Réduction du nombre de variables en analyse de relations linéaires

David Merchat

UNIVERSITE JOSEPH FOURIER - GRENOBLE I
SCIENCES ET GEOGRAPHIE

THESE

pour obtenir le grade de

DOCTEUR DE L'UNIVERSITE JOSEPH FOURIER

Discipline : Informatique

présentée et soutenue publiquement par

Merchat David

Le 18 05 2005

Titre :

Réduction du nombre de variables en analyse de relations linéaires

Directeurs de thèse :

Halbwachs Nicolas et Parent Catherine

COMPOSITION DU JURY :

Présidente	**Borrione Dominique**
Directeurs	**Halbwachs Nicolas**
	Parent Catherine
Rapporteurs	**De Simone Robert**
	Irigoin François
Examinateur	**Jeannet Bertrand**

Table des matières

Chapitre 1

Introduction

La vérification de programmes est un enjeu dont l'importance n'est plus à souligner. Depuis les travaux fondateurs de Floyd et Hoare, il y a près de quarante ans, les recherches dans ce domaine se sont développées dans diverses directions, mais principalement basées, d'une part, sur la preuve formelle — nécessairement interactive —, et d'autre part sur l'exploration exhaustive de modèles finis des programmes ("model-checking").

Les logiciels les plus critiques, du point de vue de la sûreté, sont les logiciels embarqués temps réel. Dans ce domaine, comme dans beaucoup d'autres, la complexité croissante des applications rend irréaliste l'application des méthodes de preuve interactive. Par ailleurs, même si l'on se restreint aux noyaux de contrôle de tels systèmes, il est rare que l'on puisse en extraire automatiquement un modèle fini suffisamment précis : la raison en est que même le contrôle fait appel à des variables numériques (par exemple, des compteurs de temps, d'événements, ...) dont le comportement est significatif dans le respect des propriétés cruciales du système. C'est pourquoi nous nous intéressons ici à la vérification *automatique* de programmes manipulant des variables numériques, et donc de programmes *à nombre infini d'états*.

Cet objectif se heurte à l'indécidabilité de la vérification de programmes généraux. On est ainsi amené, soit à se restreindre à certaines classes de programmes et de propriétés dont la vérification est décidable — mais les applications réelles appartiennent rarement à ces classes —, soit à se contenter de vérification approchée : le résultat fourni sera soit "*la propriété est satisfaite*", soit "*je ne sais pas*". Dans ce domaine, les principales approches relèvent de l'interprétation abstraite, une théorie générale proposée dès 1977 [CC77], théorie qui a été très tôt instanciée pour l'analyse de propriétés numériques [CC76, CH78, Hal79a, Gra89]. Parmi ces analyses, l'*analyse de relations linéaires* [CH78, Hal79a] consiste à calculer, en chaque point de contrôle d'un programme impératif, un système de contraintes linéaires (égalités et inégalités) toujours satisfaites par les variables du programme lorsque le contrôle est en ce point.

Quoique ancienne, cette analyse continue d'être utilisée [LMQ91, ACIK97, HPR97]. Les invariants linéaires construits par l'analyse sont utilisés, en vérification, pour montrer l'inaccessibilité de certains points de contrôle (ce qui est suffisant pour vérifier toutes les propriétés de sûreté, cf. §2.3.5). Mais ces invariants ne sont pas utilisés seulement

en vérification : en compilation, ils peuvent servir à déterminer statiquement l'absence d'erreur à l'exécution (indices de tableaux, par exemple) et à optimiser le code produit ; en parallélisation automatique [IJT91], ils permettent d'analyser finement les dépendances entre éléments de tableaux ; en preuve formelle [BBC+00, BLO98], ils constituent une aide précieuse dans la recherche d'invariants inductifs.

L'analyse de relations linéaires est aussi encore l'objet d'études et d'améliorations [JHR99, BRZH02, BHRZ03, SSM05], concernant les performances et/ou la précision de l'analyse. Notre travail vise à en améliorer les performances sans perte de précision.

L'analyse de relations linéaires consiste essentiellement à propager des systèmes de contraintes linéaires le long des chemins d'exécution d'un programme. Cette propagation fait donc appel à une algorithmique des systèmes linéaires, ou, de manière équivalente, des *polyèdres convexes*. Les algorithmes clés concernent l'intersection des polyèdres, leur *enveloppe convexe* (i.e., le plus petit polyèdre convexe contenant deux ou plusieurs polyèdres, et qui est utilisé comme approximation de leur union), et la minimisation de leur représentation. Dans la plupart des implémentations, ces opérations sont effectuées grâce à la *double description* des polyèdres : un polyèdre peut être représenté par un système de contraintes linéaires, mais aussi par un *système générateur*, composé de ses sommets et de ses rayons extrêmes. Selon l'opération à effectuer, l'une ou l'autre des représentations est préférable. Par ailleurs, la connaissance des deux représentations permet de les minimiser.

Un problème sérieux posé par la double description est que la taille de chacune des représentations peut croître exponentiellement par rapport à celle de l'autre, *quand la dimension de l'espace croît*. Par exemple, un hypercube en dimension n, défini par les contraintes $\{0 \leq x_i \leq 1, i = 1..n\}$, a $2n$ contraintes, mais 2^n sommets. En conséquence, la dimension de l'espace, c'est-à-dire le nombre des variables du programme, a une influence très importante sur les performances de l'analyse.

Un autre problème, plus technique, posé par le nombre des variables, concerne la représentation des nombres : pour éviter les problèmes d'imprécision dus aux calculs en nombres flottants, les bibliothèques de manipulation de polyèdres opèrent généralement sur les rationnels. Pour éviter alors le doublement de la taille mémoire, les coefficients d'une même contrainte, comme les composantes d'un même vecteur, sont réduits au même dénominateur qui n'est stocké qu'une fois. Mais lorsque le nombre des variables augmente, ce dénominateur doit être commun à un nombre croissant de coefficients, ce qui entraîne une explosion en taille des nombres manipulés.

L'objectif de ce travail est de proposer des techniques pour réduire le nombre de variables dans les manipulations des polyèdres. Il existe pour cela des techniques bien connues en analyse de programmes :

– La détermination précise des zones de vie ("life range" [Muc97]) des variables : en un point de contrôle du programme à partir duquel sur tout chemin d'exécution, une variable x sera affectée (d'une expression indépendant de x) avant d'être utilisée, ou alors ne sera plus jamais utilisée, la valeur de x n'a pas d'importance pour l'exécution du programme. On peut donc "oublier" la variable x en ce point.

– Le tranchage ("slicing" [Tip95]) : Il s'agit de déterminer les éléments (instructions, variables) du programme qui peuvent influencer la propriété que l'on cherche à prouver (généralement l'inaccessibilité d'un ou plusieurs points de contrôle). Le tranchage consiste à ne garder du programme que les fragments pertinents vis-à-vis de la propriété à prouver.

Ces solutions consistent à tenir compte du programme analysé, et doivent évidemment être appliquées avant toute analyse risquant d'être coûteuse. Mais nous allons nous intéresser à d'autres techniques de réduction du nombre des variables, qui, elles, ne dépendront plus du programme analysé (à l'exception de l'une d'entre-elles), mais s'appliqueront au niveau des opérations sur les polyèdres.

Une première idée consiste à utiliser des *équations affines* satisfaites par les polyèdres pour "tirer" une variable de chaque équation. C'est l'approche décrite au Chapitre 6. Pour connaître à l'avance des équations affines invariantes, et les utiliser ainsi, une première solution est d'appliquer la méthode proposée par Karr [Kar76], qui déduit ces équations d'une analyse préalable du programme. Cette approche s'avérant peu satisfaisante du fait du faible nombre d'équations invariantes trouvées, nous utiliserons les opérateurs de Karr pour prévoir pas-à-pas les équations satisfaites par le résultat d'une opération : l'avantage est que l'on peut utiliser des équations satisfaites temporairement, à une étape de l'analyse, et qui sont bien plus nombreuses que les équations réellement invariantes.

Une deuxième idée, plus prometteuse en termes de gain de performances, est de détecter quand un polyèdre est en fait un *produit cartésien* de polyèdres de dimensions plus petites, c'est-à-dire quand les variables peuvent être partitionnées en plusieurs classes indépendantes (i.e., les contraintes du polyèdre ne liant pas des variables de classes différentes). L'exemple des hypercubes mentionné ci-dessus suggère que le traitement séparé des termes d'un produit cartésien peut conduire à une réduction logarithmique de la taille des systèmes générateurs : un hypercube en dimension n peut être vu comme le produit cartésien de n intervalles en dimension 1, et ses 2^n sommets se déduisent des $2n$ sommets (extrémités) des intervalles. Cette idée est développée au chapitre 7. Les problèmes traités concernent la détection des produits cartésiens, et l'adaptation des opérations pour opérer sur des produits cartésiens.

La factorisation en produits cartésien est évidemment fortement dépendante du choix des variables dans le programme source : un hypercube sera bien factorisé dans une base adéquate, mais un simple changement de variables peut faire disparaître la factorisation. Au chapitre 8, nous proposerons un algorithme simple pour trouver la base la plus favorable, et diminuer ainsi la dépendance entre les performances de l'analyse et le choix des variables du programme.

Ce travail repose sur un prérequis important qu'il nous faut d'abord rappeler. Le chapitre 2 a pour but de fixer les notions et notations relatives aux programmes — que nous considérerons simplement comme des automates interprétés —, à leur sémantique, leurs propriétés et les méthodes de vérification. Le chapitre 3 est un rappel des principes de l'interprétation abstraite, en mettant l'accent sur les applications concernant les propriétés numériques. Le chapitre 4 expose de façon détaillée les algorithmes classiques sur

les polyèdres, nécessaires à la mise en oeuvre de l'analyse de relations linéaires.

Pour évaluer expérimentalement l'effet des techniques que nous proposons, nous avons du développer un nouvel analyseur de programmes qui est décrit au chapitre 5. En effet, l'analyseur de programmes LUSTRE existant précédemment (l'outil NBAC de Bertrand Jeannet [Jea00]) est trop sophistiqué — il prend en charge, en particulier, la découverte d'une structure de contrôle adéquate, en fonction de la propriété à vérifier — pour permettre d'étudier simplement les performances des opérations sur les polyèdres, la part des ces opérations dans le coût total de l'analyse étant trop faible. Nous avons donc programmé un prototype simple qui s'appuie sur la structure de contrôle de l'automate produit par les compilateurs des langages synchrones LUSTRE et ESTEREL.

L'implémentation de nos nouvelles techniques de manipulation des polyèdres est décrite au chapitre 9, ainsi que les expérimentations que nous avons menées. Ces expérimentations donnent des résultats très décevants concernant l'utilisation des équations affines, mais bien plus intéressants pour la factorisation cartésienne et le changement de base. La conclusion générale est donc que ces deux dernières techniques méritent d'être intégrées aux bibliothèques de polyèdres existantes. Nous conclurons par quelques perspectives.

Chapitre 2

Modèles, Programmes, Vérification

Dans ce chapitre, nous allons fixer les principales définitions et notations liées aux programmes, à leur sémantique, et aux propriétés que l'on veut vérifier à leur propos. Nous introduirons d'abord les systèmes de transitions (§2.1), qui constituent le modèle sémantique classique des programmes, puis nous définirons les propriétés (§2.2) de ces systèmes, et la problématique générale de la vérification. Au §2.3, nous introduirons les automates interprétés, choisis comme modèle de programmes très général.

2.1 Modèle sémantique : Les systèmes de transitions

Les systèmes de transitions constituent le modèle sémantique classique des programmes. Nous fixons ici les définitions et notations relatives aux systèmes de transitions.

Définition 2.1 (Système de transitions)
Un système de transitions est un triplet $(Q, \rightarrow, Q_{init})$ *où* Q *est un ensemble d'états,* $Q_{init} \subseteq Q$ *est un sous-ensemble d'états appelés "états initiaux", et* $\rightarrow \subseteq Q \times Q$ *est une relation binaire sur* Q*, appelée "relation de transition".*

On notera généralement $q \rightarrow q'$ le fait que $(q, q') \in \rightarrow$.

Définition 2.2 (Etat successeur, état prédécesseur)
Un état q' *(respectivement* q*) est* successeur *(respectivement* prédécesseur*) d'un état* q *(respectivement* q'*) si* $q \rightarrow q'$.

Définition 2.3 (Chemin, trace)
Un chemin *de l'état* q *à l'état* q' *est une suite* $(q_0, ..., q_n)$ *telle que* $q = q_0$, $q' = q_n$ *et* $\forall i < n, q_i \rightarrow q_{i+1}$. *Une* trace *est un chemin qui a pour origine un état de* Q_{init}.

Définition 2.4 (Etat accessible depuis q, co-accessible depuis q')
Un état q' *(resp.* q*) est* accessible *(resp.* co-accessible*) depuis un état* q *(resp.* q'*) si il existe un chemin entre* q *et* q' *(i.e., si* q *et* q' *sont reliés par la fermeture transitive de* $\rightarrow$*).*

2.2 Vérification

Nous nous plaçons dans le cadre de la vérification *de propriétés* (par opposition, par exemple, à la comparaison de programmes [BRdSV89, Fer90]). Plus spécifiquement, nous considérerons une sémantique linéaire (par opposition, par exemple, aux sémantiques arborescentes [Lam80]) : un comportement d'un système de transitions est une trace, une propriété est un ensemble de traces. Enfin, nous nous intéresserons essentiellement aux propriétés de *sûreté*.

2.2.1 Propriétés

Nous considérerons donc qu'une propriété d'un système de transitions est un ensemble de traces. Dans ce paragraphe, nous supposerons toutes ces traces infinies. Soit Q^ω l'ensemble des suites infinies d'états de Q. Il est classique [Lam77, AS87] de distinguer deux grands types de propriétés, principalement parce que leur vérification fait appel à des techniques différentes : informellement,

- une propriété de *sûreté* exprime que "quelque chose de mauvais" ne se produit jamais au cours de l'exécution ;
- une propriété de *vivacité* exprime que "quelque chose de bon" se produit immanquablement durant l'exécution du système.

La violation d'une propriété de sûreté est détectable par une exécution finie. En effet, si une trace $\sigma = (q_0, q_1, \ldots)$ viole la propriété, alors il existe un état q_i de σ où "quelque chose de mauvais" se produit et cela est irrémédiable car la propriété suppose que cela n'arrive jamais. Donc toute trace ayant $(q_0, q_1, \ldots, q_i)$ (noté $\sigma[\ldots i]$) comme préfixe, viole la propriété. Une trace satisfait la propriété de sûreté P si et seulement si tous ses préfixes finis vérifient P, ou, plus exactement, si tous ses préfixes finis sont prolongeables en une trace vérifiant P :[1]

Définition 2.5 (Propriété de sûreté)
$P \subseteq Q^\omega$ est une propriété de sûreté si et seulement si

$$\sigma \in P \iff \forall i \geq 0, \exists \sigma' \in Q^\omega \text{ telle que } \sigma[\ldots i].\sigma' \in P$$

où $\sigma[\ldots i]$ dénote la suite finie des i premiers termes de σ, et ou "." dénote la concaténation des suites.

Des exemples classiques de propriétés de sûreté sont l'exclusion mutuelle ou l'absence de bloquage.

Si P est une propriété de vivacité, aucune exécution finie n'est irrémédiablement mauvaise, puisque "quelque chose de bon" peut toujours survenir dans le futur. Toute exécution finie est prolongeable en une trace satisfaisant P :

[1] C'est pourquoi, dans la suite, on parlera souvent de la satisfaction d'une propriété de sûreté par une trace *finie*.

Définition 2.6 (Propriété de vivacité)
$P \subseteq Q^\omega$ est une propriété de vivacité si et seulement si

$$\forall \sigma \in Q^*, \exists \sigma' \in Q^\omega, \text{ telle que } \sigma[\dots i].\sigma' \in P$$

Un exemple classique de propriété de vivacité est la terminaison de programme.

Dans la suite, on ne s'intéresse qu'à la vérification de propriétés de sûreté. Nous verrons au §2.3.5 que ces propriétés se ramènent au cas, plus simple, des propriétés d'accessibilité.

2.2.2 Accessibilité et Vérification

Nous avons vu ci-dessus qu'une propriété de sûreté exprime que "quelque chose de mauvais ne se produit jamais". Les propriétés d'accessibilité constituent un sous-ensemble, où l'on exprime qu'un certain ensemble d'états — représentant "quelque chose de mauvais", et que nous noterons Q_{err} — ne peuvent pas être atteints. Leur vérification fait appel au calcul des états accessibles (depuis les états initiaux Q_{init}) ou coaccessibles (depuis les états d'erreur Q_{err}).

On introduit quelques définitions classiques relatives à l'accessibilité. Soit X un sous-ensemble de Q.

Définition 2.7 (Postconditions, Préconditions)
On note :
- $Post(X) = \{q' \in Q | \exists q \in X, q \to q'\}$,
- $Pre(X) = \{q \in Q | \exists q' \in X, q \to q'\}$,

$Post(X)$ (resp. $Pre(X)$) est l'ensemble des états successeurs (resp. prédécesseurs) d'un état de X.

Définition 2.8 (Etats accessibles, états co-accessibles)
On note $acc(X)$ (resp. $coacc(X)$) l'ensemble de tous les états accessibles (resp. co-accessibles) depuis un état de X. Il est bien connu que $acc(X)$ (resp. $coacc(X)$) est la plus petite solution de l'équation de point fixe $Y = X \cup Post(Y)$ (resp. $Y = X \cup Pre(Y)$). On a donc :

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

A l'évidence, la vérification d'une propriété d'accessibilité peut être effectuée de deux manières :
- en calculant $acc(Q_{init})$, et en vérifiant que $acc(Q_{init}) \cap Q_{err} = \emptyset$
- en calculant $coacc(Q_{err})$, et en vérifiant que $Q_{init} \cap coacc(Q_{err}) = \emptyset$

Le coeur du problème de la vérification des propriétés d'accessiblité est donc le calcul de points-fixes. Dans certains cas, ce calcul peut-être effectué exactement : on parle, dans ce cas, de "*model-checking*" [QS82, CES86]. Mais dans le cas général, on ne sait calculer automatiquement que des *approximations* des points-fixes ; on parle alors de *vérification conservative*.

2.2.3 Model checking

Le model checking concerne essentiellement les systèmes *à nombre fini d'états*. Dans ce cas, il est clair que les calculs itératifs de points-fixes convergent toujours, et que toutes les propriétés sont, théoriquement, décidables. Les travaux sur le model checking ont concerné d'une part la vérification de propriétés très générales — aussi bien de sûreté que de vivacité, avec une sémantique linéaire aussi bien qu'arborescente — et d'autre part, les techniques permettant de prendre en compte de très grand ensembles d'états. La vérification étant exacte, sa conclusion consiste soit en un verdict de satisfaction de la propriété, soit en un verdict d'erreur qui peut alors être accompagné d'un contre-exemple.

Par ailleurs, le model checking est aussi utilisé dans le cas de systèmes d'états infinis, lorsqu'on sait abstraire ceux-ci (en ignorant certains aspects) en des systèmes finis [GL93, CGL94]. On parle dans ce cas de model checking *abstrait*. Notons qu'en cas d'échec, ce type de vérification ne peut conclure à la non-satisfaction de la propriété, puisque l'échec peut être du aux abstractions effectuées.

2.2.4 Vérification conservative

Si tous les types de propriétés nécessitent des calculs de points-fixes, dans le cas des propriétés d'accessiblité, on peut se contenter de calculs approchés : si l'on calcule un sur-ensemble de $acc(Q_{init})$ (resp., de $coacc(Q_{err})$) et que celui-ci n'intersecte pas Q_{err} (resp. Q_{init}), la propriété est certainement satisfaite. Par contre, si l'intersection est non vide, on ne peut savoir si la propriété est violée ou si la sur-approximation utilisée est trop grossière. Inversement, si l'on calcule un sous-ensemble de $acc(Q_{init})$ (resp., de $coacc(Q_{err})$) et que celui-ci intersecte Q_{err} (resp. Q_{init}), la propriété est certainement violée. Dans les deux cas, on parlera de vérification *conservative* : en cas de succès, la vérification fournit un résultat garanti, mais en cas d'échec, la vérification est inconclusive.

L'approximation de points-fixes est le thème central de la théorie de l'*interprétation abstraite*, que nous rappellerons au chapitre 3.

2.3 Modèles de programmes

Les systèmes de transitions constituent un modèle de très bas niveau. Nous aurons besoin d'un modèle de programme de plus haut niveau, mais, pour ne pas nous baser sur un langage de programmation particulier, nous choisirons un modèle relativement

abstrait et général : les automates interprétés. Ceux-ci peuvent être la cible de traducteurs de langages particuliers (voir §5.2.2). Nous donnerons au §2.3.2 une sémantique des automates interprétés en termes de systèmes de transitions. C'est également au niveau des automates interprétés que nous définirons la traduction d'une propriété de sûreté en propriété d'accessibilité (§2.3.5).

