reduces the encoding biases, by merging the model to a unique control point. The other one splits control nodes involving concurrent loops that are difficult to analyze. Used separately or together, they lead to significant improvements in analysis results and help to identify the strengths and weaknesses of each analysis tool.

The second experimental observation is that PIPS default algorithms have poor accuracy when dealing with concurrent loops. Unlike Aspic and iscc, invariant computation in PIPS relies on a first step that computes transformers, used later to propagate invariants. The greater scalability of this approach is useful to analyze large applications, at the cost of additional approximations. We present new improvements in transformer computation to improve invariant accuracy in PIPS. Combined with other existing techniques, PIPS is able to perform at comparable levels with Aspic and iscc.

Perspectives

Linear Systems

Regarding perspectives, several improvements could be introduced in our invariant translation tool LyaFloat. Currently, only the most common floating-point arithmetic is supported; yet it would be relatively easy to add support for fixed-point arithmetic, at least from a theoretical standpoint (the implementation raises technical difficulties, due to the lack of a suitable parametric fixed-point library for Python). In the same vein, it would be interesting to add support for more functions and propagation arguments, in order to extend the scope of the tool.

In its current design, LyaFloat generates invariants for machine arithmetic code following a formally proved pattern but there is no formal guarantee that LyaFloat itself is flawless. Thus, we would gain extra confidence if we could check the generated invariants by some other means. For this, LyaFloat could be rewritten and formalized in Coq rather than Python. This would

require to prove (or at least formalize) the invariant propagation arguments needed for stability proofs, which could be a huge work. An alternative idea is to make LyaFloat generate invariants along with Coq scripts acting as certificates that the invariant propagation is correct, which would be sufficient to easily verify the generated invariants. This would require the development of a Coq library of theorems linked to stability theory.

From a general point of view, many steps are required to translate an idealized, proved linear controller model into the low-level C code that implements it, and shifting to the arithmetic representation is just one of them. Other outstanding issues include the discretization of continuous controllers, which poses the problem of guaranteeing a bounded computation time in the control loop, or the data acquisition, which is implemented on most controllers as interrupt handlers.

Linear Relation Analysis

In ALICe, dealing with C source code has proven problematic. Currently, all test cases are implemented as finite state machines in fsm format and compiled in C prior to being analyzed by PIPS. The translation from fsm to C is not straightforward and the generated C files tend to be overly long and complex: this is not optimal for analysis purposes, and is a disadvantage for PIPS in comparisons with Aspic and iscc. Even if the problem can be partially circumvented with an intermediate simplification step, we believe a better strategy should be employed to handle C code, either by storing a C version of each model alongside with the fsm version (which brings several other problems concerning representation choices, expressivity, consistency and restructuring), or by drastically improving the fsm2c tool to tackle these issues.

The scope of ALICe could be enhanced too, by collecting more models in the test suite on the one hand, and by supporting more analysis tools on the other. The later point would be time-consuming because of the lack of standardization of formats used by analysis tools, requiring conversion routines for each added analysis tool. We should also check that the analysis tools generate realistic invariants, for instance shipping each model with minimal invariants. An analysis tool would be considered as correct only if the generated invariants are superset of the minimal ones. This would prevent a trivially faulty tool to trash the results.

Finally, even if the improvements made to transformer loop computations allowed to significantly improve the analysis results in PIPS, their computational complexity makes them unsuited for large code analysis. Accurate algorithms with a decent complexity class still remain to be found.

Bibliography

[ACD93] R. Alur, C. Courcoubetis, and D. Dill. Model-Checking in Dense Real-Time. *Information and Computation*, 104(1):2–34, May 1993. doi:10.1006/inco.1993.1024.

[ACI10] C. Ancourt, F. Coelho, and F. Irigoin. A Modular Static Analysis Approach to Affine Loop Invariants Detection. *Electronic Notes in Theoretical Computer Science*, 267(1):3–16, October 2010. doi:10.1016/j.entcs.2010.09.002.

[ACIK97] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A Linear Algebra Framework for Static HPF Code Distribution. *Scientific Programming*, 6(1):3–27, 1997.

[AD90] R. Alur and D. Dill. Automata for Modeling Real-Time Systems. In *Automata, Languages and Programming*, volume 443, pages 322–335. Springer-Verlag, Berlin/Heidelberg, 1990.