2.3.1 Automates interprétés

Un automate interprété sera constitué d'un ensemble de points de contrôle (dont un point initial), à chacun desquels est associé un ensemble de commandes gardées, agissant sur des variables, et menant à d'autres points de contrôle.

Soit *Var* un ensemble fini de variables typées, *Valeur* l'ensemble des valeurs de ces variables (i.e., l'union des ensembles de valeurs de leurs types). Une *valuation* est une fonction de *Var* dans *Valeur*, associant à chaque variable une valeur de son type. Soit Val_{Var} (ou simplement *Val*) l'ensemble des valuations sur *Var*.

Soit K un ensemble fini, dont les éléments seront appelés *points de contrôle*.

Définition 2.9 (Garde, action, commande gardée)

- *Une* garde *est une fonction* $g : Val \to \{vrai, faux\}$*, donnant la valeur d'une formule logique sur une valuation* V*. Dans la pratique, on assimile la garde à son ensemble de vérité :* $\{V \mid g(V) = vrai\}$*. On note Gardes l'ensemble des gardes.*
- *Une action est une relation binaire sur les valuations (*$\subseteq Val \times Val$*). On note* Act *l'ensemble des actions possibles.*
- *Une* commande *est un quadruplet* $(k, g, a, k') \in K \times Gardes \times \mathrm{Act} \times K$*. Intuitivement, lorsque le contrôle de l'automate est en* k*, et si la valuation courante* V *satisfait la garde* g *(i.e.,* $g(V) = vrai$*), l'exécution de la commande fait passer au point de contrôle* k'*, avec une valuation* V' *telle que* $(V, V') \in a$*.*

Les actions considérées peuvent donc être non déterministes. En particulier, une action de lecture (notée $\mathsf{x} := ?$ ou $\mathsf{lire(x)}$) associe à une valuation V toute valuation V' qui ne diffère de V que par la valeur de x (autrement dit, x prend une valeur quelconque). Certaines variables (variables d'entrée) peuvent être ainsi complètement non spécifiées.

Définition 2.10 (Automate interprété)

Un automate interprété est un triplet (K, Com, k_{init})*, où* K *est un ensemble fini de points de contrôle,* Com *est un ensemble fini de commandes, et* $k_{init} \in K$ *est le point de contrôle initial.*

2.3.2 Sémantique en termes de systèmes de transitions

On donne la sémantique des automates interprétés en termes de systèmes de transitions. Soit (K, Com, k_{init}) un automate interprété. On lui associe un système de transitions $(Q, \to, Q_{init})$ défini comme suit :

- L'ensemble des états est l'ensemble des couples (k, V), où k est un point de contrôle et V une valuation : $Q = K \times Val$.
- Les états initiaux ont k_{init} pour point de contrôle, et n'importe quelle valuation : $Q_{init} = \{k_{init}\} \times Val$.
- La relation de transition obéit à la sémantique des commandes :

$$(k, V) \rightarrow (k', V') \iff \exists (k, g, a, k') \in Com \ \wedge \ g(V) = vrai \ \wedge \ (V, V') \in a$$

2.3.3 Exemple

On va illustrer la modélisation d'un programme par un automate interprété sur un tout petit exemple, tiré de [Hal93]. On veut modéliser une voiture :

1. dont la vitesse est inférieure ou égale à 2 m/s
2. qui s'arrête en 4 secondes
3. et percute un mur après 10 mètres

Il s'agira, bien sûr, plus tard, de montrer que, sous les hypothèses (1) et (2), la voiture s'arrête avant le mur !

Le programme de simulation prend 2 entrées booléennes "mètre" et "seconde" (vraies lorsque l'événement correspondant est perçu ; les événements sont supposés non simultanés). Le programme compte la distance parcourue ("distance", un compteur de "mètre"), le temps écoulé ("temps", un compteur de "seconde"), et la vitesse instantanée ("vitesse", un compteur de "mètre" réinitialisé chaque "seconde") et détermine si la voiture roule trop vite, s'arrête, ou percute le mur.

L'automate correspondant est représenté Fig 2.1. Il se trouve que, ce système étant d'états finis, on peut représenter le système de transitions correspondant (cf. Fig. 2.2). Sur ce système de transitions, il est facile de vérifier que l'état "Mur" est inaccessible.

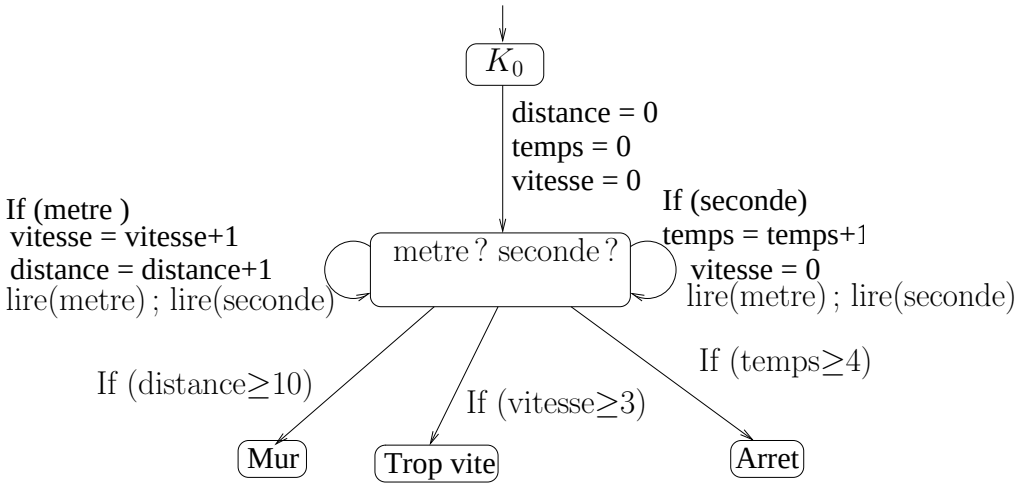


FIG. 2.1 – Automate interprété simulant l'exemple de la voiture

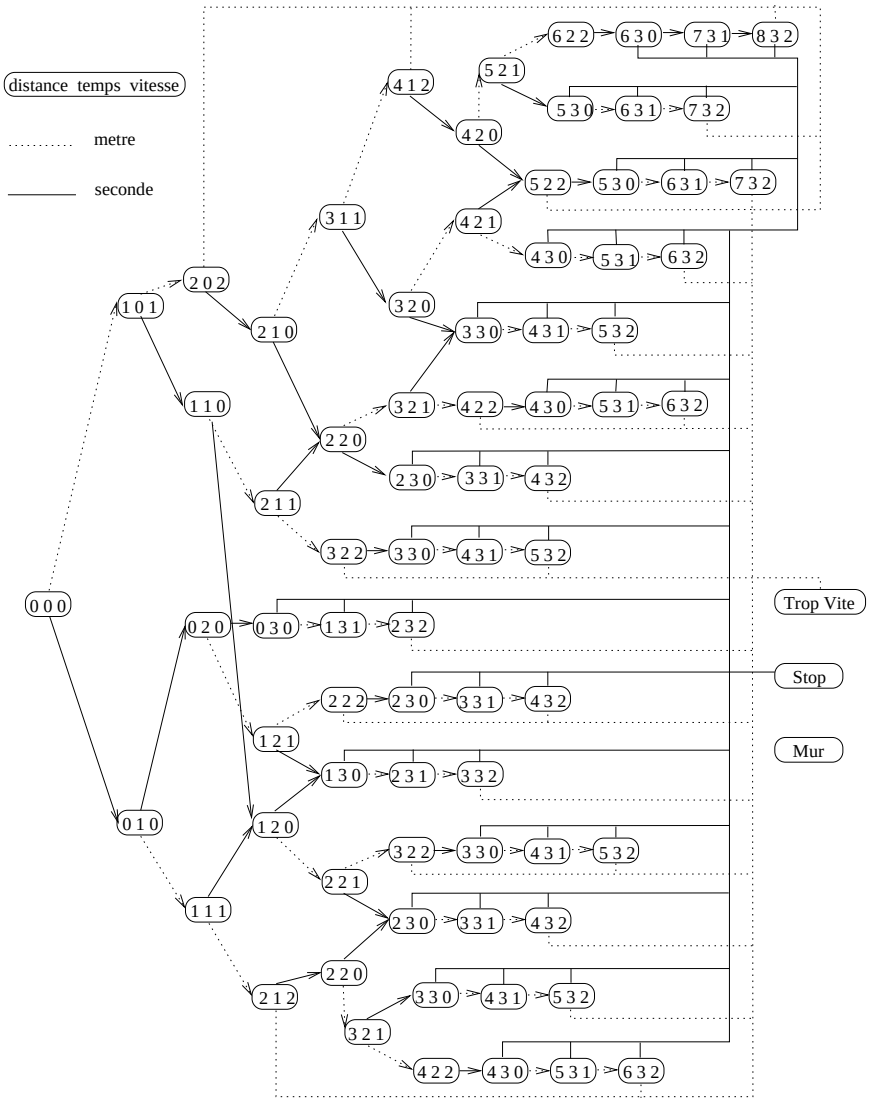


FIG. 2.2 – Système de transitions correspondant à l'exemple de la voiture

Remarques :

- Cet exemple étant d'états finis, le model checking est utilisable.
- Si les constantes apparaissant dans les hypothèses (2 m/s, 4 sec., 10m) augmentent, le nombre d'états du système de transitions tend à exploser, rendant le model checking problématique. Si ces constantes sont des paramètres, donnés symboliquement, l'utilisation du model checking devient impossible.

2.3.4 Une autre sémantique

Pour vérifier une propriété d'accessibilité d'un automate interprété, il va falloir calculer, ou approximer, soit l'ensemble $acc(\{k_{init}\} \times Val)$ des états accessibles du système de transitions associé, soit l'ensemble $coacc(\{k_{err}\} \times Val)$ des états co-accessibles, en supposant que les états d'erreur sont caractérisés par un point de contrôle, k_{err} (dans l'exemple de la voiture, il s'agit du point de contrôle "Mur").

Pour cela, au lieu d'appliquer brutalement la sémantique définie au §2.3.2, on peut chercher à tirer parti de la structure de contrôle de l'automate pour décomposer le problème.

Pour tout $k \in K$, notons A_k (resp., C_k) l'ensemble des valuations accessibles (resp., co-accessibles) *au point* k :

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

On a, évidemment

$$acc(\{k_{init}\} \times Val) = \bigcup_{k \in K} A_k \quad , \quad coacc(\{k_{err}\} \times Val) = \bigcup_{k \in K} C_k$$

Alors, on peut caractériser les A_k (resp., les C_k) comme plus petites solutions d'un *système d'équations sémantiques "en avant"* (resp., *"en arrière"*) défini comme suit :

– Système "en avant" :

$$\forall k \neq k_{init},\ A_k = \bigcup_{<k',g,a,k> \in Com} a(A_{k'} \cap g) \quad , \quad A_{k_{init}} = Val \tag{2.1}$$

où $a(X) = \{V' \mid \exists V \in X, (V, V') \in a\}$

– Système "en arrière" :

$$\forall k \neq k_{err},\ C_k = \bigcup_{<k,g,a,k'> \in Com} a^-(C_{k'}) \cap g \quad , \quad C_{k_{err}} = Val \tag{2.2}$$

où $a^-(X) = \{V \mid \exists V' \in X, (V, V') \in a\}$

Intuitivement, en tout point k non initial, une valuation V accessible en k est obtenue par exécution d'une commande $< k', g, a, k >$ sur une valuation V' accessible en k', satisfaisant g, et reliée à V par a ($(V', V) \in a$). De même, en tout point k différent du but k_{err}, une valuation V co-accessible en k doit mener, par l'exécution d'une commande $< k, g, a, k' >$, à une valuation V' co-accessible en k', c'est-à-dire que V doit satisfaire g, et que (V, V') doit appartenir à a.

L'avantage de cette sémantique (souvent appelée "collecting sémantics") est qu'on obtient un *système d'équations* de points-fixe, au lieu d'un équation unique. Nous verrons au chapitre 3 que ce *partitionnement* permet d'améliorer de manière très importante la précision des calculs dans le cas d'un calcul approché de la solution.

2.3.5 Composition des automates interprétés et observateurs

Le but de cette section est de montrer comment les propriétés de sûreté peuvent se ramener à des propriétés d'accessibilité, grâce à la notion d'"observateur synchrone" [HLR93]. L'idée est d'associer à une propriété de sûreté un automate interprété appelé observateur, qui va fonctionner comme un reconnaisseur des traces satisfaisant la propriété. Composé de manière adéquate (c'est à dire en parallélisme synchrone) avec l'automate à vérifier, l'observateur détecte toute violation de la propriété en rejoignant un point de contrôle d'erreur. Ainsi, la vérification consiste à s'assurer que la composition de l'automate initial avec son observateur ne peut pas atteindre un état où l'observateur est à son point d'erreur, ce qui est une propriété d'accessibilité.

Nous définissons d'abord la composition parallèle synchrone sur les automates interprétés : intuitivement, composés en parallèle synchrone, deux automates vont exécuter des commandes ensemble, à condition que les valeurs des variables coíncident des deux cotés — l'idée étant qu'une variable est soit locale à un automate (et donc laissée non spécifiée par l'autre), soit calculée par un automate et lue (et donc laissée non spécifiée) par l'autre.

Définition 2.11 (Produit synchrone)
Le produit synchrone $\mathcal{A} \times \mathcal{A}'$ *de deux automates interprétés* $\mathcal{A} = (K, Com, k_{init})$ *et* $\mathcal{A}' = (K', Com', k'_{init})$ *est l'automate* $(K \times K', Com^{\times}, (k_{init}, k'_{init})$ *où*

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

Autrement dit, une commande du produit à partir d'un état composé (k_1, k'_1) consiste en la composition (synchrone) d'une commande de chaque composant : la composition synchrone de deux commandes (g, a) et (g', a') sur une valuation V n'est possible que si les deux gardes sont satisfaites $((g \wedge g')(V))$, et son résultat V' est un résultat commun des deux actions $((V, V') \in (a \cap a'))$.

Soit $\mathcal{A}$ un automate interprété, agissant sur un ensemble de variables Var. Soit P une propriété de sûreté.

Définition 2.12 (Observateur)
Un observateur *de* P *est un automate interprété* $\mathcal{O}$ *agissant sur un ensemble de variables* $Var \cup Var'$*, avec* $Var \cap Var' = \emptyset$*, et tel que*

- *les variables de* Var *sont laissées non spécifiées dans* $\mathcal{O}$
- $\mathcal{O}$ *possède un point de contrôle distingué, noté* k_{err}*, que l'automate rejoint (et ne quitte plus jamais) si et seulement si l'évolution des variables de* Var *viole* P*. Formellement, considérons* $(< k_0, V_0 \uplus V'_0 >, < k_1, V_1 \uplus V'_1 >, \ldots, < k_n, V_n \uplus V'_n)$ *une trace de* $\mathcal{O}$ *; alors*

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

Proposition 1 *Si* $\mathcal{O}$ *est un observateur de* P*, alors* $\mathcal{A}$ *satisfait* P *si et seulement si* $\mathcal{A} \times \mathcal{O}$ *n'atteint jamais un état de* $K_{err} = \{(k, k_{err}) \mid k \in K\}$.

Ainsi, toute propriété de sûreté dont on sait fournir un observateur, peut être transformée en propriété d'accessibilité. C'est la méthode classique, en particulier, pour spécifier les propriétés de sûreté en programmation synchrone.

Chapitre 3

Interprétation Abstraite

3.1 Introduction

Dans le chapitre précédent, nous avons vu quels types de systèmes et quelles propriétés nous voulons vérifier. En particulier, nous avons vu

- comment une propriété de sûreté pouvait être traduite en une propriété d'accessibilité (§2.3.5),
- comment la vérification d'une propriété d'accessibilité se ramenait à un calcul d'ensemble d'états accessibles $acc(Q_{init})$ ou co-accessibles $coacc(Q_{err})$ (§2.2.2),
- enfin, que ces ensembles d'états étaient solutions d'équations de points-fixes (§2.2.2)

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

ou de systèmes d'équations de points-fixes (§2.3.4)

$$\left\{ A_k = \bigcup_{<k',g,a,k>\in Com} a(A_{k'} \cap g) \right\}_{k \neq k_{init}} \quad , \quad A_{k_{init}} = Val$$
$$\left\{ C_k = \bigcup_{<k,g,a,k'>\in Com} a^-(C'_k) \cap g \right\}_{k \neq k_{err}} \quad , \quad C_{k_{err}} = Val$$

Les théorèmes classiques nous assurent que les équations de points fixes considérées ont bien des plus petites solutions, lesquelles sont bien caractérisées : d'après le théorème de Kleene, l'équation $X = F(X)$ — où F est une fonction continue d'un treillis complet dans lui-même — a une plus petite solution

$$\mathrm{lfp}(F) = \bigsqcup_{n \geq 0} F^n(\bot) \tag{3.1}$$

où $\sqcup$ est l'opération de borne supérieure, et $\bot$ est le plus petit élément du treillis complet. Dans le cas qui nous intéresse, le treillis complet est l'ensemble des parties de Q, ordonné par l'inclusion, l'opération de borne supérieure est l'union des ensembles d'états, et le plus petit élément est l'ensemble vide.

En général, la résolution de l'équation (3.1) est impossible. Lorsque l'ensemble Q des états du système est infini, le calcul des itérés $F(\bot), F(F(\bot)), F(F(F(\bot))), \ldots$ pose deux types de problèmes :

– il faut savoir savoir représenter des ensembles arbitraires d'états et calculer sur ces ensembles ;
– le calcul itératif peut ne pas terminer.

Cependant nous avons aussi remarqué, §2.2.4, que les propriétés d'accessibilité pouvaient être prouvées de manière conservative, en ne calculant qu'une approximation de la solution. La résolution approchée des équations de point-fixe est l'objectif principal de l'*interprétation abstraite*, domaine introduit par Patrick et Radhia Cousot dans les années 70 [CC77], et dont nous rappellerons les principes dans la première partie de ce chapitre. Dans une deuxième partie, nous ferons un rapide tour d'horizon des interprétations abstraites dédiées aux propriétés numériques.

3.2 Principes de l'interprétation abstraite

Nous avons identifié les deux obstacles à la résolution exacte des équations de points-fixes, qui sont

– l'impossibilité de représenter des "valeurs" complexes — les ensembles infinis quelconques d'états — et de calculer sur ces valeurs ;
– l'impossibilité, en général, de calculer la limite d'une itération infinie.

L'interprétation abstraite propose des solutions approchées à ces deux problèmes : d'une part, elle offre une notion de domaine abstrait, permettant de calculer sur des "valeurs abstraites" plus simples que les ensembles d'états : d'autre part, selon le choix de ces valeurs abstraites, il se peut que la finitude des itérations soit assurée ; dans le cas contraire, des techniques d'extrapolation de la limite sont proposées.

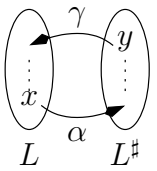
3.2.1 Domaine abstrait

Une interprétation abstraite est d'abord caractérisée par un domaine de valeurs abstraites. Ces valeurs servent à approcher, ou abstraire les valeurs concrètes (ensembles d'états) trop complexes. La notion d'approximation est formalisée par une relation d'ordre : le domaine abstrait doit être un treillis complet, comme le domaine concret. À l'équation de point-fixe initiale, portant sur des valeurs concrètes, sera alors associée une équation de point-fixe abstraite, à résoudre dans le domaine abstrait. Les valeurs abstraites doivent donc être choisies pour que leur représentation en machine soit possible, ainsi que les opérations nécessaires au calcul itératif de l'équation abstraite.

Nous avons vu que le treillis concret est $(2^Q, \subseteq, \cup, \cap, \{Q\}, \emptyset)$, soit l'ensemble des parties de Q, ordonné par l'inclusion, avec $\cup$ et $\cap$ comme opérations de borne supérieure et inférieure, et $\{Q\}$ et $\emptyset$ comme plus grand et plus petit élément. Comme le cadre théorique de l'interprétation abstraite est plus général, nous noterons $(L, \sqsubseteq, \sqcup, \sqcap, \top, \bot)$ ce treillis concret. Le treillis des valeurs abstraites sera noté $(L^\sharp, \sqsubseteq^\sharp, \sqcup^\sharp, \sqcap^\sharp, \top^\sharp, \bot^\sharp)$.

La nécessaire relation entre valeurs concrètes et abstraites est capturée par la notion de connexion de Galois : une connection de Galois entre L et $L^\sharp$ est un couple de fonctions

$\alpha : L \mapsto L^\sharp$ (abstraction) $\qquad \gamma : L^\sharp \mapsto L$ (concrétisation)



tel que

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

L'existence d'une connexion de Galois entre les treillis concret et abstrait permet d'appliquer le théorème suivant au calcul d'approximations supérieures de points-fixes :

Théorème 1 (Approximation de point-fixe)
Si L et $L^\sharp$ sont reliés par une connexion de Galois (α, γ), si F est une fonction continue de L dans lui-même, si G est une fonction continue de $L^\sharp$ dans lui-même, si

$$\forall x \in L,\ \alpha(F(x)) \sqsubseteq^\sharp G(\alpha(x)) \tag{3.2}$$

alors $lfp(F) \sqsubseteq \gamma(lfp(G))$.