[ADFG10] C. Alias, A. Darte, P. Feautrier, and L. Gonnord. Multi-Dimensional Rankings, Program Termination, and Complexity Bounds of Flowchart Programs. In *Static Analysis*, volume 6337, pages 117–133. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[Amm92] Z. Ammarguellat. A Control-Flow Normalization Algorithm and its Complexity. *IEEE Transactions on Software Engineering*, 18(3):237–251, March 1992. doi:10.1109/32.126773.

[AS87] B. Alpern and F. B. Schneider. Recognizing Safety and Liveness. *Distributed Computing*, 2(3):117–126, September 1987. doi:10.1007/BF01782772.

[Avi98] D. Avis. Computational Experience with the Reverse Search Vertex Enumeration Algorithm. *Optimization Methods and Software*, pages 107–124, 1998.

[Bar05] S. Bardin. *Vers un model checking avec accélération plate des systèmes hétérogènes*. PhD thesis, École normale supérieure de Cachan, 2005.

[BBC+00] N. S. Bjørner, A. Browne, M. A. Colón, B. Finkbeiner, Z. Manna, H. B. Sipma, and T. E. Uribe. Verifying Temporal Properties of Reactive Systems: A STeP Tutorial. *Formal Methods in System Design*, 16(3):227–270, 2000. doi:10.1023/A:1008700623084.

[BEGFB94] S. Boyd, L. El Ghaoui, E. Feron, and V. Balakrishnan. *Linear Matrix Inequalities in System and Control Theory*. Society for Industrial and Applied Mathematics, 1994.

[BFL04] S. Bardin, A. Finkel, and J. Leroux. FASTer Acceleration of Counter Automata in Practice. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2988, pages 576–590. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic Model Checking of Infinite State Systems Using Presburger Arithmetic. In *Computer Aided Verification*, volume 1254, pages 400–411. Springer Berlin Heidelberg, Berlin, Heidelberg, 1997.

[BHMR07] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. page 300. ACM Press, 2007. doi:10.1145/1250734.1250769.

[BHRZ05] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise Widening Operators for Convex Polyhedra. *Science of Computer Programming*, 58(1-2):28–56, October 2005. doi:10.1016/j.scico.2005.02.003.

[BKK13] W. Bielecki, K. Kraska, and T. Klimek. Transitive Closure of a Union of Dependence Relations for Parameterized Perfectly-Nested Loops. In *Parallel Computing Technologies*, volume 7979 of *Lecture Notes in Computer Science*, pages 37–50. Springer Berlin Heidelberg, 2013.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. InVeSt : A Tool for the Verification of Invariants. In *Computer Aided Verification*, volume 1427, pages 505–510. Springer-Verlag, Berlin/Heidelberg, 1998.

[bor95] *Specification and Validation Methods*. Oxford University Press, Inc., New York, NY, USA, 1995.

[Bou92] F. Bourdoncle. Abstract Interpretation by Dynamic Partitioning. *Journal of Functional Programming*, 2(04):407, October 1992. doi:10.1017/S0956796800000496.

[BRSV90] G. Boudol, V. Roy, R. Simone, and D. Vergamini. Process Calculi, from Theory to Practice: Verification Tools. In *Automatic Verification Methods for Finite State Systems*, volume 407, pages 1–10. Springer Berlin Heidelberg, Berlin, Heidelberg, 1990.

[Brz64] J. A. Brzozowski. Derivatives of Regular Expressions. *Journal of the ACM*, 11(4):481–494, October 1964. doi:10.1145/321239.321249.

[BRZH02] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly Not Closed Convex Polyhedra and the Parma Polyhedra Library. In *Static Analysis*, volume 2477 of *Lecture Notes in Computer Science*, pages 213–229. Springer Berlin Heidelberg, 2002.

[BW94] B. Boigelot and P. Wolper. Symbolic Verification with Periodic Sets. In *Computer Aided Verification*, volume 818, pages 55–67. Springer Berlin Heidelberg, Berlin, Heidelberg, 1994.