Ce théorème signifie que, pour calculer une approximation supérieure du plus petit point fixe d'une fonction concrète F, on peut choisir ou construire une fonction abstraite G satisfaisant (3.2), calculer le plus petit point fixe de G dans le treillis abstrait, et enfin "concrétiser" le résultat.

3.2.2 Opérations abstraites

On peut montrer que le meilleur choix pour la fonction abstraite G est $\alpha \circ F \circ \gamma$. Cependant cette fonction n'est en général pas calculable, puisqu'elle implique un calcul dans le domaine concret. D'autre part, on veut obtenir un moyen automatique pour construire G à partir de F. Or, la fonction F, dans le cas de l'analyse des programmes, est construite par composition de fonctions élémentaires sur les ensembles d'états (les post- ou pre-conditions des instructions élémentaires du langage de programmation), d'unions et d'intersections. De la même manière, la fonction abstraite G sera construite à partir de fonctions abstraites élémentaires, associées aux instructions du langage, et d'opérations de borne supérieure ($\sqcup^\sharp$) et inférieure ($\sqcap^\sharp$). Après le choix d'un treillis abstrait, la conception d'une interprétation abstraite doit donc s'attacher à définir ces fonctions abstraites, et à en fournir des algorithmes effectifs.

Ainsi, dans le cas des automates interprétés, et pour l'analyse "en avant" (équations (2.1), page 12), il faut donner

- une interprétation des gardes dans le treillis abstrait, c'est-à-dire un moyen effectif d'associer à toute garde g une valeur abstraite $g^\sharp$ telle que $\alpha(g) \sqsubseteq^\sharp g^\sharp$ (ou $g \sqsubseteq \gamma(g^\sharp)$)
- une interprétation des actions, c'est à dire un moyen effectif d'associer à toute action a une fonction abstraite $a^\sharp$ telle que

$$\forall y \in L,\ \alpha(a(\gamma(y)) \sqsubseteq^\sharp a^\sharp(y) \text{ (ou } \forall x \in L,\ a(x) \sqsubseteq \gamma(a^\sharp(\alpha(x))) \text{)}$$

Alors, à l'équation concrète

$$A_k = \bigcup_{<k',g,a,k>\in Com} a(A_{k'} \cap g) \quad (2.1)$$

correspond l'équation abstraite

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

3.2.3 Elargissement – Séquence descendante

Dans de nombreux cas, le calcul itératif du point fixe dans le treillis abstrait converge en un nombre fini d'itérations : c'est évidemment le cas lorsque le treillis abstrait est fini, mais aussi lorsqu'il est simplement de profondeur finie (i.e., toute chaîne strictement croissante est finie). Dans ces cas-là, le problème des itérations infinies ne se pose donc plus. Cependant, il peut s'avérer plus judicieux, pour la précision des résultats [CC92], de se placer dans des treillis plus riches qui ne satisfont pas ces propriétés. Dans ce cas, l'idée est de "deviner" la limite d'une suite infinie à partir de ses premiers termes. Pour cela, le concepteur de l'interprétation doit fournir un opérateur d'"élargissement", destiné à extrapoler la limite de la suite des itérés :

Définition 3.1 (Elargissement)
Un opérateur d'élargissement est une fonction $\nabla : L^\sharp \times L^\sharp \mapsto L^\sharp$ satisfaisant les propriétés suivantes :

1. $\forall y_1, y_2 \in L^\sharp,\ (y_1 \sqcup^\sharp y_2) \sqsubseteq^\sharp (y_1 \nabla y_2)$
2. *Pour toute suite croissante $(x_n)_{n\geq 0}$ dans $L^\sharp$, la suite définie par $y_0 = x_0,\ y_{n+1} = y_n \nabla x_{n+1}$ converge en un nombre fini d'itérations.*

Théorème 2 (Extrapolation)
Si ∇ est un élargissement sur $L^\sharp$, si G est une fonction continue de $L^\sharp$ dans $L^\sharp$, alors la suite $y_0 = \bot,\ y_{n+1} = y_n \nabla G(y_n)$ converge en un nombre fini de termes vers une limite $\hat{y}$, qui est un post-point-fixe de G (i.e., $\hat{y} \sqsupseteq^\sharp G(\hat{y})$), et donc telle que $\hat{y} \sqsupseteq^\sharp lfp(G)$.

Le théorème ci-dessus permet donc d'obtenir dans tous les cas une approximation du plus petit point-fixe de G. Si la limite $\hat{y}$ n'est pas un point-fixe, le théorème suivant permet d'améliorer la solution :

Théorème 3 (Séquence descendante)
Si $\hat{y} \neq G(\hat{y})$, alors la suite $z_0 = \hat{y},\ z_{n+1} = G(z_n)$ décroît, et tous ses termes sont des approximations supérieures de $lfp(G)$.

En résumé, pour approcher supérieurement le plus petit point-fixe d'une fonction G, au lieu de calculer la séquence exacte (et généralement infinie)

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

on calcule d'abord une séquence d'extrapolation

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

qui converge en un nombre fini de termes vers une limite $\hat{y}$, qui est une approximation correcte de $\mathrm{lfp}(G)$. Si, de plus, $\hat{y} \neq G(\hat{y})$, on peut améliorer le résultat en calculant les premiers termes de la suite

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

qui peut être finie ou infinie, mais dont les termes sont tous des approximations correctes de $\mathrm{lfp}(G)$.

3.3 Interprétations abstraites des automates interprétés

On a vu, au §2.3.4, que pour analyser un automate interprété, on pouvait résoudre un *système d'équations de point-fixe*, calqué sur la structure de contrôle de l'automate, plutôt qu'une équation simple. Un tel système sera dit *partitionné*. Concentrons nous sur le système "en avant" (le cas "en arrière" étant analogue) :

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

À ce système, on va associer un système d'équations abstraites :

$$\forall k \neq k_{init},\ A^\sharp_k = \bigsqcup^\sharp_{<k',g,a,k> \in Com} a^\sharp(A^\sharp_{k'} \sqcap g^\sharp) \quad , \quad A_{k_{init}} = \top^\sharp$$

Ce système peut être résolu itérativement, en considérant les équations une à une, dans un ordre quelconque, jusqu'à stabilisation. Ceci dit, l'ordre dans lequel les équations sont considérées a une influence sur la rapidité de la convergence, et l'application de l'élargissement peut être optimisée.

– Il est logique de considérer les équations selon la structure de contrôle : lorsque $A^\sharp_k$ change, les valeurs $A^\sharp_{k'}$ associées aux points de contrôle k' successeurs de k sont susceptibles de changer. Par ailleurs, il est inutile de considérer les points de contrôle situés en aval d'une composante fortement connexe du graphe de contrôle, avant d'avoir fait converger les valeurs associées aux points de contrôle de la composante.

– Il est inutile, pour assurer la convergence, d'appliquer l'élargissement (qui fait perdre de l'information) à chaque équation du système ; il est suffisant de choisir un ensemble $K_\nabla \subseteq K$ de *points de contrôle d'élargissement*, de telle sorte que toute boucle du graphe contienne un point de K_∇, et de n'appliquer l'élargissement qu'aux équations correspondant à des points d'élargissement. Le choix d'un ensemble minimal de points d'élargissement étant NP-complet, on se contente d'un choix heuristique (nous y reviendrons au §5.3.2).

3.4 Approximation des ensembles numériques

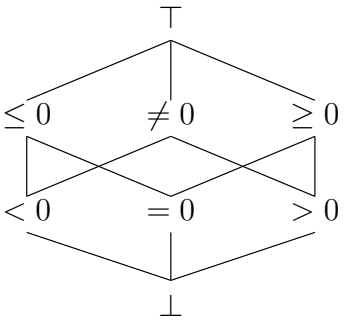
Nous nous intéressons à la vérification des programmes numériques, et nous considérerons spécifiquement le cas où un état du programme est formé d'un point de contrôle et d'une valuation des variables dans un ensemble numérique $\mathcal{N}$ ($= \mathbb{Z}$ ou $\mathbb{Q}$ ou $\mathbb{R}$). Avec la sémantique définie en §2.3.4, les ensembles que nous voulons représenter sont donc les sous-ensembles de $\mathcal{N}^n$, où n est le nombre de variables. Le treillis concret est donc

$$(2^{\mathcal{N}^n}, \subseteq, \cup, \cap, \{\mathcal{N}^n\}, \emptyset)$$

Nous faisons maintenant une revue succincte des principaux treillis abstraits qui ont été proposés dans la littérature, pour abstraire ce domaine.

3.4.1 Le treillis des signes

Une abstraction brutale consiste à associer à toute variable une valeur abstraite dans le treillis des signes représenté ci-contre. Par exemple, l'ensemble de points de la Figure 3.1.(a) serait abstrait par le premier quadrant strict $\{x > 0, y > 0\}$ (Figure 3.1.(b)). Dans ce treillis, les opérations abstraites sont sans mystère, et comme le treillis est fini, l'élargissement est inutile.



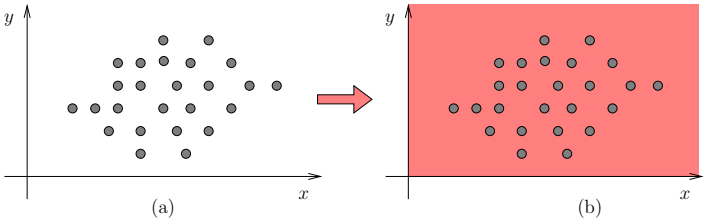


FIG. 3.1 – Abstraction selon le treillis des signes

3.4.2 Le treillis des équations affines

Une des premières abstractions non triviales, proposée par [Kar76], consiste à synthétiser des équations linéaires invariantes. Dans cette analyse, un ensemble de points de $\mathcal{N}^n$ est abstrait en la plus petite variété affine qui le contient, c'est-à-dire en l'ensemble des solutions d'un système d'équations affines. Selon cette analyse, l'ensemble de la Figure 3.1.(a) serait abstrait en l'espace toute entier (pas d'information, pas d'équation), mais celui de la Figure 3.2.(a) fournit l'équation $x - 2y = 1$ (Figure 3.2.(b). Le treillis des équations affines est infini, mais de profondeur finie (il y a au plus n variétés linéaires imbriquées dans $\mathcal{N}^n$). Il n'y a donc pas lieu de définir un élargissement. Nous reviendrons sur ce treillis au chapitre 6.

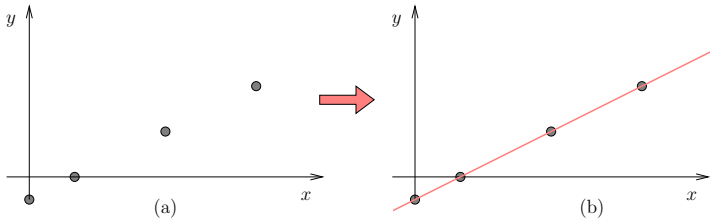


FIG. 3.2 – Abstraction en équations affines

3.4.3 Le treillis des congruences linéaires

Dans sa thèse [Gra91], Philippe Granger a proposé, en particulier, un treillis pour abstraire les parties de $\mathbb{Z}^n$ sous forme d'équations de congruences linéaires, de la forme $a\vec{x} \equiv b(c)$ (soit $\exists k \in \mathbb{Z},\ a\vec{x} = k.c + b$). Par exemple, l'ensemble de la Figure 3.3.(a) est abstrait comme sur la Figure 3.3.(b), soit en $x - 2y \equiv 2\ (6)$. Le treillis est infini, de profondeur infinie, mais satisfait la condition de chaîne ascendante : il n'existe pas de suite infinie strictement croissante de valeurs abstraites. Là encore, l'élargissement est inutile.

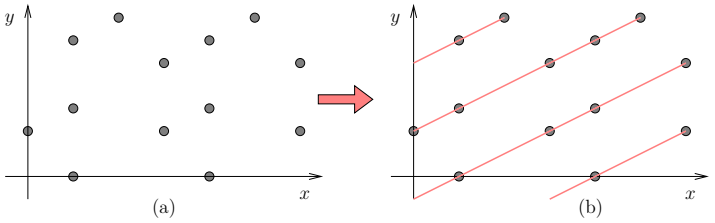


FIG. 3.3 – Abstraction en congruences linéaires

3.4.4 Le treillis des intervalles

Le treillis des intervalles [CC76] sert à déterminer l'intervalle de variation des variables numériques. L'abstraction d'un ensemble E consiste à associer à chaque variable v le couple $[\min_v, \max_v]$ des valeurs minimales (éventuellement $-\infty$) et maximales (éventuellement $+\infty$) de v dans E. L'abstraction en intervalles de l'ensemble représenté Figure 3.1.(a) est donnée Figure 3.4. Dans le treillis des intervalles, il est nécessaire de définir un élargissement pour assurer la convergence. Dans [CC76], l'élargissement suivant est proposé :

$$[a,b]\nabla[c,d] = [\text{si } c < a \text{ alors } -\infty \text{ sinon } a \text{ , si } d > b \text{ alors } +\infty \text{ sinon } b]$$

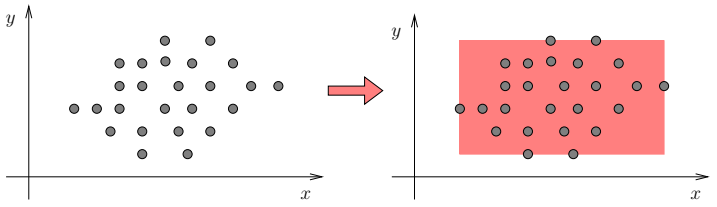


FIG. 3.4 – Abstraction en intervalles

3.4.5 Les polyèdres convexes

Le treillis des polyèdres convexes a été introduit [CH78, Hal79a] pour analyser les inégalités linéaires vérifiées de manière invariante par les variables d'un programme. L'abstraction d'un ensemble de points de $\mathcal{N}^n$ est alors le plus petit polyèdre convexe le contenant (Fig. 3.5). Notons que si cette définition à un sens pour un ensemble fini de points, ce n'est pas le cas d'un ensemble infini, dont l'enveloppe convexe n'est pas forcément polyédrique (une sphère, par exemple). Cette complication vient du fait que le treillis des polyèdres n'est pas complet (la borne supérieure d'un ensemble infini de polyèdres n'est pas forcément un polyèdre). Le véritable treillis complet est celui des convexes de $\mathcal{N}^n$, mais comme les calculs effectués par l'analyse sont toujours finis (grâce, en particulier, à l'élargissement), les solutions calculées sont toujours des polyèdres. Ce treillis étant celui sur lequel porte notre travail, nous y reviendrons en détail au chapitre 4.

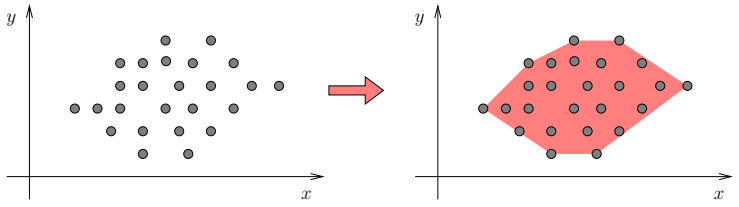


FIG. 3.5 – Abstraction en polyèdres

3.4.6 Cas particuliers de polyèdres

Les opérations sur les polyèdres convexes étant coûteuses, comme nous le verrons, certaines sous-classes ont été identifiées pour des applications spécifiques :

Les zones

Les zones sont des cas très particuliers de polyèdres, dont les contraintes ne sont que des bornes de variables (comme pour les intervalles) ou de différences de variables (Fig. 3.6). Autrement dit, toutes les contraintes sont de la forme $x\ op\ c$ ou $x - y\ op\ c$, où op est un opérateur dans $\{<, \leq, =, \geq, >\}$, et où c est une constante. Les zones sont utilisées pour l'analyse des automates temporisés [AD90, ACD93, HNSY92], peuvent être représentées sous forme de "matrices de bornes", et normalisées au moyen d'algorithmes de graphes.

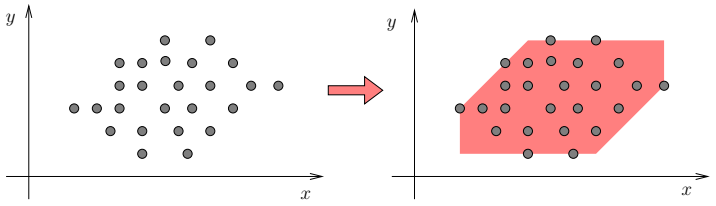


FIG. 3.6 – Approximation en "zone"

Les octogones

Les zones ont été généralisées, sans accroître la complexité des opérations, par Antoine Miné [Min01], qui permet aussi les contraintes de la forme $x + y\ op\ c$. Les polyèdres résultant sont appelés "octogones" (Fig. 3.7).

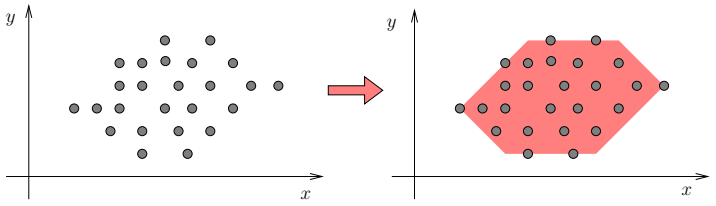


FIG. 3.7 – Approximation octogone

Les octaèdres

Les octogones ont été généralisés à leur tour par les "octaèdres" [CC04], en considérant les contraintes linéaires à coefficients dans $\{-1, 0, 1\}$ (donc, pouvant impliquer plus que 2 variables). Cependant, la complexité polynomiale n'est pas préservée.

3.4.7 Ensembles semi-linéaires et arithmétique de Presburger

Nous finirons ce rapide tour d'horizon des techniques de représentation des ensembles numériques en citant des travaux moins directement liés à l'interprétation abstraite, et qui visent à analyser de manière exacte les valeurs possibles des variables entières au cours de l'exécution d'un programme : certains auteurs [BW94, BGP97, CJ98, FS00] exploitent la décidabilité de l'arithmétique de Presburger pour représenter des ensembles de vecteurs d'entiers exactement. Aussi longtemps que les ensembles d'états à représenter sont *semi-linéaires*, on dispose des algorithmes nécessaires à leur manipulation. Pour certains programmes, l'ensemble des états accessibles peut être calculé par *accélération*, une technique analogue à l'élargissement, mais qui donne des résultats exacts.

3.5 Utilisations de l'analyse des variables numériques

L'analyse du comportement des variables numériques a des applications variées. Nous les classifierons en deux parties :

- les applications où on dispose d'un but a-priori, en général un ensemble d'états dont on veut montrer l'inaccessibilité, et qui peut être pris en compte par l'analyse ;
- les applications où on veut seulement extraire de l'information aussi précise que possible, en général des invariants, pour l'exploiter ensuite.

3.5.1 Vérification d'inaccessibilité

Nous avons vu au §2.3.5 que toutes les propriétés de sûreté se ramenaient à l'inaccessibilité de certains états. Il en est de même de la vérification d'absence d'erreurs à l'exécution (cohérence dynamique) pour laquelle il n'est même pas besoin de spécifier la propriété, et qui fait souvent appel à des aspects numériques (division par zéro, débordement de tableaux [NIAK01], débordements arithmétiques).

Dans toutes ces applications, on connaît l'ensemble Q_{err} des états dont on veut montrer l'inaccessibilité, et on peut procéder soit "en avant" — pour prouver $acc(Q_{init}) \cap Q_{err} = \emptyset$ — soit "en arrière" — pour prouver $Q_{init} \cap coacc(Q_{err}) = \emptyset$. Comme l'analyse est approchée, on calcule en fait soit $\widehat{\mathrm{acc}}(Q_{init})$, soit $\widehat{\mathrm{coacc}}(Q_{err})$, avec

$$acc(Q_{init}) \subseteq \widehat{\mathrm{acc}}(Q_{init}) \quad \text{et} \quad coacc(Q_{err}) \subseteq \widehat{\mathrm{coacc}}(Q_{err})$$

Il est alors avantageux (plus précis) d'itérer des analyse "en avant" et "en arrière", en calculant successivement

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

qui forment une séquence de plus en plus précise, tant que ni les X_n ni les Y_n ne sont vides, ni ne stabilisent.

3.5.2 Synthèse d'invariants

Une deuxième classe d'applications concerne la découverte d'invariants, sans objectif a-priori, pour une utilisation ultérieure. Ces applications concernent :

- la compilation, avec, par exemple, la découverte de code mort, la découverte d'expressions pouvant être calculées à la compilation ou à l'extérieur des boucles, la détermination de l'intervalle de variation des variables pour leur implantation optimale en mémoire, la détermination précise de la durée de vie des éléments de tableau pour l'optimisation de la gestion mémoire [LMC02] ...
- la parallélisation de programmes [ACIK97], où l'on doit analyser finement les dépendances entre variables, et notamment entre éléments de tableaux.
- la preuve interactive, où les invariants trouvés par l'analyse sont utilisés pour faciliter la tâche des démonstrateurs [BBC+00, BLO98].
- certaines applications nécessitant l'adjonction de compteurs auxiliaires dans le programme, et l'analyse de leur comportement. Par exemple, [Mer91] propose une méthode d'analyse des communications dans un système multi-processus en adjoignant des compteurs à chaque "rendez-vous", en analysant le code de chaque processus séparément, puis en combinant les résultats en égalisant les compteurs des rendez-vous correspondants. On peut aussi chercher à montrer la terminaison des boucles (une propriété de vivacité), voire chercher à évaluer le nombre d'itérations [Pod04], en ajoutant un compteur de boucle, puis en trouvant une borne à ce compteur.

Chapitre 4

Le treillis des polyèdres

Comme nous l'avons annoncé précédemment, notre travail porte sur l'analyse des relations linéaires, qui est basée sur le treillis des polyèdres convexes. Dans ce chapitre, nous rappelons les principaux algorithmes relatifs aux polyèdres, en détaillant certains d'entre-eux qui sont plus ou moins bien explicités dans la littérature. Les algorithmes que nous utilisons sont basés sur la double caractérisation des polyèdres — par des systèmes de contraintes et par des systèmes générateurs —, que nous rappelons 4.1. Le 4.2 détaille la traduction d'une représentation dans l'autre, qui est l'algorithme clé permettant d'effectuer toutes les autres opérations, et surtout de simplifier les représentations. Le §4.3 décrit les opérations usuelles effectuées sur les polyèdres convexes. Un exemple d'analyse est ensuite détaillé §4.4. Le §4.6 met en évidence les problèmes de complexité qui motivent notre travail. Enfin, on présentera au §4.5 les différentes bibliothèques de polyèdres disponibles.

Les références générales de ce chapitre sont [Hal79a] et [Jea00].

On considère un ensemble numérique $\mathcal{N}$ et on se place dans l'espace affine $\mathcal{N}^n$. On note $X \cdot Y$, ou simplement XY le produit scalaire de X par Y, $\sum_{i=1}^{n} X_i\ Y_i$ dans $\mathcal{N}^n$.

Dans la suite U désignera un vecteur de $\mathcal{N}^n$ et b une valeur de $\mathcal{N}$.

4.1 La double représentation

Définition 4.1 (Polyèdre convexe)

On appelle polyèdre convexe fermé *(ou simplement polyèdre) l'intersection d'un nombre fini m de demi-espaces fermés de $\mathcal{N}^n$.*

4.1.1 Contraintes linéaires

Proposition 2 (Représentation par des contraintes linéaires)

Tout polyèdre convexe peut être caractérisé comme l'ensemble des solutions d'un système de contraintes linéaires $AX \leq B$, *où A est une matrice $m \times n$ et B est un*

vecteur de $\mathcal{N}^m$. *On appellera* contrainte linéaire *une inéquation* $aX \leq b$ *ou une équation* $aX = b$, $a \in \mathcal{N}^n$, $b \in \mathcal{N}$.

Exemple : La figure 4.1 représente le polyèdre correspondant au système de contraintes

$$\begin{aligned} 2X_2 - X_1 &\leq 7 \\ -X_1 - X_2 &\leq -5 \\ -X_2 &\leq -1 \end{aligned}$$

soit encore $AX \leq B$ avec

$$A = \begin{bmatrix} -1 & 2 \\ -1 & -1 \\ 0 & -1 \end{bmatrix} \quad \text{et} \quad b = \begin{bmatrix} 7 \\ -5 \\ -1 \end{bmatrix}$$

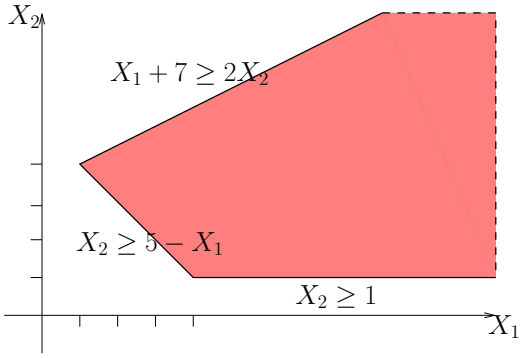


FIG. 4.1 – Représentation sous forme de contraintes

C'est évidemment cette représentation qui nous intéressera, en pratique, puisqu'elle donne des informations directes sur les variables.

4.1.2 Non canonicité des systèmes de contraintes

Dans ce paragraphe, nous mettons en lumière la non canonicité de la représentation par des contraintes linéaires. D'une part, un système de contraintes peut contenir des contraintes *redondantes*, et d'autre part lorsque le polyèdre est contenu dans un sous-espace strict de $\mathcal{N}^n$(i.e., quand il satisfait des équations linéaires), il peut être caractérisé par plusieurs systèmes de contraintes minimaux.

Définition 4.2 (contraintes redondantes)
Une contrainte $aX \leq b$ *sera dite* redondante *dans le système* $AX \leq B$ *s'il est possible de la supprimer sans changer le polyèdre décrit. En d'autres termes, la satisfaction des autres contraintes implique la satisfaction de* $aX \leq b$. *Un système de contraintes sans contrainte redondante sera dit* minimal.

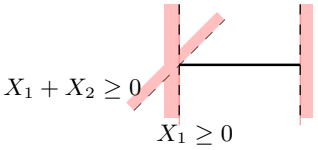


FIG. 4.2 – Contraintes mutuellement redondantes

La prolifération des contraintes redondantes pénalise évidemment les opérations sur les systèmes de contraintes. Il sera donc essentiel d'être capable de minimiser les systèmes de contraintes.

Définition 4.3 (dimension d'un polyèdre)
On appelle dimension *d'un polyèdre P, la dimension du plus petit sous-espace affine contenant P. Autrement dit, la dimension de P est égale à n diminué du nombre $\sharp e$ d'équations indépendantes satisfaites par P.*

Si la dimension d'un polyèdre P est égale à n (i.e., $\sharp e = 0$), et si son système de contraintes ne contient pas de contraintes redondantes, alors ce système de contrainte est unique, à un facteur multiplicatif près (puisque des contraintes multiples l'une de l'autre sont équivalentes).

Définition 4.4 (contraintes mutuellement redondantes)
Deux contraintes c_1 et c_2 sont dites mutuellement redondantes *dans un système de contraintes, s'il est possible de supprimer l'une ou l'autre sans changer le polyèdre décrit. Cette situation n'intervient que lorsque la dimension du polyèdre est strictement inférieure à n.*

Exemple : Dans le système $X_2 = 0, 0 \leq X_1 \leq 1, X_1 + X_2 \geq 0$, les contraintes $0 \leq X_1$ et $X_1 + X_2 \geq 0$ sont mutuellement redondantes (cf. Fig 4.2).

Il en résulte que, dès que la dimension d'un polyèdre est strictement inférieure à n, celui-ci n'admet pas de représentation minimale unique sous forme de système de contraintes.

Si le polyèdre P contient l'origine, on peut définir avec le produit scalaire une forme dite normale. Si P ne contient pas l'origine, on peut s'y ramener par translation. Cette représentation sera utilisée dans le paragraphe 4.1.5 afin de permettre le passage d'une représentation à l'autre.

Définition 4.5 (forme normale des contraintes)
Si le polyèdre P contient l'origine, on peut obtenir une forme dite normale (A_1, A_2, A_3). Ces 3 ensembles séparent les contraintes selon leur nature : une inégalité avec constante positive, avec constante nulle ou une égalité à 0.

- *la forme normale d'une inéquation $ax \leq b$ avec $b > 0$ est $(a/b)x \leq 1$ $(a/b \in A_1)$*
- *la forme normale d'une inéquation $ax \leq b$ avec $b = 0$ est inchangée $(a \in A_2)$*
- *la forme normale d'une équation $ax = 0$ est inchangée $(a \in A_3)$*

4.1.3 Générateurs

Définition 4.6 (système générateur)

Tout polyèdre convexe P peut être caractérisé par un système générateur*, formé de trois ensembles finis S, R et D de vecteurs de $\mathcal{N}^n$, respectivement appelés ensemble des* sommets*, des* rayons*, et des* droites[1] *de P. P* est alors l'ensemble des points que l'on peut obtenir comme somme d'une *combinaison convexe* des sommets, d'une *combinaison positive* des rayons, et d'une *combinaison linéaire* des droites :

$$P = \left\{ \sum_{s_1 \in S} \lambda_i s_i + \sum_{r_j \in R} \mu_j r_j + \sum_{d_k \in D} \nu_k d_k \mid \lambda_i \in [0,1], \Sigma_i \lambda_i = 1, \mu_j \geq 0, \nu_k \in \mathcal{N} \right\}$$

Exemple : Le polyèdre de la figure 4.1 peut être représenté de manière duale sous forme de générateurs (Fig. 4.3) :

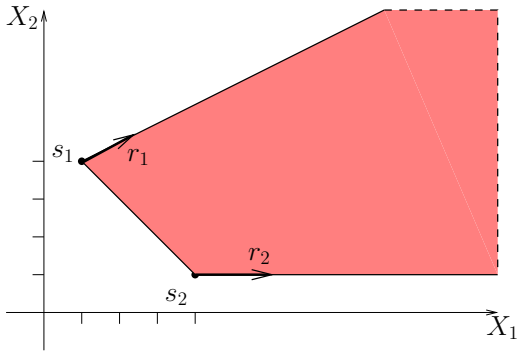


FIG. 4.3 – Représentation sous forme de générateurs

Avec :

$$S = \left\{ s_1 = \begin{bmatrix} 1 \\ 4 \end{bmatrix}, s_2 = \begin{bmatrix} 4 \\ 1 \end{bmatrix} \right\} \quad , \quad R = \left\{ r_1 = \begin{bmatrix} 2 \\ 1 \end{bmatrix}, r_2 = \begin{bmatrix} 1 \\ 0 \end{bmatrix} \right\} \quad , \quad D = \emptyset$$

4.1.4 Non canonicité des systèmes générateurs

Comme dans le cas des contraintes, la représentation sous forme de générateurs n'est en général pas canonique : elle peut contenir des éléments redondants (appelés ici "non extrémaux"), et, lorsque le polyèdre contient une droite, il admet un nombre infini de système générateurs minimaux.

[1] Dans toute la suite on confondra une droite avec un de ses vecteurs directeurs. De même, on confondra sous le terme de "rayon", une demi-droite et un de ses vecteurs directeurs.

Définition 4.7 (points et rayons extrémaux)
Un point (resp., un rayon) d'un polyèdre sera dit extrémal *s'il ne peut être obtenu comme combinaison convexe (resp., positive) d'autres points (resp., d'autres rayons qui ne lui sont pas colinéaires) de P.*

Les sommets s_1 et s_2, les rayons r_1 et r_2 de l'exemple de la figure 4.3 sont extrémaux. Le point $(4, 4)$ appartient au polyèdre, mais n'est pas extrémal, le vecteur $(3, 1)$ est un rayon non extrémal.

Définition 4.8 (codimension d'un polyèdre)
On appelle codimension *d'un polyèdre P, la dimension du plus grand sous-espace affine contenu dans P. Autrement dit, la codimension de P est égale au nombre ♯d de droite indépendantes contenues dans P.*

Un polyèdre de codimension nulle (pas de droite), admet un système générateur minimal unique (à un facteur multiplicatif près en ce qui concerne les rayons), dont les sommets sont les points extrémaux du polyèdre, et dont les rayons sont ses rayons extrémaux (normalisés).

Si le polyèdre contient une droite, il n'a pas de point extrémaux, ni de système générateur minimal unique. Dans ce cas, un système générateur minimal est constitué d'un ensemble minimal de droites (une base du plus grand sous-espace affine inclus dans le polyèdre), et des points et rayons extrémaux d'une *section* du polyèdre : une section de P est l'intersection de P avec n'importe que sous-espace supplémentaire[2] du plus grand sous-espace inclus dans le polyèdre.

Dans l'exemple de la figure 4.4, le polyèdre contient la droite d d'équation $x_1 = x_2$, de vecteur directeur $(1, 1)$. La droite d'équation $x_1 = 0$ est un sous-espace supplémentaire à d, et le segment $-1 \leq x_2 \leq 1$ est donc une section du polyèdre, de sommets $s_1 = (0, -1), s_2 = (0, 0)$ et qui n'a pas de rayon. $(\{s_1, s_2\}, \emptyset, \{d\})$ est donc un système générateur minimal du polyèdre. Mais on aurait pu choisir un autre sous-espace supplémentaire (par exemple $x_2 = 0$, ou $x_1 + x_2 = 0$), donc une autre section, et obtenir un autre système générateur minimal.

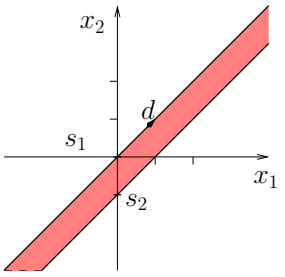


FIG. 4.4 – Polyèdre contenant une droite

[2]Rappelons que deux sous espaces affines, E et E' sont supplémentaires, si leur intersection est réduite à un point, et si tout point de l'espace tout entier est combinaison linéaire de points de E et de E'.

4.1.5 Lien entre les deux représentations

Satisfaction des contraintes par les éléments générateurs

Soit P un polyèdre défini par le système de contraintes $AX \leq B$. Alors :
- Evidemment, un point s appartient à P si et seulement si $As \leq B$;
- Un vecteur r est un rayon de P si et seulement si $Ar \leq 0$;
- un vecteur d est une droite de P si et seulement si $Ad = 0$.

Polyèdre polaire

Les §4.1.1 et 4.1.3 suggèrent une dualité forte entre les deux représentations, dualité complètement confirmée par la notion de polyèdre polaire :

Définition 4.9 (polyèdre polaire)
On appelle polyèdre polaire de P, le polyèdre P^+ défini par : $P^+ = \{ X \in \mathcal{N}^n \mid \forall Y \in P \; X.Y \leq 1\}$

Propriétés :
- P^+ contient toujours l'origine
- Si P contient l'origine alors $P^{++} = P$
- Si (S, R, D) est un système générateur de P, alors $X \in P^+$ si et seulement si :

$$\forall s \in S,\ s \cdot X \leq 1\ ,\ \forall r \in R,\ r \cdot X \leq 0\ ,\ \forall d \in D,\ d \cdot X = 0$$

- Si P contient l'origine et si $(A_1,\ A_2,\ A_3)$ est la forme normale de son système de contraintes (voir définition 4.5) alors $(A_1,\ A_2,\ A_3)$ constitue un système générateur de P^+.

Le polyèdre polaire permet, en ayant une représentation et une caractérisation ou un algorithme sur cette représentation, d'obtenir une propriété duale ou un algorithme dual sur l'autre représentation. Par exemple avec un algorithme calculant l'intersection de 2 polyèdres donnés sous forme de contraintes, on peut déduire un algorithme pour calculer l'enveloppe convexe (opération duale) de 2 polyèdres donnés sous forme de générateurs.

4.1.6 Minimisation des représentations

Dans ce paragraphe, nous allons montrer que la connaissance des deux représentations d'un même polyèdre permet de minimiser ces représentations.

Définition 4.10 (saturation)
Soit $aX \leq b$ une inéquation linéaire. Alors on dit qu'un point s (resp., un rayon r) sature *la contrainte, si et seulement si $as = b$ (resp., $ar = 0$). Soient $AX \leq B$ et (S, R, D) un système de contrainte et un système générateur d'un même polyèdre P. Alors,*

- *pour toute inéquations c, $Sat(c)$ dénote le sous-ensemble de $S \cup R$ des sommets et rayons qui saturent c.*
- *pour tout sommet $s \in S$ (resp., rayon $r \in R$, on note $Sat(s)$ (resp., $Sat(r)$) l'ensemble des inéquations, dans $AX \leq B$, saturées par s (resp., r)*

Proposition 3 (redondance, extrémalité)
- *Une contrainte c est redondante pour P, si et seulement si il existe une autre contrainte c' de P telle que $Sat(c) \subseteq Sat(c')$.*
- *Un sommet s (resp., un rayon r) est non extrémal dans P, si et seulement si il existe un autre sommet s' (resp., un autre rayon r') de P, tel que $Sat(s) \subseteq Sat(s')$ (resp., $Sat(r) \subseteq Sat(r')$).*
- *Deux contraintes c et c' de P sont mutuellement redondantes si et seulement si $Sat(c) = Sat(c')$.*

La précédente proposition permet de simplifier les représentations, et d'identifier les contraintes mutuellement redondantes, au prix du calcul des saturations (i.e., le produit de la matrice de contraintes par la matrice des générateurs). Ce calcul peut être coûteux si les éléments redondants prolifèrent et font croître la taille des matrices, c'est pourquoi on tentera d'identifier les éléments redondants au plus tôt, au cours de la traduction d'une représentation dans l'autre (voir §4.2).

La minimisation des équations et des droites est standard (extraction d'une base irredondante). Nous y reviendrons au §6.1.1.

Nous utiliserons par la suite les définitions suivantes :

Définition 4.11 (adjacence)

Soit P un polyèdre de dimension $n - \sharp e$, de codimension $\sharp d$. Alors,
- *deux sommets s et s' sont dits* adjacents *si et seulement si ils saturent $n - \sharp e - \sharp d - 1$ inéquations communes.*
- *un sommet s et un rayon r sont dits* adjacents *si et seulement si ils saturent $n - \sharp e - \sharp d - 1$ inéquations communes.*
- *deux rayons r et r' sont dits* adjacents *si et seulement si ils saturent $n - \sharp e - \sharp d - 2$ inéquations communes.*

4.2 Calcul de la représentation duale

Motzkin [MT53] a décrit un algorithme itératif qui résout le problème de calcul de la forme duale. Un algorithme similaire a été décrit par Chernikova [Che68] et redécouvert par [Hal79b]. Par la suite, Le Verge [LeV92] a proposé une optimisation en améliorant la détection des éléments redondants.

Dans la pratique, ces algorithmes ne manipulent pas des polyèdres de dimension n mais des cônes de dimension $n + 1$, centrés à l'origine. Un cône est représenté par un sommet (l'origine) et par un ensemble de rayons et de droites (vues comme des

rayons bidirectionels). Le polyèdre est alors vu comme l'intersection du cône avec un hyperplan. Ce choix technique ne change en rien la généralité de la méthode. Comme pour les polyèdres, un cône peut être décrit sous forme de contraintes ou sous forme de générateurs. Le fait de transformer le polyèdre en cône permet plusieurs simplifications du traitement. Tout d'abord en ajoutant une dimension les sommets et les rayons se traitent de la même façon. Un autre avantage est que le cône contient toujours l'origine (cf. §4.1.5), et qu'on peut se ramener à chercher les solutions positives d'un système, auquel cas il est possible d'utiliser des méthodes basées sur le simplexe.

On donnera l'intuition de l'algorithme de Motzkin sur des polyèdres avant de donner sa version sur des cônes.

4.2.1 Principe de l'algorithme de Motzkin

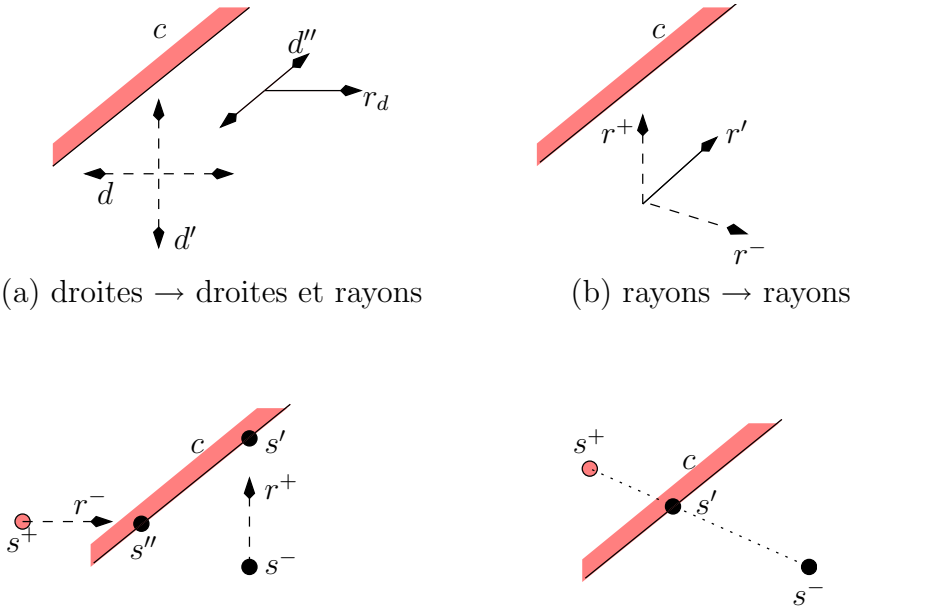
L'algorithme de Motzkin sert aussi bien à calculer le système générateur d'un polyèdre connaissant son système de contraintes, qu'à opérer la traduction inverse, par dualité. Nous nous placerons dans le premier cas.