[CC77a] P. Cousot and R. Cousot. Abstract Interpretation: A Unified Lattice Model for Static Analysis of Programs by Construction or Approximation of Fixpoints. pages 238–252. ACM Press, 1977. doi:10.1145/512950.512973.

[CC77b] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Generalized Type Unions. pages 77–94. ACM Press, 1977. doi:10.1145/800022.808314.

[CC92] P. Cousot and R. Cousot. Abstract Interpretation and Application to Logic Programs. *The Journal of Logic Programming*, 13(2-3):103–179, July 1992. doi:10.1016/0743-1066(92)90030-7.

[CC07] R. Clarisó and J. Cortadella. The Octahedron Abstract Domain. *Science of Computer Programming*, 64(1):115–139, January 2007. doi:10.1016/j.scico.2006.03.009.

[CCF+15] P. Cousot, R. Cousot, J. Feret, A. Miné, and X. Rival. The Astrée Static Analyzer, 2001–2015. http://www.astree.ens.fr/.

[CCG+08] A. Chawdhary, B. Cook, S. Gulwani, M. Sagiv, and H. Yang. Ranking Abstractions. In *Programming Languages and Systems*, volume 4960, pages 148–162. Springer Berlin Heidelberg, Berlin, Heidelberg, 2008.

[Cen15] Centre de Recherche en Informatique, MINES ParisTech. PIPS: Automatic Parallelizer and Code Transformation Framework, 1988–2015. http://pips4u.org/.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic Verification of Finite-State Concurrent Systems Using Temporal Logic Specifications. *ACM Transactions on Programming Languages and Systems*, 8(2):244–263, April 1986. doi:10.1145/5397.5399.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model Checking and Abstraction. *ACM Transactions on Programming Languages and Systems*, 16(5):1512–1542, September 1994. doi:10.1145/186025.186051.

[CH78] P. Cousot and N. Halbwachs. Automatic Discovery of Linear Restraints among Variables of a Program. In *Conference Record of the Fifth Annual ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages*, pages 84–96, Tucson, Arizona, 1978. ACM Press, New York, NY. doi:10.1145/512760.512770.

[Cha84] B. Chazelle. Convex Partitions of Polyhedra: A Lower Bound and Worst-Case Optimal Algorithm. *SIAM Journal on Computing*, 13(3):488–507, August 1984. doi:10.1137/0213031.

[Che68] N. Chernikoba. Algorithm for Discovering the Set of All the Solutions of a Linear Programming Problem. *USSR Computational Mathematics and Mathematical Physics*, 8(6):282–293, January 1968. doi:10.1016/0041-5553(68)90115-8.

[CJ98] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Computer Aided Verification*, volume 1427, pages 268–279. Springer-Verlag, Berlin/Heidelberg, 1998.

[Com15] Computer Science Laboratory of SRI International, Menlo Park, California. PVS Specification and Verification System, 1992-2015. http://pvs.csl.sri.com/.

[Cou05] P. Cousot. Proving Program Invariance and Termination by Parametric Abstraction, Lagrangian Relaxation and Semidefinite Programming. In *Verification, Model Checking, and Abstract Interpretation*, volume 3385, pages 1–24. Springer Berlin Heidelberg, Berlin, Heidelberg, 2005.

[CPR06] B. Cook, A. Podelski, and A. Rybalchenko. Termination Proofs for Systems Code. *ACM SIGPLAN Notices*, 41(6):415, June 2006. doi:10.1145/1133255.1134029.

[CS01] M. A. Colón and H. B. Sipma. Synthesis of Linear Ranking Functions. In *Tools and Algorithms for the Construction and Analysis of Systems*, volume 2031, pages 67–81. Springer Berlin Heidelberg, Berlin, Heidelberg, 2001.

[CS02] M. A. Colón and H. B. Sipma. Practical Methods for Proving Program Termination. In *Computer Aided Verification*, volume 2404, pages 442–454. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[DDLM13] I. Dillig, T. Dillig, B. Li, and K. McMillan. Inductive Invariant Generation via Abductive Inference. *SIGPLAN Notices*, 48(10):443–456, October 2013. doi:10.1145/2544173.2509511.

[Fea10] P. Feautrier. c2fsm, 2010. http://perso.ens-lyon.fr/paul.feautrier/ensl006.html.