L'algorithme part d'une double description (contraintes et générateurs) minimale d'un polyèdre qui inclut le polyèdre dont on souhaite obtenir une double description. Dans la pratique on prend le polyèdre univers, dont un système générateur est constitué par le sommet origine, un ensemble vide de rayons, et une base de l'espace comme ensemble de droites. Initialement, l'ensemble de contraintes prises en compte est vide.

Informellement, un pas de l'algorithme consiste à prendre en compte une nouvelle contrainte $c = (aX \leq b)$[3]. Alors :

1. Si toutes les droites du système générateur courant satisfont la contrainte, l'ensemble des droites ne change pas. Sinon, une droite d ne satisfaisant pas la contrainte (i.e., telle que $ad \neq 0$) est choisie : elle est composée linéairement avec toutes les autres droites d' ne satisfaisant pas la contrainte, pour donner naissance à de nouvelles droites ($d'' = (ad')d - (ad)d'$) qui satisfont la contrainte, et le vecteur d ou $-d$, selon le signe de ad, devient un nouveau rayon r_d (Fig. 4.5.(a)) ;
2. Pour tout couple (r^-, r^+) de rayons *adjacents* du système générateur courant, tel que r^- satisfait strictement la contrainte ($ar < 0$) et r^+ ne la satisfait pas ($ar^+ > 0$), r^+ est remplacé par $r' = (ar^+)r^- - (ar^-)r^+$, combinaison positive qui sature la contrainte (Fig. 4.5.(b)) ;
3. Pour tout couple (s^-, r^+) (resp., (s^+, r^-)) formé d'un sommet satisfaisant strictement (resp., ne satisfaisant pas) la contrainte (i.e., $as^- < b$ et $as^+ > b$) et d'un rayon qui ne la satisfait pas (resp., qui la satisfait strictement), (s^-, r^+) (resp., (s^+, r^-)) étant *adjacents*, r^+ (resp., s^+) est remplacé par le sommet $s' = s^- + ((b - as^-)/ar^+).r^+$ (resp., $s'' = s^+ + ((b - as^+)/ar^-).r^+$) qui sature la contrainte (Fig. 4.5.(c)) ;

[3]Nous traitons le cas d'une inéquation, celui des équations s'en déduit facilement.



(a) droites → droites et rayons (b) rayons → rayons

(c) rayon+sommet→ sommet (d) sommet+sommet→sommet

FIG. 4.5 – Les étapes de l'algorithme de Motzkin

4. Pour tout couple de sommets (s^-, s^+), adjacents et situés de part et d'autre de la contrainte, s^+ est remplacé par le sommet $s' = \lambda s^- + (1-\lambda)s^+$ — avec $\lambda = (as^+ - b)/(as^+ - as^-) \in [0,1]$ —, qui sature la contrainte (Fig. 4.5.(d)) ;
5. les éléments du système générateur courant, qui ne satisfont pas la contrainte sont supprimés ;

Le système générateur obtenu est minimal. Les contraintes sont introduites une à une jusqu'à ce que toutes les contraintes du système initial aient été prises en compte.

4.2.2 Optimisations

Comme il a été dit plus haut, l'algorithme est, en général, implanté sur une représentation des polyèdres sous forme de cônes.

L'algorithme de Motzkin construit un système générateur *minimal*, grâce au fait que seuls les éléments générateurs *adjacents* sont combinés. Malheureusement, la détermination de l'adjacence est coûteuse, c'est pourquoi les principales optimisations ont consisté à s'affranchir de ce calcul :

– Le Verge [LeV92], puis Wilde [Wil93], proposent de combiner tous les couples d'éléments situés de part et d'autre de la contrainte, sans plus se préoccuper de l'adjacence. Le résultat de la combinaison n'étant plus forcément extrémal, son extrémalité est testée a-posteriori au moyen du critère suivant :

> Soit $n-\sharp e$ la dimension d'un polyèdre. Alors un sommet (resp., un rayon) qui ne sature pas au moins $n - \sharp e$ (resp., $n - \sharp e - 1$) inéquations n'est pas extrémal.

– La condition d'extrémalité de Le Verge n'est que nécessaire, et n'implique donc pas la minimalité du résultat, qui doit finalement être assurée par une analyse complète des saturations (cf. Proposition 3). La bibliothèque de Parme [BRZH02], utilise la condition nécessaire et suffisante d'extrémalité suivante :

> Soit $n - \sharp e$ la dimension d'un polyèdre, $\sharp d$ sa codimension. Alors un sommet (resp., un rayon) est extrémal si et seulement si il sature au moins $n - \sharp e - \sharp d$ (resp., $n - \sharp e - \sharp d - 1$) inéquations.

Une autre optimisation, utilisée dans PIPS [IJT91], résulte de la remarque que, dans la pratique, les matrices intervenant dans les représentations sont souvent très creuses, et consiste à utiliser des structures de données classiques pour les matrices creuses.

4.3 Les opérations sur les polyèdres

Une interprétation abstraite des automates interprétés (cf. § 3.3) basée sur le treillis des polyèdres requiert la définition des opérations suivantes :

– les opérations du treillis :
  – calcul de la relation d'ordre (test d'inclusion des polyèdres)
  – Comparaison à $\bot$, $\top$ (test du vide, du plein)
  – borne inférieure, borne supérieure (intersection, enveloppe convexe)
– effet d'une action, en avant et/ou en arrière
– élargissement

Ces opérations utilisent tantôt une représentation, tantôt l'autre, voire les deux, ce qui confirme l'intérêt de disposer des deux représentations.

4.3.1 Les tests

Nous aurons besoins d'effectuer les tests suivants :

Le test du vide : un polyèdre est vide si ses contraintes ne sont pas satisfaisables. Ce test s'effectue aisément sur le système générateur, puisque un polyèdre est vide si et seulement si son ensemble de sommets et vide.

Le test du plein : un polyèdre est équivalent au polyèdre univers si ses contraintes sont toujours vraies, autrement dit si son système de contraintes est vide.

Le test d'inclusion : ici on utilise les deux représentations, puisque

$$P \subseteq P' \;\Leftrightarrow\; (A's \leq B', \forall s \in S,\; A'r \leq 0, \forall r \in R,\; A'd = 0, \forall d \in D)$$

où (S, R, D) est un système générateur de P, et $A'X \leq B'$ est un système de contraintes de P'.

4.3.2 L'intersection

L'intersection de deux polyèdres convexes reste un polyèdre convexe. L'intersection vérifie les contraintes des deux polyèdres. Donc, trivialement, la conjonction des systèmes de contraintes de deux (ou plusieurs) polyèdres est un système de contraintes de leur intersection. Evidemment, cette conjonction mérite en général d'être minimisée.

4.3.3 L'enveloppe convexe

L'union de deux polyèdres convexes n'est pas nécessairement convexe. L'opération de borne supérieure dans le treillis des polyèdres est l'*enveloppe convexe* (notée $\sqcup$), qui associe à deux (ou plusieurs) polyèdres le plus petit polyèdre les contenant (Fig. 4.6).

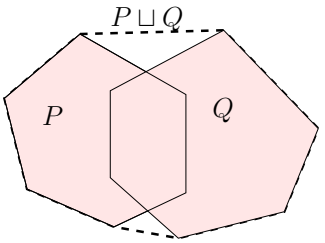


FIG. 4.6 – Enveloppe convexe de deux polyèdres

C'est l'opération duale de l'intersection, et un système générateur de $P_1 \sqcup P_2$ est formé de l'union des sommets de P_1 et P_2, l'union de leurs rayons, et l'union de leurs droites. Ici encore, le système générateur obtenu n'est en général pas minimal.

4.3.4 Effet d'une action, projection et transformation affine

Une action d'un automate interprété, telle que nous l'avons définie §2.3.1 est une relation quelconque entre les variables. On ne peut évidemment pas définir l'effet d'une telle action sur un polyèdre en toute généralité. On se contente, en général, de le définir dans le cas des relations *affines*, et de donner le moyen d'approcher brutalement le résultat dans les autres cas, par élimination des variables concernées (quantification existentielle).

Image par une relation affine

On se restreint au cas des relations a de la forme

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

où F est une matrice carrée $n \times n$, et G est un n-vecteur[4]. Ce cas est intéressant, du point de vue des programmes considérés, puisqu'il correspond aux actions définies par de affectations affines aux variables.

[4]Il ne serait pas bien compliqué de considérer des relations affines plus générales de la forme $FX + F'X' \leq G$.

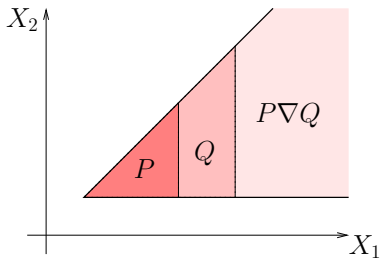


FIG. 4.7 – Elargissement standard pour les polyèdres convexes

Comme au §2.3.4, on note : $a(P) = P[X := FX + G] = \{FX + G | X \in P\}$ et $a^-(P) = P[X := FX + G]^- = \{X \mid FX + G \in P\}$

Alors, si $AX \leq B$ et (S, R, D) sont les deux représentations de P :

– $(\{Fs + G \mid s \in S\}, \{Fr \mid r \in R\}, \{Fd \mid d \in D\}$ est un système générateur de $a(P)$;

– $AFX \leq (B - AG)$ est un système de contraintes de $a^-(P)$.

Quantification existentielle

Il est souvent utile d'éliminer certaines variables (i.e., perdre toute information les concernant), par exemple pour approcher supérieurement l'effet d'une affectation non affine à ces variables. Cette opération correspond à une quantification existentielle sur un polyèdre : $\exists x.P$. Cette opération est réalisable, à partir du système générateur de P, simplement en lui adjoignant une droite qui est le vecteur de base correspondant à x ($d_y = \delta_{xy}$), et à partir du système de contraintes de P en appliquant l'élimination de Fourier-Motzkin.

4.3.5 L'élargissement

Le choix d'un opérateur d'élargissement n'est pas un choix absolu mais un compromis validé par l'expérimentation. L'opérateur standard a été proposé par [CH78]. L'idée initiale, étant donnés 2 polyèdres P et Q, avec $P \subseteq Q$, est de construire le système de contraintes de $P\nabla Q$ en ne gardant que les inégalités de P vérifiées par Q (voir figure 4.7).

L'idée est qu'une inégalité qui est transformée (par une translation, une rotation, ...) peut l'être une infinité de fois et donc elle est supprimée. Les polyèdres P et Q sont inclus dans $P\nabla Q$. On enlève des inégalités d'un système fini d'inégalités, l'arrêt se produit donc en un nombre fini d'itérations.

Le problème de cette définition est qu'elle dépend du système de contraintes choisi pour P : si ce système n'est pas canonique — et on a vu §4.1.2 que c'était le cas lorsque P satisfait des équations — l'élargissement peut perdre de l'information inutilement.

Par exemple, considérons les polyèdres suivants (Fig. 4.8) :

– $P = \{X_1 = 0,\ 0 \leq X_2 \leq 1\}$

– $Q = \{0 \leq X_1 \leq 1,\ 0 \leq X_2 \leq X_1 + 1\}$

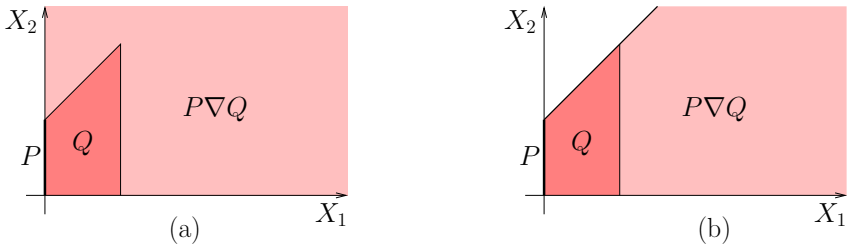


FIG. 4.8 – Elargissement d'un polyèdre satisfaisant une équation

Si on applique directement l'opérateur d'élargissement, on obtient un quart de plan (Fig. 4.8.a) :

$$P\nabla Q = \{0 \leq X_1,\ 0 \leq X_2\}$$

Par contre, si on décrit P par $P = \{X_1 = 0,\ 0 \leq X_2 \leq X_1 + 1\}$, on obtient (Fig. 4.8.b) :

$$P\nabla Q = \{0 \leq X_1,\ 0 \leq X_2 \leq X_1 + 1\}$$

Pour éviter ce problème, l'opérateur d'élargissement de [Hal79a] utilise la notion de contraintes mutuellement redondantes (cf. définition 3) : le système de contraintes de $P\nabla Q$ est formé des contraintes de Q, qui sont mutuellement redondantes, dans P, avec une contrainte de P. Dans l'exemple précédent, la contrainte $X_2 \leq X_1 + 1$ de Q est mutuellement redondante dans P avec la contrainte $X_2 \leq 1$, et l'élargissement donne le résultat de la figure 4.8.b, quelle que soit la forme choisie pour les contraintes de P.

Des travaux [BHRZ03] ont cherché à améliorer cet élargissement en distinguant des cas particuliers. Mais c'est l'opérateur standard, généralement amélioré comme suit, que nous utiliserons.

4.3.6 Elargissement limité

Dans [HPR97], une technique d'amélioration de l'élargissement est proposée : lorsqu'on connaît un ensemble de contraintes, $\mathcal{U}$ susceptibles d'être invariantes dans un état d'élargissement, on peut décider, en ce point, de garder pour le polyèdre élargi, les contraintes de $\mathcal{U}$ qui sont satisfaites par les deux opérandes de l'élargissement. Comme $\mathcal{U}$ est fini, cela ne change pas les propriétés essentielles de l'élargissement. Par contre, cette optimisation permet souvent d'obtenir tout de suite les résultats que fournirait la séquence descendante, et fournit même souvent aussi des résultats plus précis.

4.4 Un exemple d'analyse

Revenons à l'exemple de la voiture présenté au §2.3.3, légèrement simplifié pour ne faire apparaître que les aspects numériques. A chaque état de l'automate on va associer

un polyèdre (voir figure 4.9), comme indiqué au §3.3.

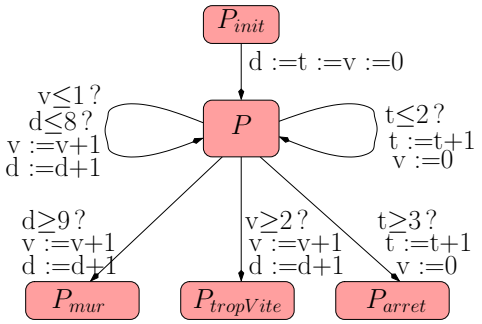


FIG. 4.9 – Exemple de la voiture avec un polyèdre par état

Le système d'équations abstrait associé à cet automate est le suivant :

$$\begin{array}{lcl} P_{init} & = & \top \\ P & = & [d := 0][t := 0][v := 0]P_{init} \sqcup \\ & & \quad [v := v + 1][d := d + 1](P \cap (v \leq 2 \wedge d \leq 9)) \sqcup \\ & & \quad\quad [t := t + 1][v := 0](P \cap (t \leq 3) \\ P_{mur} & = & P \cap (d \geq 10) \\ P_{tropvite} & = & P \cap (v \geq 3) \\ P_{arret} & = & P \cap (t \geq 4) \end{array}$$

et P est choisi comme point d'élargissement, qui sera limité par l'ensemble de contraintes distinguées $\{v \leq 2, d \leq 9, t \leq 3\}$ (cf. §4.3.6). En effet, l'état auquel est attaché P est quitté (définitivement) lorsque ces contraintes sont violées, donc elles ont de bonnes chances, a priori, d'être invariantes dans cet état.

La résolution itérative est la suivante (nous nous contenterons de donner les valeurs successives de P, jusqu'à la convergence où la valeur finale de P nous permettra de calculer P_{mur}, $P_{tropvite}$ et P_{arret}) :

1. Tous les polyèdres sont initialisés à la valeur $\bot$. La première itération fournit donc :

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

2. A la deuxième itération, on obtient :

$$\begin{array}{lcl} P^{(2)} & = & P^{(1)} \nabla ((d = 0 \wedge t = 0 \wedge v = 0) \sqcup \\ & & \quad\quad (d = 1 \wedge t = 0 \wedge v = 1) \sqcup \\ & & \quad\quad\quad (d = 0 \wedge t = 1 \wedge v = 0)) \\ & = & P^{(1)} \nabla (0 \leq d \leq 1 \wedge 0 \leq t \leq 1 \wedge t + d \leq 1 \wedge v = d) \end{array}$$

L'élargissement standard fournirait $P^{(2)} = (v = d \wedge 0 \leq t \wedge 0 \leq d)$, mais comme les contraintes distinguées sont satisfaites par les deux opérandes de l'élargissement, l'élargissement limité fournit :

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

3. Le troisième itéré de P est :

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

4. et à la quatrième itération, on a :

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

5. On trouve alors

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

Le calcul a donc convergé vers un point-fixe, qui ne peut donc pas être amélioré par séquence descendante.

Nous pouvons maintenant calculer les derniers polyèdres :

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

et le résultat $P_{mur} = \bot$ prouve bien que la voiture s'arrête avant le mur !

4.5 Bibliothèques existantes

Plusieurs bibliothèques, offrant l'essentiel des opérateurs sur les polyèdres, sont disponibles ; nous présentons les principales dans cette section. Toutes les bibliothèques mentionnées ci-dessous opèrent en nombres rationnels, pour des raisons d'exactitude des calculs.

4.5.1 La PolyLib

Initialisé à l'IRISA [Wil93] au début des années 90, le développement de la Polylib[5] a ensuite bénéficié d'une coopération entre plusieurs équipes, avec surtout comme domaine d'application la parallélisation de programmes, et la conception d'architectures systoliques.

Les opérations offertes sont les opérations usuelles sur les polyèdres ainsi que la possibilité de manipuler des unions explicites de polyèdres qui, contrairement aux enveloppes convexes, ne perdent pas d'informations.

La Polylib offre aussi des algorithmes sophistiqués (par exemple [NR00]) permettant d'étudier les solutions entières des systèmes linéaires.

4.5.2 Librairie NewPolka

L'outil Nbac [Jea00] utilise la bibliothèque NewPolka, qui s'appuie sur une version préliminaire de la Polylib, et sur des optimisations proposées dans [FP95]. Les principales différences avec PolyLib sont les suivantes :

- *Représentation des coefficients* : Le calcul en rationnels, avec réduction au même dénominateur des coefficients des contraintes et des composantes des vecteurs (réduction dont l'influence sur l'efficacité des algorithmes est relevée par [Avi98]), entraîne un accroissement de la taille des coefficients qui, dès que la dimension augmente, provoque des débordements arithmétiques. Pour résoudre le problème, NewPolka permet de choisir la représentation des entiers (32 bits, 64 bits et entiers multi-précision [gnu]).
- *Calcul paresseux* : la bibliothèque ne conserve en général qu'une seule représentation du polyèdre. La conversion vers l'autre représentation tout comme la minimisation sont retardées et ne s'effectuent que si elles sont nécessaires. Lorsque les deux représentations sont présentes alors elles le sont sous une forme minimale.
- *Tri des contraintes et générateurs* : les lignes des matrices sont triées (paresseusement) par ordre lexicographique pour deux raisons. D'une manière générale, ce tri entraîne une accélération dans l'algorithme de conversion. Et, de manière spécifique, il devient plus facile d'éliminer les contraintes identiques, par exemple lors d'une intersection.
- NewPolka permet la prise en compte des inégalités strictes, par rajout d'une dimension supplémentaire [HPR97].

4.5.3 PPL

L'université de Parme a également développé une bibliothèque de polyèdres, la *Parma Polyhedra Library* (ou PPL) qui est disponible sur son site[6]. PPL, au moins dans sa version initiale, est extrêmement proche de la bibliothèque réalisée pour Nbac. La princi-

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

[6] Voir `www.cs.unipr.it/ppl/`

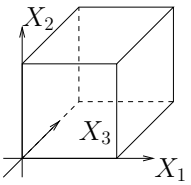
pale différence est le langage utilisé, C++ pour PPL au lieu de C. Les autres différences découlent de manière plus ou moins directe du choix du langage.

PPL est complètement dynamique, aucune taille maximale n'est spécifiée lors de son initialisation comme c'est le cas dans NewPolka.

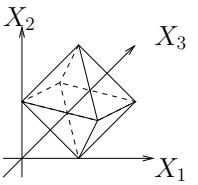
Le traitement des inégalités strictes [BRZH02] est différent de celui de NewPolka. La PPL propose aussi d'autres opérateurs d'élargissement [BHRZ03].

4.6 Le problème de la taille des représentations

Les polyèdres permettent de représenter de manière finie des ensembles infinis, et les algorithmes nécessaires à leur utilisation en interprétation abstraite des programmes sont disponibles et bien connus. Le principal problème est la taille de chaque représentation, qui peut croître exponentiellement avec la dimension de l'espace. Par exemple, un hypercube (cf. Fig. 4.10.a) présente 2^n sommets pour $2n$ contraintes, en dimension n. Inversement et par dualité, un "hyperoctaèdre" (cf. Fig. 4.10.b) a $2n$ sommets et 2^n contraintes.



(a) (hyper-)cube en dimension 3



(b) (hyper-)octaèdre en dimension 3

FIG. 4.10 – Représentations exponentielles

Pour essayer d'éviter l'explosion exponentielle des systèmes générateurs, on présentera au chapitre 7 une autre façon de représenter les polyèdres.

Chapitre 5

Un Analyseur d'Automates Interprétés

5.1 Motivations

L'analyse de relations linéaires est coûteuse, la taille de la représentation des polyèdres pouvant être exponentielle par rapport au nombre de dimensions. On a donc intérêt à pouvoir réduire la dimension des polyèdres manipulés. Pour cela, nous allons considérer deux approches (voir les chapitres 6 et 7). Afin d'évaluer la pertinence de ces solutions, nous devons les mettre en oeuvre dans un analyseur de programmes.

Ce travail prend place dans une recherche à long terme sur la vérification de programmes réactifs écrits dans le langage déclaratif synchrone Lustre [HCRP91]. Un outil d'analyse de relations linéaires des programmes Lustre, Nbac, a fait l'objet de la thèse de Bertrand Jeannet [Jea00, JHR99, Jea03]. Cependant, l'objectif principal de cet outil (voir §5.1.2), est la synthèse d'une structure de contrôle adéquate, à partir d'un programme tel que Lustre, qui n'a aucune notion de structure de contrôle. Il est apparu, dès les premières expérimentations, que la part du coût des opérations sur les polyèdres dans le coût total d'une analyse par Nbac, était trop faible pour mettre clairement en évidence l'influence des optimisations que nous proposons. C'est pourquoi nous avons développé un nouvel analyseur, basé sur la structure de contrôle construite par la compilation de Lustre en automates [HRR91]. C'est cet analyseur que nous décrivons dans ce chapitre.

Auparavant, nous étayons ces motivations par une présentation succincte de Lustre (et de l'expression d'un problème de vérification en Lustre) et de Nbac (§5.1.1, §5.1.2). Nous décrirons ensuite (§5.2) le format d'automates interprétés accepté par notre analyseur, et la façon dont ce format est généré par le compilateur Lustre. Nous présenterons ensuite notre analyseur, en mettant l'accent sur ses aspects originaux.

5.1.1 Le langage LUSTRE

LUSTRE est un langage déclaratif synchrone, pour la programmation des systèmes réactifs. LUSTRE est le socle de l'environnement de programmation SCADE, utilisé dans l'industrie pour la conception de logiciels critiques, par exemple dans l'avionique, l'automobile, l'énergie... La vérification de programmes LUSTRE est donc un enjeu particulièrement important.

Intuitivement, un programme LUSTRE reçoit des *flots* (i.e., des suites, potentiellement infinies, de valeurs typées) d'entrée, et calcule des flots de sortie, en utilisant éventuellement des flots internes (variables locales). Le comportement d'un programme est cyclique, et la n-ième valeur d'un flot représente sa valeur au cycle n. Les flots internes et de sortie sont définis par des *équations*, au sens mathématique : l'ordre des équations est sans signification, et une équation définit une synonymie parfaite entre son membre gauche (le flot défini) et son membre droit (une *expression* de flots). Les expressions de flots sont construites à partir de constantes (représentant des flots de valeurs constantes), de variables de flot, et d'opérateurs. Les opérateurs arithmétiques, logiques, conditionnels standard sont implicitement étendus pour opérer point-à-point sur les flots :

> si $x = (x_0, x_1, \ldots, x_n, \ldots)$ et $y = (y_0, y_1, \ldots, y_n, \ldots)$ sont des flots de valeurs entières, alors "$x + y$" est une expression dénotant le flot $(x_0 + y_0, x_1 + y_1, \ldots, x_n + y_n, \ldots)$

Deux opérateurs temporels, notés "`pre`" et "`->`", sont plus spécifiques :

- L'opérateur unaire "`pre`", "précédent", retarde son opérande d'un cycle :

  > si $x = (x_0, x_1, \ldots, x_n, \ldots)$ est un flot, $\texttt{pre}(x)$ est un flot de même type, et dont la valeur à chaque cycle est définie par
  >
  > $$\forall n \geq 1, \texttt{pre}(x)_n = x_{n-1}, \quad \texttt{pre}(x)_0 \text{ est indéfini}$$

- L'opérateur binaire "`->`", ou "suivi de", permet de changer, ou de définir, la valeur initiale d'un flot : si $x = (x_0, x_1, \ldots, x_n, \ldots)$ et $y = (y_0, y_1, \ldots, y_n, \ldots)$ sont des flots de même type, $x \texttt{->} y$ est un flot dont la valeur à chaque cycle est définie par
  $$(x \texttt{->} y)_0 = x_0, \quad \forall n \geq 1, (x \texttt{->} y)_n = y_n$$

Les programmes sont structurés en noeuds : un noeud est un sous programme qui a ses propres entrées et sorties. Il peut être appelé à l'intérieur d'un autre noeud, ce qui permet d'obtenir une notion de hiérarchie. Dans le cadre d'applications industrielles, les programmes sont constitués de plusieurs modules, ces modules n'ont pas forcément besoin d'être calculés au même rythme. Il est alors possible d'associer aux composants des horloges définissant chacune une période propre. Les composants effectuent le calcul des sorties à la réception d'un top de leur horloge.

La figure 5.1 donne un exemple de noeud LUSTRE qui calcule la somme cumulée des valeurs de son flot d'entrée depuis l'instant initial ou depuis le dernier instant où son entrée "reset" vaut "vrai".

```
node accumulateur(val : int ; reset : bool) returns(acc : int) ;
let
    acc = val -> if(reset) then 0 else pre(acc) + val ;
tel
```

FIG. 5.1 – Exemple de noeud LUSTRE

Un des intérêts du langage LUSTRE, comme de tous les langages synchrones, et de permettre l'expression des propriétés du sûreté au travers de la notion d'*observateur synchrone* [HLR93] : un observateur est un noeud, prenant en entrée les variables influençant la propriété, et calculant une unique sortie booléenne, dont la valeur est mise à "faux" dès que la propriété est violée.

En général, un problème de vérification ne consiste pas simplement à vérifier une propriété : la validité de la propriété dépend d'hypothèses connues sur l'environnement, c'est-à dire sur ses entrées, et les hypothèses peuvent contraindre les entrées en fonction du comportement passé des sorties du programme (c'est le cas lorsque le programme contrôle son environnement). Ces hypothèses doivent être exprimées, comme la propriété, au moyen d'un observateur. Le schéma général d'un programme de vérification est donc celui représenté Fig 5.2.

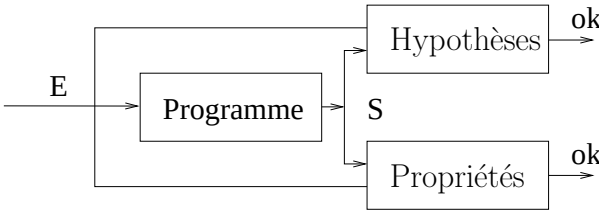


FIG. 5.2 – Forme générale d'un programme de vérification

L'objectif de la vérification est de montrer que, quelles que soient les entrées, aussi longtemps que la sortie de l'observateur des hypothèses est vraie, celle de l'observateur des propriétés est vraie aussi. Cette approche a l'avantage de réduire tout problème de vérification de propriétés de sûreté à un problème d'accessibilité : il s'agit de montrer l'inaccessibilté de certains "mauvais états" : ceux où la propriété est violée alors que les hypothèses ont toujours été vérifiées.

Plusieurs outils ont été développés pour effectuer ce type de vérification :

– LESAR [HLR92] est un "model-checker" dédié à LUSTRE, capable de vérifier des programmes purement booléens, ou des abstractions booléennes de programmes généraux, dont les aspects non booléens (en particulier, les aspects numériques) sont ignorés.

– pour prendre en compte les aspects numériques, l'outil NBAC met en oeuvre une analyse de relations linéaires.

5.1.2 NBAC

Une idée importante dans NBAC [Jea00] est la notion de partitionnement dynamique qui joue sur la structure de contrôle utilisée lors de l'analyse. La structure de contrôle influence directement la précision de l'analyse : en effet, à chaque point de contrôle, on associe une valeur abstraite qui approxime l'ensemble des valeurs des variables numériques en ce point. Plus la structure est détaillée et plus la précision est importante, puisque, lorsque plusieurs points de contrôle sont confondus, leurs valeurs abstraites associées sont composées par enveloppe convexe. Evidemment, inversement, une structure de contrôle trop détaillée conduit à un accroissement de la complexité : résoudre des systèmes abstraits de plusieurs millions d'équations de polyèdres est irréaliste.

Un programme LUSTRE n'a pas de notion évidente de structure de contrôle. NBAC considère d'abord une structure grossière, distinguant les états initiaux (ceux où l'opérateur "`->`" s'évalue comme son premier argument), les états d'erreur (ceux où l'observateur des hypothèses rend "vrai", alors que celui des propriétés rend "faux"), et les autres. Il effectue alors une succession d'analyses "en avant" et "en arrière" (cf. §3.5.1), pour calculer, à chaque point de cette structure de contrôle primitive, les ensembles de valeurs des variables numériques qui sont à la fois accessibles depuis les états initiaux, et co-accessibles depuis les états d'erreur. Si la structure de contrôle s'en trouve coupée, c'est à dire si les états d'erreur sont déconnectés des états initiaux, la vérification a réussi. Sinon, l'outil applique diverses heuristiques pour affiner la structure de contrôle, en distinguant les états clairement différents du point de vue de l'accessiblité/co-accessibilité, ou ceux dont la confusion entraîne clairement une approximation dommageable à la preuve.

Paradoxalement, cette technique de partitionnement, qui rend NBAC performant, en fait un outil peu propice à l'expérimentation des opérations sur les polyèdres. En effet, NBAC passe souvent au moins autant de temps à effectuer le partitionnement (qui est réalisé de manière symbolique à l'aide de BDDs) qu'à réaliser des analyses de relations linéaires. C'est pourquoi, pour pouvoir étudier de manière plus fine nos stratégies de manipulation des ensembles numériques, nous avons développé un nouvel analyseur plus simple.

5.2 Le format d'entrée : OC

Notre analyseur est fondé sur un schéma de compilation de LUSTRE [HRR91], qui a longtemps été utilisé avant d'être abandonné en raison de la taille du code qu'il produisait. Cette technique sera rappelée au §5.2.2, mais nous décrivons d'abord les automates interprétés qu'elle produit, et qui seront les entrées de notre analyseur.

Le format Oc [PS87] (pour *object code*) a été défini pour représenter les automates interprétés générés par les premiers compilateurs des langages synchrones Lustre et Esterel. Un avantage de baser notre analyseur sur ce format est d'ailleurs que l'analyseur devrait pouvoir être utilisé tant pour des programmes Lustre que pour des programmes Esterel[1].

5.2.1 Description de Oc

Nous allons présenter la structure générale d'un automate Oc, sans nous attacher à la syntaxe. Un fichier Oc est organisé de la façon suivante :

- Une série de tables, définissant, pour ce qui nous intéresse, des signaux, des variables, et des actions.
- Un automate.

Les signaux peuvent être d'un type prédéfini (booléens, entiers, chaîne de caractères, ...) ou d'un type défini par l'utilisateur. La définition d'un nouveau type et des fonctions qui le manipulent se fait au moyen de tables (des types ou des fonctions). Pour les types prédéfinis, on dispose déjà d'un certain nombre de fonctions prédéfinies, opérateurs booléens ou arithmétiques standards,... Les signaux correspondent aux entrées et sorties du programme, chaque entrée de la table comporte le nom, la nature (entrée ou sortie), le type et l'indice du signal. L'indice fait référence à la table des variables.

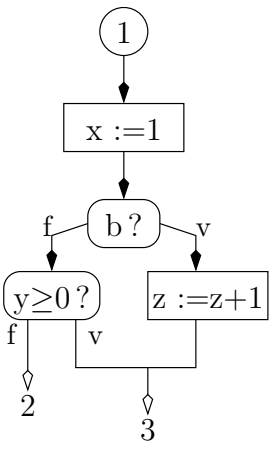
La table des variables comporte toutes les variables du programme : ce sont les entrées/sorties, les variables internes ou des variables spécifiques (comme une mémoire associée à un `pre`). Cette table donne le type de chaque variable.

La table des actions spécifie toutes les actions élémentaires que l'automate peut effectuer lors de ses transitions. Une action élémentaire est une suite de fonctions (prédéfinies ou non) à exécuter en séquence. Les actions qui vont nous intéresser sont les actions conditionnelles (if then else) et les transformations de variables (affectations). D'autres actions (émission ou test de présence d'un signal, ...) sont spécifiques à Esterel.

L'automate est décrit comme une suite d'états : à chaque état est associé une "transition", en fait un *dag* (directed acyclic graph) composé d'actions (faisant référence à la table d'actions). La structure de dag est due à l'ouverture, et à la fermeture éventuelle, d'actions conditionnelles. Chaque branche du dag se termine par une action de branchement à l'état suivant.

[1] Ceci dit, certaines primitives spécifiques à Esterel n'ont pas encore été implantées.

Ainsi, le dag ci-contre représente les transitions issues de l'état "1" : en supposant que b et y sont des variables d'entrée, celles ci sont implicitement lues sur toutes les transitions. Alors,
- si b est faux, et si $y < 0$, une transition met x à 1, et conduit à l'état 2 ;
- si b est faux, et si $y \geq 0$, une transition met x à 1, et conduit à l'état 3 ;
- si b est vrai, une transition met x à 1, incrémente z, et conduit à l'état 3.



Comme il a été dit plus haut, chaque état de l'automate correspond à un nouvel instant de l'exécution du programme Lustre ou Esterel, et par conséquent, en début de chaque transition, toutes les variables d'entrée sont lues et prennent une nouvelle valeur.

5.2.2 Passage de Lustre à Oc

La compilation de Lustre en automates interprétés a été décrite dans [HRR91]. Le principe est le suivant :

- Les variables d'état (mémoire du programme) sont les valeurs retournées par les opérateurs "`pre`" du programme, ainsi que celle d'une variable auxiliaire utilisée pour distinguer le premier cycle de tous les autres (vraie au premier cycle, fausse ensuite).
- Parmi ces variables d'états, certaines sont d'états finis (en particulier, les booléens) et peuvent être codées dans la structure de contrôle : connaissant la valeur v_x d'une variable x au cycle n, on peut spécialiser le code à effectuer au cycle $n+1$, en remplaçant toutes les occurrences de `pre`(x) par v_x. Ce sont ces fragments de code spécialisés qui constituent les états de l'automate.

Prenons un petit exemple. Le "programme" suivant incrémente la variable x sur chaque front montant de l'entrée booléenne b :

```
node compte_fronts (b : bool) returns (x : int) ;
var front : bool ;
let
    x = 0 -> if front then pre(x)+1 else pre(x) ;
    front = false -> (b and not pre(b)) ;
tel
```

La génération de l'automate procède comme suit :

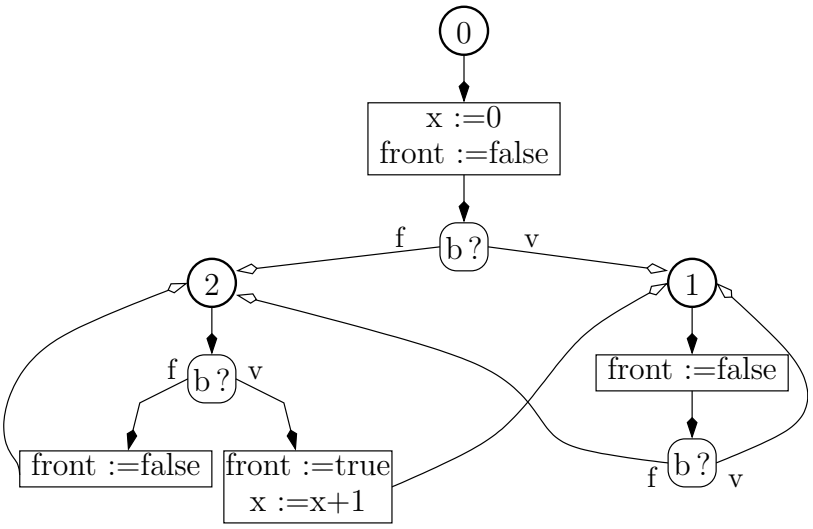
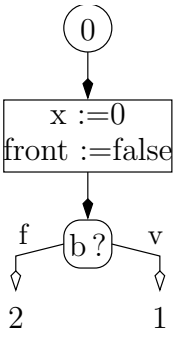


FIG. 5.3 – Automate interprété du compteur de fronts

– Au cycle initial (état 0), les opérateurs "`->`" s'évaluent comme leurs premiers arguments. On a donc : `x=0` ; `front=false`. Si `b` est vrai, "`pre`(b)" sera vrai à l'instant suivant (état 1), et inversement (état 2) si `b` est faux. Les transitions OC correspondant à l'état 0 sont représentées ci contre.



– Dans l'état 1, on sait que le cycle n'est pas initial, et que "`pre`(b)" vaut vrai. Donc, "`front`" est faux, et la valeur de `x` ne change pas. Comme dans l'état 0, l'état suivant (1, ou 2) est choisi selon la valeur de `b`.
– Dans l'état 2, on sait que le cycle n'est pas initial, et que "`pre`(b)" vaut faux. Donc,
  – si `b` est vrai, `front` est vrai aussi, `x` est incrémenté, et le prochain état est l'état 1
  – sinon, `front` est faux, `x` est inchangé, et le prochain état est l'état 2.

L'automate complet est représenté Fig.5.3.

La traduction de LUSTRE en OC est effectuée par le compilateur LUSTRE-V4. Un certain nombre d'options de compilation permettent de jouer sur la structure de contrôle obtenue. Ces options permettent de

– choisir les variables booléennes qui sont traduites dans la structure de contrôle, les autres restant traitées comme des données ;
– choisir la stratégie d'ouverture et de fermeture des test dans les transitions, entre une version minimale, où seuls les branchements au prochain état font l'objet d'actions de test (les autres restant des expressions conditionnelles), et une version maximale, où tous les test sont ouverts, et jamais refermés (le dag est alors un arbre de transitions).

Ces options conduisent à une structure de contrôle plus où moins complexe, tant en nombre d'états qu'en complexité des transitions Oc, ce qui, en ce qui nous concerne, permettra d'ajuster le compromis entre précision et complexité de l'analyse.

Par ailleurs, l'automate obtenu peut être minimisé par bisimulation (en confondant les états qui correspondent au même code), soit à la compilation, soit au moyen de l'outil Ocmin. Cette minimisation, en général peu coûteuse, est évidemment souhaitable avant toute application de notre analyseur de code Oc.

5.3 L'Analyseur

A partir d'un fichier Oc, on génère un automate sur lequel on effectue une analyse des variables numériques. Notre but reste de tester des méthodes simplifiant la manipulation des ensembles numériques.

5.3.1 Automate : états et transitions

L'automate Oc est repris dans un format interne qui conserve sa structure. On crée un état interne pour chaque état Oc et on lui associe une liste d'actions qui correspond aux actions Oc.

On associe également plusieurs polyèdres à un état :

- Un polyèdre P_{in} qui représente l'ensemble des valeurs possibles à l'entrée dans l'état.
- Un polyèdre P_{succ_i} par état successeur qui représente les valeurs que le successeur peut recevoir de l'état courant (fin de transition).
- Un polyèdre P_{widen} qui sert à l'élargissement (voir 5.3.2).

Initialement tous ces polyèdres sont vides à l'exception de P_{in} dans l'état initial qui est le polyèdre univers. On parcourt alors l'automate pour mettre à jour les différents polyèdres de chaque état.

La liste d'actions associée à un état comporte toutes les actions à effectuer avant de passer dans un successeur, c'est donc l'ensemble des transitions sortantes. Ces transitions indiquent comment P_{in} est transformé en P_{succ_i}. On note E1(P) le polyèdre obtenu en transformant le polyèdre P par la liste d'actions E1.

Comme chaque état correspond à un nouvel instant, il faut en entrant dans un état "oublier" (par quantification existentielle, cf. §4.3.4) toutes les informations dont on dispose sur les entrées. Puis, on parcourt la liste des actions afin de transformer P_{in}.