[Fed88] Federal Aviation Administration. System Design and Analysis. Technical Report Advisory Circular 25.1309-1A, 1988. http://www.faa.gov/documentLibrary/media/Advisory_Circular/AC25.1309-1A.pdf.

[Fer90] J.-C. Fernandez. An Implementation of an Efficient Algorithm for Bisimulation Equivalence. *Science of Computer Programming*, 13(2-3):219–236, May 1990. doi:10.1016/0167-6423(90)90071-K.

[Fer10] E. Feron. From Control Systems to Control Software. *IEEE Control Systems Magazine*, 30(6):50–71, December 2010. doi:10.1109/MCS.2010.938196.

[FG10] P. Feautrier and L. Gonnord. Accelerated Invariant Generation for C Programs with Aspic and C2fsm. *Electronic Notes in Theoretical Computer Science*, 267(2):3–13, October 2010. doi:10.1016/j.entcs.2010.09.014.

[Flo93] R. W. Floyd. Assigning Meanings to Programs. In *Program Verification*, volume 14, pages 65–81. Springer Netherlands, Dordrecht, 1993.

[FP96] K. Fukuda and A. Prodon. Double Description Method Revisited. In *Combinatorics and Computer Science*, volume 1120 of *Lecture Notes in Computer Science*, pages 91–111. Springer Berlin Heidelberg, 1996.

[FS00] A. Finkel and G. Sutre. An Algorithm Constructing the Semilinear Post for 2-Dim Reset/Transfer VASS. In *Mathematical Foundations of Computer Science 2000*, volume 1893, pages 353–362. Springer Berlin Heidelberg, Berlin, Heidelberg, 2000.

[GHK+03] G. Gierz, K. H. Hofmann, K. Keimel, J. D. Lawson, M. Mislove, and D. S. Scott. *Continuous Lattices and Domains*, volume 93 of *Encyclopedia of Mathematics and its Applications*. Cambridge University Press, 2003.

[GHK+06] B. S. Gulavani, T. A. Henzinger, Y. Kannan, A. V. Nori, and S. K. Rajamani. SYNERGY: A New Algorithm for Property Checking. page 117. ACM Press, 2006. doi:10.1145/1181775.1181790.

[GJK09] S. Gulwani, S. Jain, and E. Koskinen. Control-Flow Refinement and Progress Invariants for Bound Analysis. *ACM SIGPLAN Notices*, 44(6):375, May 2009. doi:10.1145/1543135.1542518.

[GL93] S. Graf and C. Loiseaux. A Tool for Symbolic Program Verification and Abstraction. In *Computer Aided Verification*, volume 697, pages 71–84. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[GMC08] S. Gulwani, K. K. Mehra, and T. Chilimbi. SPEED: Precise and Efficient Static Estimation of Program Computational Complexity. page 127. ACM Press, 2008. doi:10.1145/1480881.1480898.

[Gol91] D. Goldberg. What Every Computer Scientist Should Know About Floating-Point Arithmetic. *ACM Computing Surveys*, 23(1):5–48, March 1991. doi:10.1145/103162.103163.

[Gon07] L. Gonnord. *Accélération abstraite pour l'amélioration de la précision en analyse des relations linéaires*. PhD thesis, Université Joseph-Fourier - Grenoble 1, 2007.

[Gon13] L. Gonnord. Aspic: Accelerated Symbolic Polyhedral Invariant Computation, 2010–2013. http://laure.gonnord.org/pro/aspic/aspic.html.

[GR06] D. Gopan and T. Reps. Lookahead Widening. In *Computer Aided Verification*, volume 4144, pages 452–466. Springer Berlin Heidelberg, Berlin, Heidelberg, 2006.

[GR07] D. Gopan and T. Reps. Guided Static Analysis. In *Static Analysis*, volume 4634, pages 349–365. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[Gra91] P. Granger. *Analyses sémantiques de congruence*. PhD thesis, École polytechnique, Paris, July 1991.

[Gul09] S. Gulwani. SPEED: Symbolic Complexity Bound Analysis. In *Computer Aided Verification*, volume 5643, pages 51–62. Springer Berlin Heidelberg, Berlin, Heidelberg, 2009.