Si l'action est simple ($x := 4$, $x := 3y + z - 2 \ldots$), on effectue la transformation correspondante sur P_{in}.

Les choses sont plus complexes avec un test de la forme $(if(C)\ then\ E1\ else\ E2)\ E3$ avec C une condition. Si la condition porte sur des variables numériques (exemple $2x \leq z - 3$), il faut l'ajouter dans la branche vraie et ajouter sa négation dans la branche fausse. On note P^+ l'intersection de P et de C vraie et P^- l'intersection de P et de C fausse. Si la condition est purement booléenne, on ne gagne pas d'information,

dans ce cas $P = P^+ = P^-$. Suivant que le test se referme ou non et qu'il comporte ou non un saut, on obtient différents polyèdres. On a les cas suivants (Fig.5.4) :

- Le test se referme : on calcule E1(P^+), résultat de la branche "then" et E2(P^-), résultat de la branche "else". Le résultat final est E3(E1(P^+) $\sqcup$ E2(P^-)).
- Si une branche se termine par un saut vers l'état i, disons la branche vraie, alors E1(P^+) est transmis à l'état i E1(P^+) et on conserve en sortie de test E3(E2(P^-)).
- Si le test ne se referme pas, on transmet deux nouveaux polyèdres : E3(E1(P^+)) et E3(E2(P^-)) aux états successeurs.

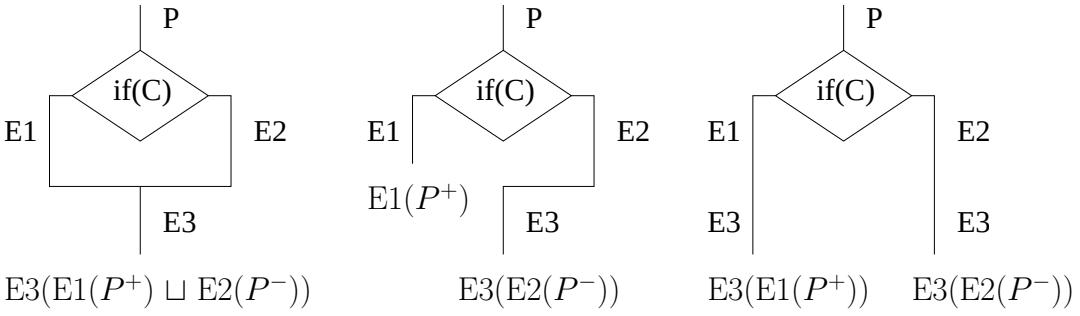


FIG. 5.4 – Cas possibles de polyèdres en sortie d'un test

A la fermeture d'un test, l'enveloppe convexe effectuée fait perdre de l'information. C'est pourquoi la stratégie de génération d'automates OC qui ne referme pas les tests conduit à des résultats plus précis.

Une fois la liste des actions complètement parcourue, on met à jour les successeurs. Plusieurs cas peuvent se produire selon que l'état a déjà été visité et selon que le polyèdre à mettre à jour est vide. Si le polyèdre à modifier est vide, on le remplace par le P_{succ_i} correspondant, si il est non vide, on le remplace par son enveloppe convexe avec P_{succ_i}. Si l'état n'a pas encore été visité, on modifie P_{in}, sinon on modifie P_{widen}.

5.3.2 Elargissement

Choix des points d'élargissement

Le problème du choix des points d'élargissement est de sélectionner un ensemble (minimal) d'états de l'automate, de sorte que toute composante fortement connexe du graphe d'états soit coupée par au moins un état de l'ensemble. Le problème de la coupure minimale étant NP-complet, il est classique d'appliquer l'heuristique des *sous-composantes fortement connexes* proposé par F. Bourdoncle [Bou92].

Cet algorithme est basé sur l'algorithme de Tarjan [Tar72] pour rechercher les composantes fortement connexes (CFC) d'un graphe. Cet algorithme effectue une recherche en profondeur d'abord, et fournit la liste des CFC ainsi qu'un *point d'entrée* de chaque CFC, qui est le premier état de la CFC rencontré deux fois au cours du parcours. Sur l'exemple de la figure 5.5 on a une décomposition en 3 CFC :

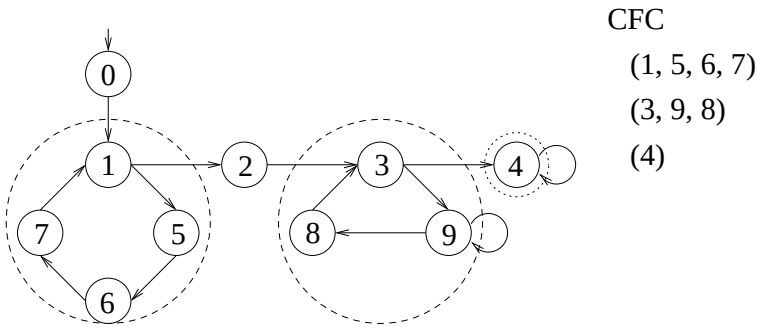


FIG. 5.5 – Automate avec ses CFC

Les points d'entrée sont de bons candidats pour déconnecter les CFC. Cependant, lorsque ces points ont été retirés du graphe, chacune des anciennes CFC peut encore contenir des sous-graphes fortement connexes. Bourdoncle propose donc d'itérer le processus en réappliquant l'algorithme de Tarjan à chacune des CFC, en retirant leurs points d'entrées, etc. jusqu'à obtenir un graphe sans sous-graphe fortement connexe. Tous les points retirés au cours de l'algorithme constituent les points d'élargissement. La figure 5.6 montre les points d'élargissements trouvé sur l'exemple précédent. Notons que l'ensemble trouvé, $\{1, 3, 4, 9\}$, n'est pas minimal, puisque $\{1, 4, 9\}$ déconnecte aussi le graphe.

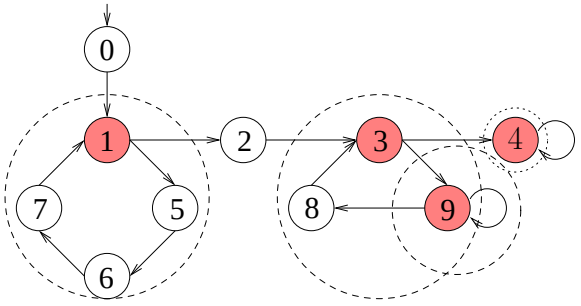


FIG. 5.6 – Sous-composantes fortement connexes et points d'élargissement

Analyse avec élargissement

A chaque point d'élargissement est associé un polyèdre spécifique, P_{widen}, qui sert à mémoriser le résultat de la fonction sans élargissement. Pour ces points, le calcul du polyèdre associé, P_{in}, pendant la phase de calcul de la séquence croissante, est alors : $P_{in} = P_{in} \nabla (P_{in} \sqcup P_{widen})$, tant que $P_{widen} \not\in P_{in}$.

5.3.3 Elargissement limité

Nous avons déjà mentionné (§4.3.6) et illustré (§4.4) la technique d'élargissement limité : lorsque certaines contraintes (un ensemble de contraintes $\mathcal{U}$) ont de bonnes chances d'être invariantes, on définit l'élargissement limité par $\mathcal{U}$, $\nabla_{\mathcal{U}}$ par

$$P\nabla_{\mathcal{U}}Q = P\nabla Q \cap \bigwedge\{c \in \mathcal{U} | P \models c \wedge Q \models c\}$$

c'est-à-dire qu'on garde, pour le polyèdre élargi, les contraintes de $\mathcal{U}$ qui sont satisfaites par les deux opérandes. Cette technique — qui peut améliorer sensiblement la précision de l'analyse, et dans certains cas éviter la séquence descendante (§3.2.3) — a été initialement proposée dans [HPR97], mais le choix de $\mathcal{U}$ dans cet article n'est explicité que dans le cas des boucles simples : lorsqu'une transition $s \stackrel{g:a}{\rightarrow} s$ boucle sur un point d'élargissement s, alors que la négation de la garde g fait quitter la composante fortement connexe de s, la postcondition de g par a a de fortes chances d'être invariante en s. Ici, nous généralisons cette idée : pour toute transition $\stackrel{g:a}{\rightarrow} s$ menant à un point d'élargissement s, toutes les contraintes de la postcondition de g par a sont mises dans l'ensemble $\mathcal{U}$ qui limite l'élargissement en s. Ce choix est intuitivement raisonnable, et d'un coût (en termes du cardinal de $\mathcal{U}$) acceptable, selon nos expérimentations.

Chapitre 6

Utilisation des équations linéaires

Dans ce chapitre, nous examinons une première solution pour réduire le nombre de dimensions au cours des manipulations de polyèdres. Cette solution consiste à détecter d'abord les *équations linéaires* (ou affines) invariantes en chaque point du programme, puis à utiliser ces équations pour éliminer des variables (en "tirant" une variable par équation). Pour détecter les équations, nous utilisons la méthode proposée par Mike Karr dans [Kar76], que nous rappelons d'abord (§6.1). Ensuite (§6.3), nous modifions l'analyseur, pour appliquer d'abord la synthèse d'équations, avant de nous servir des résultats pour l'analyse d'équations. Enfin, nous combinerons plus étroitement les deux analyses au §6.4.

6.1 Le treillis des variétés linéaires [Kar76]

Définition 6.1 (Variété linéaire)
Rappelons qu'une variété linéaire *(ou affine) de $\mathcal{N}^n$, est un sous-ensemble de points de $\mathcal{N}^n$ satisfaisant un système d'équations linéaires. Un tel système d'équations linéaires sera représenté matriciellement :*

$$AX = B, \text{ ou encore } AX - B = 0$$

où A est une matrice à m lignes et n colonnes, et B est un m-vecteur de $\mathcal{N}^n$. Posons $A' = [A, -B]$, donc nous représentons une variété linéaire par une matrice $m \times (n+1)$.

L'ensemble des variétés linéaires de $\mathcal{N}^n$ forme un treillis (ordonné par inclusion) *de profondeur finie* : l'intersection de deux variétés linéaires est une variété linéaire ; étant données deux variétés linéaires, il existe une plus petite variété linéaire (leur *enveloppe linéaire*) qui les contient ; la plus petite (resp., plus grande) variété linéaire de $\mathcal{N}^n$ est l'ensemble vide (resp., l'ensemble $\mathcal{N}^n$ tout entier). Enfin, si une variété linéaire est strictement incluse dans une autre, la seconde est de dimension strictement plus grande que la première ; il s'ensuit que tout chaîne strictement croissante de variétés linéaires est de longueur bornée par $n + 1$.

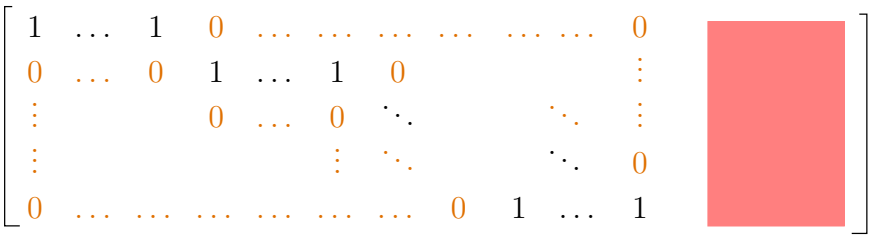


FIG. 6.1 – Forme en échelon réduit d'une matrice

6.1.1 Minimisation et forme canonique

La minimisation d'un système d'équations est classique : on exprime chaque variable v, une à une, en fonction des autres, en remplaçant v par son expression dans les autres équations. Après remplacement d'au maximum n variables, on obtient, en plus des équations tirées, des équations de constantes, qui sont soit triviales, soit contradictoires (auquel cas la variété est vide). Nous illustrons cet algorithme sur deux petits exemples :

Exemple 1 : Considérons le système d'équations :

$$\begin{array}{rrrcl} x & & +2z & = & 1 \\ & y & -z & = & 0 \\ 2x & +y & +3z & = & 2 \end{array}$$

On tire $x = 1 - 2z$ de la première équation et $y = z$ de la deuxième, et l'on reporte dans la troisième, et on obtient $0 = 0$. La variété linéaire est donc non vide, et son système minimal est :

$$\begin{array}{rrcl} x & +2z & = & 1 \\ y & -z & = & 0 \end{array}$$

En termes de matrices, on a mis la matrice initiale sous forme *d'échelon réduite* (cf. Figure 6.1) :

$$\begin{bmatrix} 1 & 0 & 2 & -1 \\ 0 & 1 & -1 & 0 \\ 2 & 1 & 3 & -2 \end{bmatrix} \longrightarrow \begin{bmatrix} 1 & 0 & 2 & -1 \\ 0 & 1 & -1 & 0 \\ 0 & 0 & 0 & 0 \end{bmatrix}$$

Exemple 2 : Par contre, partant du système :

$$\begin{array}{rrrcl} x & & +2z & = & 2 \\ & y & -z & = & 0 \\ 2x & +y & +3z & = & 2 \end{array}$$

on obtient $x = 2 - 2z$ et, comme précédemment, $y = z$, ce qui, reporté dans la troisième équation, fournit $2 = 0$, la variété linéaire est vide.

La forme en échelon réduit est non seulement minimale, mais, si on normalise à 1 la dernière colonne (correspondant aux constantes) elle est aussi canonique, pour un ordre fixé des variables.

6.2 Opérations de manipulations

On va présenter les opérations nécessaires à l'analyse d'un programme. C'est-à-dire les opérations de base : une transformation, une intersection et une union ainsi que les tests : vide, univers et conditionnel.

6.2.1 Transformation

On s'intéresse à une affectation linéaire qui transforme (A_e, B_e) en (A_s, B_s). On cherche à obtenir un maximum d'informations sur (A_s, B_s) grâce à (A_e, B_e) et à la transformation. On distingue 2 cas selon que l'affectation est inversible ou non.

Affectation inversible

On suppose que l'affectation est de la forme $X_{j_0} \leftarrow \sum_j \alpha_j X_j + \beta$ avec $\alpha_{j_0} \neq 0$.

On peut démontrer que, dans ce cas, on a :

- $A_{s\ i,j} = A_{e\ i,j} - (A_{e\ i,j_0}/\alpha_{j_0})\alpha_j$ pour tout $j \neq j_0$ et pour tout i.
- $A_{s\ i,j} = A_{e\ i,j_0}/\alpha_{j_0}$ pour tout i.
- $B_{s\ i} = B_{e\ i} + (A_{e\ i,j_0}/\alpha_{j_0})\beta$ pour tout i.

Affectation non inversible

Si l'affectation de la variable X_{j_0} est non inversible alors on a une perte d'information. En effet les égalités faisant intervenir X_{j_0} ne sont plus vérifiées. Imaginons que l'on avait $X_{j_0} = X_k$ et que l'on effectue $X_{j_0} = 3$, il n'y a aucune raison pour que $X_{j_0} = X_k$ reste valide.

Une affectation non inversible se traite en deux temps. Dans un premier temps on transforme la matrice pour avoir au plus une ligne dont le coefficient de la variable X_{j_0} est non nul puis on supprime cette ligne. Dans un deuxième temps on rajoute l'égalité qui correspond à l'affectation.

Affectation non linéaire

Si l'affectation est non linéaire, par exemple $X_{j_0} = X_{j_0} * X_k$, on a une perte d'information. Dans ce cas on procède comme pour la première étape d'une affectation non inversible afin de perdre toute les informations disponibles sur X_{j_0}.

6.2.2 Intersection

L'intersection de (A, B) et de (A', B') se fait en ajoutant les matrices correspondantes. Les deux ensembles étant donnés sous forme canonique, on peut syntaxiquement identifier les égalités présentes dans les deux ensembles et ne garder qu'une seule fois l'égalité. Mais, même en éliminant ainsi les égalités trivialement redondantes, le résultat n'est pas forcément sous forme canonique. Après une intersection, on minimise le résultat afin d'en obtenir une forme canonique. Si l'intersection est vide alors la minimisation fait apparaître une égalité où tous les coefficients des variables sont nuls et ou le terme constant est non nul, i.e. on a une égalité de la forme $0 = c$ avec $c \neq 0$. Dans ce cas on peut éliminer toutes les autres égalités.

6.2.3 Union

L'enveloppe affine (notée $\sqcup$) de deux variétés affine est la plus petite variété affine qui les contient. L'enveloppe affine contient ses opérandes (représentés par les matrices A et A'), donc elle ne peut pas avoir d'autres égalités que celles obtenues comme combinaison des opérandes.

Le problème est de déterminer les égalités satisfaites par A et A'. De manière évidente si une égalité est présente dans les deux descriptions alors elle est satisfaite par A et A'. Malheureusement une égalité peut être satisfaite par les deux ensembles sans être présente dans les deux descriptions. C'est le cas dans l'exemple suivant, $A = \{x + y = 2\}$ et $A' = \{x = 1, y = 1\}$. L'égalité $x + y = 2$ bien que satisfaite par A' n'est pas présente dans sa description.

Le calcul de l'union affine consiste en une réécriture des matrices qui vise à les rendre syntaxiquement égales. Les égalités satisfaites par un seul opérande sont supprimées lors de la réécriture. Les égalités présentes dans les deux descriptions sont laissées. Les égalités satisfaites mais présentes sous des formes différentes (comme dans l'exemple précédent) sont rendues présentes dans les deux descriptions par combinaisons linéaires des lignes de la matrice.

Initialement, les matrices sont données sous forme canonique. On note C la matrice qui représente l'union affine et A^S la matrice A lors de l'étape S du calcul. A chaque étape S, les conditions suivantes sont vérifiées :

- $A^S \sqcup A'^S = A \sqcup A'$
- C^S a S colonnes.
- A^S, A'^S et C^S sont sous forme canonique.
- $A^S = \left(\begin{array}{c|c} C^S & A_0^S \end{array} \right)$ $A'^S = \left(\begin{array}{c|c} C^S & A'^S_0 \end{array} \right)$ avec A_0^S qui représente les colonnes de A^S d'indices supérieurs à S.

Lors de l'étape S, on note r l'indice de la première ligne qui suit C^S. Lors d'une itération, trois cas peuvent se produire :

Cas 1

$A^{S-1}_{rS} = {A'}^{S-1}_{rS} = 1$

$$A^{S-1} = \left(\begin{array}{c|c|c} C^{S-1} & \vdots & \\ \hline 0\dots0 & 1 & A^S_0 \end{array}\right) \quad A'^{S-1} = \left(\begin{array}{c|c|c} C^{S-1} & \vdots & \\ \hline 0\dots0 & 1 & A'^S_0 \end{array}\right)$$

La colonne de 0 vient du fait que les matrices sont sous forme canonique. On définit les matrices suivantes de l'itération par : $A^S = A^{S-1},\ A'^S = A'^{S-1}$ et $C^S = \left(\begin{array}{c|c} C^{S-1} & \vdots \\ \hline 0\dots0 & 1 \end{array}\right)$

Cas 2

$A^{S-1}_{rS} = 1,\ {A'}^{S-1}_{rS} = 0$ (et le cas où le 0 et le 1 sont inversés)

A^{S-1} est comme dans le cas 1 et $A'^{S-1} = \left(\begin{array}{c|c|c} C^{S-1} & \beta & A'^S_0 \end{array}\right)$

Cela couvre aussi le cas où r est plus grand que le nombre de lignes de A'^{S-1} (mais pas plus grand que celui de A^{S-1}, cas 3).

On commence par obtenir β dans A^{S-1} : pour cela, on ajoute à chaque ligne i ($1 \leq i \leq r$) de A^{S-1} la ligne $\beta_i A^{S-1}_i$. Comme A^{S-1}_{rS} est le premier coefficient non nul de la ligne r, la sous matrice C^{S-1} reste inchangée ; par contre les premières lignes de A^{S-1}_0 peuvent avoir changé. La nouvelle matrice A^S est obtenue en supprimant la ligne r de A^{S-1}. Les autres matrices sont définies par $A'^S = A'^{S-1}$ et $C^S = (C^{S-1}|\beta)$.

cas 3

Le dernier cas possible est le suivant : $A^{S-1}_{rS} = {A'}^{S-1}_{rS} = 0$

Il couvre aussi le cas où r est plus grand que le nombre de lignes de A^{S-1} et de A'^{S-1}.

$$A^{S-1} = \left(\begin{array}{c|c|c} C^{S-1} & \alpha & A^S_0 \end{array}\right) \quad A'^{S-1} = \left(\begin{array}{c|c|c} C^{S-1} & \beta & A'^S_0 \end{array}\right)$$

Si $\alpha = \beta$, on a trivialement $A^S = A^{S-1}$, $A'^S = A'^{S-1}$ et $C^S = (C^{S-1}|\alpha)$.