[GZ10] S. Gulwani and F. Zuleger. The Reachability-Bound Problem. page 292. ACM Press, 2010. doi:10.1145/1806596.1806630.

[Hal79] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme*. PhD thesis, Grenoble INP, 1979.

[Hal93] N. Halbwachs. Delay Analysis in Synchronous Programs. In *Computer Aided Verification*, volume 697, pages 333–346. Springer Berlin Heidelberg, Berlin, Heidelberg, 1993.

[Hal10] N. Halbwachs. Linear Relation Analysis Principles and Recent Progress, 2010.

[HC11] W. Haddad and V. Chellaboina. *Nonlinear Dynamical Systems and Control: A Lyapunov-Based Approach*. Princeton University Press, 2011. http://books.google.fr/books?id=bUQN6Ph7YEIC.

[Hen11] J. Henry. Static Analysis by Abstract Interpretation, Path Focusing, August 2011.

[HH12] N. Halbwachs and J. Henry. When the Decreasing Sequence Fails. In *Static Analysis*, volume 7460, pages 198–213. Springer Berlin Heidelberg, Berlin, Heidelberg, 2012.

[Hig02] N. J. Higham. *Accuracy and Stability of Numerical Algorithms*. SIAM, 2002.

[HJMS02] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy Abstraction. pages 58–70. ACM Press, 2002. doi:10.1145/503272.503279.

[HLR94] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous Observers and the Verification of Reactive Systems. In *Algebraic Methodology and Software Technology (AMAST'93)*, pages 83–96. Springer London, London, 1994.

[HMM12a] J. Henry, D. Monniaux, and M. Moy. PAGAI: A Path Sensitive Static Analyser. *Electronic Notes in Theoretical Computer Science*, 289:15–25, December 2012. doi:10.1016/j.entcs.2012.11.003.

[HMM12b] J. Henry, D. Monniaux, and M. Moy. Succinct Representations for Abstract Interpretation. In *Static Analysis*, volume 7460 of *Lecture Notes in Computer Science*, pages 283–299. Springer Berlin Heidelberg, 2012.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic Model Checking for Real-Time Systems. pages 394–406. IEEE Comput. Soc. Press, 1992. doi:10.1109/LICS.1992.185551.

[Hoa69] C. A. R. Hoare. An Axiomatic Basis for Computer Programming. *Communications of the ACM*, 12(10):576–580, October 1969. doi:10.1145/363235.363259.

[HPR97] N. Halbwachs, Y.-E. Proy, and P. Roumanoff. Verification of Real-Time Systems using Linear Relation Analysis. *Formal Methods in System Design*, 11(2):157–185, August 1997. doi:10.1023/A:1008678014487.

[IEE08] IEEE Computer Society. IEEE Standard for Floating-Point Arithmetic. Technical report, August 2008.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical Interprocedural Parallelization: An Overview of the PIPS Project. pages 244–251. ACM Press, 1991. doi:10.1145/109025.109086.

[INR15] INRIA. The Coq Proof Assistant, 1989–2015. http://coq.inria.fr/.

[Jea00] B. Jeannet. *Partitionnement dynamique dans l'analyse de relation linéaire et application à la vérification de programmes synchrones*. PhD thesis, Grenoble INP, 2000.

[Jea03] B. Jeannet. Dynamic Partitioning in Linear Relation Analysis: Application to the Verification of Reactive Systems. *Formal Methods in System Design*, 23(1):5–37, July 2003. doi:10.1023/A:1024480913162.

[Jea10] B. Jeannet. The NBAC Verification/Slicing Tool, 2010. http://pop-art.inrialpes.fr/~bjeannet/nbac/index.html.

[Joh98] L. A. Johnson. DO-178B, Software Considerations in Airborne Systems and Equipment Certification. *Crosstalk*, 1998.

[Jor06] P. Jordan. Standard IEC 62304 – Medical Device Software – Software Lifecycle Processes. In *Software for Medical Devices, 2006. The Institution of Engineering and Technology Seminar on*, pages 41–47. IET, 2006.

[Kar76] M. Karr. Affine Relationships among Variables of a Program. *Acta Informatica*, 6(2):133–151, 1976. doi:10.1007/BF00268497.

[KEW+85] R. H. Katz, S. J. Eggers, D. A. Wood, C. L. Perkins, and R. G. Sheldon. Implementing a Cache Consistency Protocol. *ACM SIGARCH Computer Architecture News*, 13(3):276–283, June 1985. doi:10.1145/327070.327237.

[Kha13] D. Khaldi. *Parallélisation automatique et statique de tâches sous contraintes de ressources : une approche générique*. PhD thesis, Centre de Recherche en Informatique, MINES ParisTech, November 2013. http://pastel.archives-ouvertes.fr/pastel-00935483.

[KTDA12] M. Kucharski, A. Trujillo, C. Dunlop, and B. Ahdab. ISO 26262 Software Compliance: Achieving Functional Safety in the Automotive Industry. Technical report, 2012.

[Lab06] Laboratoire Spécification et Vérification, ÉNS Cachan. FAST: Fast Acceleration of Symbolic Transition systems, 2003–2006. http://www.lsv.ens-cachan.fr/Software/fast/.

[Lam77] L. Lamport. Proving the Correctness of Multiprocess Programs. *IEEE Transactions on Software Engineering*, SE-3(2):125–143, March 1977. doi:10.1109/TSE.1977.229904.

[Lam80] L. Lamport. "Sometime" is Sometimes "Not Never": On the Temporal Logic of Programs. pages 174–185. ACM Press, 1980. doi:10.1145/567446.567463.

[Lee02] C. S. Lee. Program Termination Analysis in Polynomial Time. In *Generative Programming and Component Engineering*, volume 2487, pages 218–235. Springer Berlin Heidelberg, Berlin, Heidelberg, 2002.

[LMC02] V. Loechner, B. Meister, and P. Clauss. Precise Data Locality Optimization of Nested Loops. *The Journal of Supercomputing*, 21:37–76, 2002.

[LV94] H. Le Verge. A Note on Chernikova's Algorithm. Technical report, 1994.

[Lya92] A. M. Lyapunov. *The General Problem of the Stability of Motion*. Control Theory and Applications Series. Taylor & Francis, 1992.

[Mai12] V. Maisonneuve. Convex Invariant Refinement by Control Node Splitting: A Heuristic Approach. *Electronic Notes in Theoretical Computer Science*, 288:49–59, December 2012. doi:10.1016/j.entcs.2012.10.007.

[Mas14] D. Massé. Policy Iteration-Based Conditional Termination and Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*, volume 8318 of *Lecture Notes in Computer Science*, pages 453–471. Springer Berlin Heidelberg, 2014.

[Mer92] N. Mercouroff. An Algorithm for Analyzing Communicating Processes. In *Mathematical Foundations of Programming Semantics*, volume 598, pages 312–325. Springer Berlin Heidelberg, Berlin, Heidelberg, 1992.

[Mer05] D. Merchat. *Réduction du nombre de variables en analyse de relations linéaires*. PhD thesis, Université Joseph-Fourier - Grenoble 1, 2005.

[Min06] A. Miné. The Octagon Abstract Domain. *Higher-Order and Symbolic Computation*, 19(1):31–100, March 2006. doi:10.1007/s10990-006-8609-1.

[MIS98] MISRA. *Guidelines for the Use of the C Language in Vehicle Based Software*. 1998.

[MRTT53] T. Motzkin, H. Raiffa, G. L. Thompson, and R. M. Thrall. The Double Description Method. In *Contributions to the Theory of Games II*. Princeton University Press, 1953.

[Ner13] P. Neron. *A Quest for Exactness: Program Transformation for Reliable Real Numbers*. PhD thesis, Ecole Polytechnique X, October 2013. http://tel.archives-ouvertes.fr/tel-00924379.

[NI05] T. V. N. Nguyen and F. Irigoin. Efficient and Effective Array Bound Checking. *ACM Transactions on Programming Languages and Systems*, 27(3):527–570, May 2005. doi:10.1145/1065887.1065893.

[NIAK01] N. T. V. Nguyen, F. Irigoin, C. Ancourt, and R. Keryell. Efficient Intraprocedural Array Bound Checking. In *In Second International Workshop on Automated Program Analysis, Testing and Verification*, 2001.

[NR00] S. P. K. Nookala and T. Risset. A Library for Z-Polyhedral Operations. Technical report, IRISA, May 2000.

[PC07] C. Popeea and W.-N. Chin. Inferring Disjunctive Postconditions. In *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues*, volume 4435, pages 331–345. Springer Berlin Heidelberg, Berlin, Heidelberg, 2007.

[Pel01] D. Peled. *Software Reliability Methods*. Springer, 2001.

[PNW14] L. Paulson, T. Nipkow, and M. Wenzel. The Isabelle theorem prover, 2002–2014. http://isabelle.in.tum.de/.

[PR04] A. Podelski and A. Rybalchenko. A Complete Method for the Synthesis of Linear Ranking Functions. In *Verification, Model Checking, and Abstract Interpretation*, volume 2937, pages 239–251. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[PRK+14] W. Pugh, E. Rosser, W. Kelly, D. Wonnacot, T. Shpeisman, and V. Maslov. The Omega Project, 1995–2014. http://www.cs.umd.edu/projects/omega/.

[PSS02] F. Pelletier, G. Sutcliffe, and C. Suttner. The Development of CASC. *AI Communications*, 15(2-3):79–90, 2002.

[QS82] J. P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in CESAR. In *International Symposium on Programming*, volume 137, pages 337–351. Springer Berlin Heidelberg, Berlin, Heidelberg, 1982.

[RTC92] RTCA. *Software Considerations in Airborne Systems and Equipment Certification*. 1992.

[Sch86] A. Schrijver. *Theory of Linear and Integer Programming*. John Wiley & Sons, Inc., New York, NY, USA, 1986.

[SS06] G. Sutcliffe and C. Suttner. The State of CASC. *AI Communications*, 19(1):35–48, 2006.

[Sut09] G. Sutcliffe. The TPTP Problem Library and Associated Infrastructure: The FOF and CNF Parts, v3.5.0. *Journal of Automated Reasoning*, 43(4):337–362, December 2009. doi:10.1007/s10817-009-9143-8.

[VCB11] S. Verdoolaege, A. Cohen, and A. Beletska. Transitive Closures of Affine Integer Tuple Relations and Their Overapproximations. In *Proceedings of the 18th International Conference on Static Analysis*, SAS'11, pages 216–232, Berlin, Heidelberg, 2011. Springer-Verlag.

[Ver10] S. Verdoolaege. isl: An Integer Set Library for the Polyhedral Model. In *Mathematical Software – ICMS 2010*, volume 6327, pages 299–302. Springer Berlin Heidelberg, Berlin, Heidelberg, 2010.

[VSB+07] S. Verdoolaege, R. Seghir, K. Beyls, V. Loechner, and M. Bruynooghe. Counting Integer Points in Parametric Polytopes Using Barvinok's Rational Functions. *Algorithmica*, 48(1):37–66, May 2007. doi:10.1007/s00453-006-1231-0.

[VWBC05] S. Verdoolaege, K. M. Woods, M. Bruynooghe, and R. Cools. Computation and Manipulation of Enumerators of Integer Projections of Parametric Polytopes. Technical Report CW\,392, Katholieke Universiteit Leuven, Department of Computer Science, March 2005.

[WDD+12] V. Wiels, R. Delmas, D. Doose, P.-L. Garoche, J. Cazin, and G. Durrieu. Formal Verification of Critical Aerospace Software. *AerospaceLab Journal*, 2012.

[Wil93] D. K. Wilde. A Library for Doing Polyhedral Operations. Technical report, 1993.

[Wol14] Wolfram Research. Wolfram Mathematica, 1988–2014. http://www.wolfram.com/mathematica/.

[Won01] D. Wonnacott. Extending Scalar Optimizations for Arrays. In *Languages and Compilers for Parallel Computing*, volume 2017 of *Lecture Notes in Computer Science*, pages 97–111. Springer Berlin Heidelberg, 2001.

[Yos13] J. Yoshida. Toyota Case: Single Bit Flip That Killed. *EE Times*, 2013.

List of Figures

List of Tables

Listings

Analyse statique des systèmes de contrôle-commande : invariants entiers et flottants

Résumé : Un logiciel critique est un logiciel dont le bon fonctionnement a un impact important sur la sécurité ou la vie des personnes, des entreprises ou des biens. L'ingénierie logicielle pour les systèmes critiques doit donc répondre à des exigences drastiques de qualité, souvent réglementaires, permettant d'avoir un degré de confiance très élevé dans les programmes. C'est une tâche particulièrement difficile, qui combine différentes méthodes pour garantir la qualité des logiciels produits. Parmi celles-ci, les méthodes formelles peuvent être utilisées pour prouver qu'un logiciel respecte ses spécifications. Le travail décrit dans ce manuscrit s'inscrit dans le contexte de la validation de propriétés de sûreté de programmes critiques, et plus particulièrement des propriétés numériques de logiciels embarqués dans des systèmes de contrôle-commande.

La première partie de cette thèse est consacrée aux preuves de stabilité au sens de Lyapunov. Ces preuves raisonnent sur des calculs en nombres réels ; or, ceci ne décrit pas correctement le comportement d'un programme exécuté sur une plateforme à arithmétique machine. Nous présentons un cadre théorique générique pour adapter les arguments des preuves de stabilité de Lyapunov aux arithmétiques machine. Un outil effectue automatiquement la transposition de ces preuves en nombres réels vers des preuves en nombres à virgule flottante.

La seconde partie du manuscrit porte sur l'analyse des relations affines, une interprétation abstraite qui approxime les valuations associées aux points de contrôle d'un programme par des polyèdres convexes. Nous présentons ALICe, un framework permettant de comparer différentes techniques de génération d'invariants. Il s'accompagne d'un benchmark composé d'une collection de cas de test tirés de publications sur l'analyse de programmes, et s'interface avec trois outils utilisant différents algorithmes de calcul d'invariants : Aspic, iscc et PIPS. Pour tester la sensibilité des outils à l'encodage, deux restructurations conservatives des cas de test sont proposées. Afin d'affiner les invariants générés par PIPS, nous introduisons dans un dernier chapitre plusieurs améliorations à ses algorithmes.

Mots clés : analyse statique, analyse des relations linéaires, calcul en virgule flottante, sécurité et sûreté, stabilité de Lyapunov

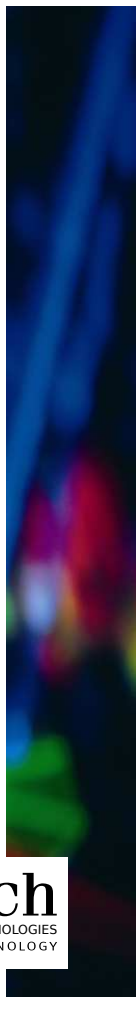
Static Analysis of Control-Command Systems: Floating-Point and Integer Invariants

Abstract: A critical software is a software whose proper functioning has an important impact on safety or life of people, companies and goods. Software engineering on critical systems must comply with drastic quality requirements, often regulatory, to achieve a higher level of trust in programs. This task is particularly difficult and combines different methods to ensure the quality of produced software. Among them, formal methods can be used to prove that a software obeys its specifications. The context of the work described in this manuscript is the validation of safety properties for critical software; specifically, of numerical properties for embedded software in control-command systems.

The first part of this thesis is dedicated to Lyapunov stability proofs. These proofs reason with exact real-number computations, but this is not an accurate description of how a program behaves on a machine-arithmetic platform. We present a generic, theoretical framework to adapt Lyapunov stability proof arguments to machine arithmetics. A tool automatically transposes these proofs on real numbers to proofs on floating-point numbers.

The second part of the manuscript is about linear relation analysis, an abstract interpretation that approximates valuations associated to a program's control points by convex polyhedrons. We present ALICe, a framework to compare different invariant generation techniques. It comes with a benchmark consisting of a collection of test cases drawn from the program analysis literature, and interfaces with three tools relying on different invariant computation algorithms: Aspic, iscc and PIPS. To evaluate the sensitivity of these tools to encoding choices, two techniques are proposed to conservatively restructure test cases. To refine invariants generated by PIPS, several improvements are made on its algorithms and introduced in the last chapter.

Keywords: Static Analysis, Linear Relation Analysis, Floating-Point Computation, Security and Safety, Lyapunov Stability



MINES ParisTech

ParisTech
INSTITUT DES SCIENCES ET TECHNOLOGIES
PARIS INSTITUTE OF TECHNOLOGY