Si $\alpha \neq \beta$, on note t l'indice maximum tel que $\alpha_t \neq \beta_t$. Pour tout i avec $1 \leq i < t$ on soustrait à la ligne i $(\alpha_i - \beta_i)/(\alpha_t - \beta_t)$ fois la ligne t. Le fait que les $S-1$ premières colonnes sont égales est une conséquence de leur égalité dans A^{S-1} et A'^{S-1}. Pour la colonne S, on a pour $1 \leq i < t$:

$$A^S_{iS} = \alpha_i - \left(\frac{\alpha_i - \beta_i}{\alpha_t - \beta_t}\right)\alpha_t = \beta_i - \left(\frac{\alpha_i - \beta_i}{\alpha_t - \beta_t}\right)\beta_t = A'^S_{iS}$$

Les lignes d'indices entre $t+1$ et r ne sont pas modifiées : on conserve l'égalité sur la colonne S. Seule la ligne d'indice t ne peut être rendue identique dans les deux matrices : on la supprime pour passer de A^{S-1} à A^S et de A'^{S-1} à A'^S. C^S est la sous matrice correspondant aux S premières colonnes et $r-2$ premières lignes.

Ces trois cas couvrent tous les cas possibles et fournissent une matrice qui correspond à l'union des deux ensembles. Il est facile de vérifier que la forme canonique est conservée.

Remarque : Le treillis des égalités étant de profondeur finie l'union affine permet d'obtenir une convergence en un nombre finie d'itérations de calcul. Il n'est pas nécessaire de définir un opérateur d'élargissement.

6.2.4 Les tests

Les tests ne sont pas des opérations de bases de la manipulation des égalités mais sont indispensables à l'analyse d'un programme.

Le vide, l'univers

Nous avons vu que la mise sous forme canonique permettait de détecter les systèmes sans solutions (test du vide). L'universalité se traduit par une absence d'équations.

Test conditionnel : if

Si la condition test n'est pas une équation (ou une diséquation) affine, on ne peut rien en déduire. Les valeurs possibles sur les branches vraie et fausse sont identiques à celles en entrée. On suppose que le test est de la forme $\sum_j \alpha_j X_j = \beta$ (le cas d'une diséquation $\sum_j \alpha_j X_j \neq \beta$ est évidemment symétrique), un système à une équation que l'on note (A_{test}, B_{test}).

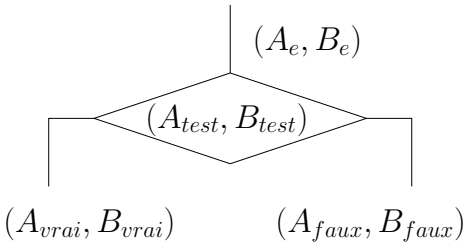


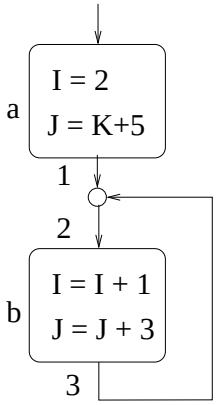
FIG. 6.2 – Test conditionnel avec une égalité

Le contexte d'entrée du test est (A_e, B_e) ; en sortie, on a plusieurs cas possibles en fonction de $(A_e, B_e) \cap (A_{test}, B_{test})$.

- Si $(A_e, B_e) \cap (A_{test}, B_{test}) = \varnothing$, le test est toujours faux, la branche vraie est inaccessible. On a $(A_{vrai}, B_{vrai}) = \varnothing$ et $(A_{faux}, B_{faux}) = (A_e, B_e)$.
- Si $(A_e, B_e) \cap (A_{test}, B_{test}) = (A_e, B_e)$, le test est toujours vrai ; dans ce cas c'est la branche fausse qui est inaccessible. On a $(A_{vrai}, B_{vrai}) = (A_e, B_e)$ et $(A_{faux}, B_{faux}) = \varnothing$.
- Si aucun des deux cas précédents n'est vérifié, alors on en déduit des informations sur la branche vraie, on a $(A_{vrai}, B_{vrai}) = (A_e, B_e) \cap (A_{test}, B_{test})$. Par contre, on ne peut rien en déduire de plus pour la branche fausse, $(A_{faux}, B_{faux}) = (A_e, B_e)$.

6.2.5 Exemple

Nous allons illustrer l'analyse des équations affines sur l'exemple suivant :



Après l'initialisation en a on obtient au point 1 les égalités suivantes : $I = 2$ et $J - K = 2$.

Lors du premier passage au point 2 on obtient les mêmes égalités, après le passage en b on obtient $I = 3$ et $J - K = 8$.

On boucle pour revenir au point 2 et on fait l'enveloppe des informations issues des points 1 et 3. On a :

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & -1 & 5 \end{pmatrix} \cup \begin{pmatrix} 1 & 0 & 0 & 3 \\ 0 & 1 & -1 & 8 \end{pmatrix} = \begin{pmatrix} 3 & -1 & 1 & 1 \end{pmatrix}$$

Après un deuxième passage en b on obtient les mêmes égalités au point 3. Au point 2 on effectue l'enveloppe suivante :

$$\begin{pmatrix} 1 & 0 & 0 & 2 \\ 0 & 1 & -1 & 5 \end{pmatrix} \cup \begin{pmatrix} 3 & -1 & 1 & 1 \end{pmatrix} = \begin{pmatrix} 3 & -1 & 1 & 1 \end{pmatrix}$$

L'analyse s'arrête en concluant qu'au point 2 les variables satisfont $3I - J + K = 1$.

6.3 Pré traitement

Notre objectif étant de réduire la dimension des ensembles numériques manipulés, nous avons cherché à utiliser les résultats d'une analyse préalable des équations affines, afin d'éliminer certaines variables.

6.3.1 Première passe

Nous effectuons donc, dans notre analyseur, une première passe d'analyse des équations affines, qui se situe entre la transformation du fichier Oc et l'analyse en polyèdres. Cette passe réutilise une grande partie du mécanisme de l'analyse : elle commence avec un état initial non contraint et tous les autres états vides et propage les valeurs accessibles.

Comme les éléments manipulés lors de l'analyse sont des équations, on ignore les inégalités rencontrées. Clairement, il est possible que certaines équations ne soient pas trouvées. C'est le cas des équations qui résultent de deux inégalités opposées (ex. $x \leq 0$ et $x \geq 0$).

6.3.2 Elimination des variables

Une fois l'analyse des équations effectuée, on obtient, pour chaque état, un ensemble d'équations vérifiées dans l'état. On va s'en servir pour éliminer des variables avant d'effectuer certaines opérations. On cherche à éliminer des variables lors des opérations coûteuses.

En particulier, on va chercher à simplifier les ensembles lors des enveloppes convexes. Cette opération est réalisée dans deux cas, lors de la fermeture d'un test et lors du saut vers un état.

Nous avons choisi, dans cette première approche, de ne pas appliquer l'optimisation à la fermeture des tests : en effet, les systèmes d'équations issus de la première passe sont associés aux états. Pour obtenir l'information nécessaire pour en bénéficier à la fermeture des tests, il faudrait propager ces équations à nouveau le long des transitions. Cette propagation n'est pas difficile, mais n'a pas été implémentée dans le prototype. En fait, cette approche globale donnant des résultats médiocres, en raison du faible nombre d'équations invariantes sur les exemples que nous avions, nous avons plutôt fait porter nos efforts sur l'approche pas-à-pas, que nous présenterons dans la section 6.4.

Dans le cas d'une transition vers un état, on dispose de l'ensemble des égalités vérifiées par le résultat — et donc aussi à la fin de toutes les transitions atteignant cet état. Les équations étant satisfaites à la fois par tous les opérandes de l'enveloppe et par son résultat, il est correct et facile d'éliminer une variable par équations dans chacun des opérandes, d'effectuer l'enveloppe dans l'espace ainsi réduit, puis d'ajouter les équations au résultat.

6.4 Approche pas à pas

Comme il a été dit plus haut, le pré-traitement présenté au paragraphe précédent produit des résultats expérimentaux décevants : sur des exemples réels, le nombre d'équations invariantes dans chaque état est faible ou nul, et chaque équation ne permettant d'éliminer qu'une variable, les bénéfices sont négligeables. Il arrive même souvent

que les performances de l'analyseur soient pénalisées par le pré-traitement des équations, le coût de l'analyse préalable des équations excédant le bénéfice de leur usage dans l'analyse en polyèdres. C'est pourquoi nous proposons maintenant d'entrelacer l'analyse des équations et l'analyse en polyèdres, en partant de la constatation que beaucoup d'équations ne sont satisfaites qu'en début d'analyse, et sont perdues ensuite. Ainsi, les équations satisfaites à la n-ième itération de l'analyse des équations peuvent être utilisées pour optimiser la n-ième itération de l'analyse en polyèdres, même si ces équations ne sont pas des invariants globaux.

L'idée est donc, lors de toute opération coûteuse sur les polyèdres, de déterminer d'abord les équations affines satisfaites par les opérandes et par le résultat de l'opération, et d'éliminer une variable par équation avant d'effectuer l'opération dans un espace réduit. Nous nous intéressons ici aux cas de l'intersection et de l'enveloppe convexe.

Dans la description d'un polyèdre sous forme de contraintes, il est facile de récupérer l'ensemble des équations affines qu'il satisfait. Notons E_P et E_Q les ensembles d'équations satisfaites par P et Q. Les équations affines E_{PopQ} satisfaites par le résultat de l'opération s'obtiennent, dans le cas d'une intersection, par intersection (normalisée) des variétés affines, et dans le cas d'une enveloppe convexe, par enveloppe affine. Notons que, comme précédemment, E_{PopQ} ne représente en général qu'un sous-ensemble des équations satisfaites par le résultat (cf. §6.3.1). Par contre, il se peut que, comme E_P et E_Q sont issus de l'analyse en polyèdres, ils contiennent des équations qui ne seraient pas trouvées par une simple analyse en équations.

Dans P et Q, on élimine alors une variable par équation présente dans E_{PopQ} avant d'exécuter l'opération dans l'espace réduit, et on ajoute les équations E_{PopQ} au résultat (les systèmes générateurs étant faciles à étendre en utilisant les équations).

Cette approche oblige à recalculer Eq à chaque opération mais elle permet d'utiliser les équations qui ne sont vraies que de manière ponctuelle pour simplifier les calculs. Elle est aussi plus simple, puisqu'elle consiste en une sur-couche au dessus de la bibliothèque de polyèdres, traitant les équations séparément, autant que faire se peut. En définitive, nous n'utilisons de la méthode de Mike Karr que les algorithmes de mise sous forme canonique et d'enveloppe affine.

Chapitre 7

Produits Cartésiens de Polyèdres

Dans ce chapitre, nous allons chercher à diminuer la dimension des polyèdres manipulés dans l'analyse de relations linéaires, en tirant parti du fait que certains polyèdres peuvent être en fait des *produits cartésiens* de polyèdres en dimension plus petite. Ce phénomène apparaît relativement souvent dans les programmes réels, composés de parties relativement indépendantes, et où certaines variables ne sont par corrélées. Notons aussi que ce phénomène est plus fréquent à la fin de l'analyse, lorsque, en raison principalement des enveloppes convexes et des élargissements, des relations liant les variables entre-elles ont tendance à disparaître. Remarquons enfin que cette indépendance des variables pourrait être forcée, en oubliant certaines relations lorsque l'analyse devient trop complexe.

Une telle factorisation des polyèdres sous forme de produits cartésiens est particulièrement prometteuse, puisqu'elle peut mener à une réduction logarithmique des coûts : en effet, le système générateur d'un hypercube ($a_i \leq x_i \leq b_i, i = 1 \ldots n$) est de taille exponentielle en n (2^n sommets), mais linéaire lorsqu'on le considère comme un produit cartésien d'intervalles.

Nous présenterons en 7.1 les notations et les notions de base que l'on utilisera dans ce chapitre. Puis nous verrons comment se comportent les opérations sur les produits cartésiens (§7.2 à §7.2.3). Nous traiterons en dernier le cas de l'enveloppe convexe qui est une opération coûteuse nécessitant un traitement spécifique (voir 7.3).

7.1 Introduction

A partir d'une bibliothèque de polyèdres (la PPL [PPL]), nous voulons créer une nouvelle bibliothèque capable de détecter les produits cartésiens et de les traiter de manière spécifique. Nous partons d'une bibliothèque existante pour ne pas avoir à réaliser les opérations de base. Nous nous intéresserons uniquement à l'expression des opérations spécifiques au produit. Cette approche sous forme de produits cartésiens a en partie était décrite dans [HMPV03].

7.1.1 Notations et définitions

Les composantes d'un produit cartésien sont des polyèdres portant sur un sous-ensemble des variables. Il nous faut donc être capables de dénoter et de manipuler de tels polyèdres. Pour ce faire, nous introduisons quelques notations et opérations.

Les polyèdres globaux, inclus dans l'espace $\mathcal{N}^n$, portent sur un ensemble de variables $V = \{x_1, x_2, \ldots, x_n\}$. Nous allons avoir à considérer des polyèdres de dimension plus petite, chacun portant sur un sous-ensemble $I \subset V$ de variables. On notera $P^{(I)}$ un polyèdre portant sur les variables de I, et on dira que $P^{(I)}$ est un polyèdre sur I.

Définition 7.1 (Partition)
Une partition σ de V est un ensemble $\{I_1, I_2, \ldots I_k\}$ de parties de V, tel que

$$(i \neq j \;\Rightarrow\; I_i \cap I_j = \emptyset) \quad et \quad \bigcup_{i=1}^{k} I_i = V$$

Les éléments d'une partition seront appelés "classes".

Définition 7.2 (Projection, Extension)
Soient I, J deux parties de V telles que $I \subset J$.
- *Si $P^{(J)}$ est un polyèdre sur J, et si $X \in P^{(J)}$, on appelle projection de X sur I, et on note $X{\downarrow}I$ le vecteur des composantes de X qui correspondent aux variables de I.*
- *Si $P^{(J)}$ est un polyèdre sur J, on appelle projection de $P^{(J)}$ sur I, et on note $P^{(J)}{\downarrow}I$ le polyèdre*

$$P^{(J)}{\downarrow}I = \{X{\downarrow}I \mid X \in P^{(J)}\}$$

- *Inversement, si $P^{(I)}$ est un polyèdre sur I, on appelle extension de $P^{(I)}$ à J, et on note $P^{(I)}{\uparrow}J$, le polyèdre sur J*

$$P^{(I)}{\uparrow}J = \{X \mid X{\downarrow}I \in P^{(I)}\}$$

On notera simplement $P^{(I)}{\uparrow}$ l'extension de P à l'espace tout entier.

Remarque : Si $I \subset J$ et si $P^{(I)}$ (resp., $P^{(J)}$) est un polyèdre sur I (resp., sur J), alors

$$(P^{(I)}{\uparrow}J){\downarrow}I \;=\; P^{(I)} \text{ et } P^{(J)} \subseteq (P^{(J)}{\downarrow}I){\uparrow}J$$

Calcul de la projection : La projection $P^{(I)}$ de $P^{(J)}$ se calcule aisément comme suit :
- le système de contraintes de $P^{(I)}$ est extrait de celui du $P^{(J)}$ en ne gardant que les colonnes correspondant à des indices dans I, et en effaçant les lignes nulles du résultat ;

– chaque élément du système générateur de $P^{(I)}$ est extrait d'un élément correspondant du système générateur de $P^{(J)}$, en ne gardant que les composantes correspondant à des indices dans I.

Les représentations obtenues ne sont évidemment pas forcément minimales.

Définition 7.3 (Produit cartésien)

Un polyèdre P dans $\mathcal{N}^n$ est un produit cartésien s'il existe une partition $\sigma = \{I_1, I_2, \ldots I_k\}$ de V (non réduite à un élément), telle que

$$P = P{\downarrow}I_1 \times P{\downarrow}I_2 \times \ldots \times P{\downarrow}I_k$$

On dira alors que P est factorisé selon σ. Evidemment, plus la partition est grande (plus k est grand) plus la factorisation est intéressante pour nous. Evidemment aussi, si P est factorisable, il existe une unique plus grande partition sur laquelle il est factorisable.

7.1.2 Factorisation

Un premier problème est la détection qu'un polyèdre est factorisable, et sa factorisation effective. Par ailleurs, la plupart des opérations ne pourront tirer parti de la factorisation que si leurs opérandes sont factorisés selon la même partition. Nous étudierons donc aussi la détermination de la plus grande partition factorisant un ensemble de polyèdres.

Factorisation d'un polyèdre

La factorisation d'un polyèdre P s'effectue à partir de son système de contraintes : deux variables sont liées, si elles apparaissent avec des coefficients dans une même contrainte. Cette relation est évidemment réflexive et symétrique, mais la relation, notée $\leftrightarrow_A^*$, qui nous intéresse est sa fermeture transitive. Les classes d'équivalence de $\leftrightarrow_A^*$ définissent la partition σ des variables. Si le système de contraintes est minimal, σ est la partition maximale.

L'algorithme de détermination de σ consiste à construire les classes d'équivalence de $\leftrightarrow_A^*$, en examinant les contraintes une à une : initialement, à chaque variable x_i est associée une classe $I_i = \{i\}$ réduite à son indice. Chaque fois que deux variables x_i et x_j sont trouvées liées dans une contraintes, leurs classes sont fusionnées :

$$I_i \ := I_j := I_i \cup I_j$$

Il s'agit donc d'une adaptation directe du célèbre algorithme "Union-Find" [AHU74]. Nous y reviendrons au §9.1.1.

Une fois la partition $\sigma = \{I_1, \ldots, I_k\}$ déterminée, et si celle-ci comporte plus d'une classe ($k > 1$), le polyèdre est factorisé en $P^{(I_1)} \times \ldots \times P^{(I_k)}$, avec $P^{(I_i)} = P{\downarrow}I_i$.

Partition commune à σ_1 et σ_2

Pour pouvoir effectuer une opération sur des polyèdres factorisés, il faut que leur factorisation soit la même. Or, pour un polyèdre de dimension strictement supérieure à 2, il peut exister plusieurs factorisations possibles.

Si les polyèdres n'ont pas la même factorisation, il faut calculer la plus petite partition commune. Pour cela, si une classe est présente dans les 2 partitions, elle est conservée dans la partition commune. Si une classe I n'apparaît que dans une partition, on construit dans σ_1 et dans σ_2 la plus petite union de classes qui contient I et qui contient les indices qui empêchaient I d'être commune. On recommence jusqu'à ce que la classe construite soit présente dans σ_1 et σ_2.

pour chaque classe I de σ_1 faire :
  tant que I n'est pas égale à une classe de σ_2 faire :
    $I' = \emptyset$
    pour toute classe I_i de σ_2 si $I \cap I_i \neq \emptyset$
      $I' = I' \cup I_i$
      $\sigma_2 = \sigma_2 - I_i$
    $\sigma_2 = \sigma_2 + I'$
    pour toute classe I_k de σ_1 telle que $I' \cap I_k \neq \emptyset$
      $I = I \cup I_k$
      $\sigma_1 = \sigma_1 - I_k$

A la fin, les partitions σ_1 et σ_2 sont égales. De plus, en triant selon l'ordre lexicographique, on fait en sorte que les classes aient des indices identiques dans les deux partitions.

7.1.3 Décomposition du résultat d'une opération

Nous allons définir les opérations classiques sur les produits cartésiens de polyèdres, en supposant dorénavant que les opérandes sont factorisés selon la même partition. Plusieurs cas se présentent, selon qu'on peut prévoir la factorisation du résultat :

- La factorisation des opérandes ne change pas.
- Le résultat du calcul a une factorisation au moins aussi fine que celle des opérandes.
- La factorisation du produit ne peut pas être prévue à l'avance car elle peut être plus fine, identique ou moins fine que la factorisation des opérandes.

7.2 Opérations simples

La plupart des opérations (tests, intersection, application affine, élargissement) s'adaptent sans difficulté aux produits cartésiens de polyèdres.

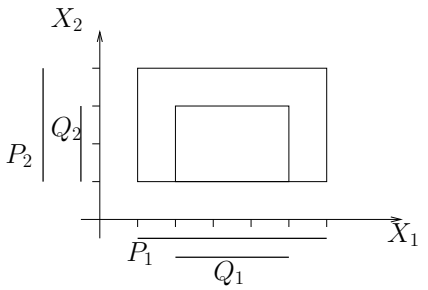


FIG. 7.1 – Inclusion de polyèdres factorisés

7.2.1 Test du vide

Le test du vide se fait en testant les composantes du produit. Un produit de polyèdres est vide si et seulement si au moins une de ses composantes est vide.

7.2.2 Test d'inclusion

Pour l'inclusion d'un produit de polyèdres $P = P_1 \times P_2$ dans $Q = Q_1 \times Q_2$ on a : $P \subseteq Q \Leftrightarrow P_1 \subseteq Q_1$ *et* $P_2 \subseteq Q_2$

Démonstration:

$$
\begin{aligned}
&(P{\downarrow}I_1 \times P{\downarrow}I_2) \subseteq (Q{\downarrow}I_1 \times Q{\downarrow}I_2) \\
&\quad \Leftrightarrow\ (X \in (P{\downarrow}I_1 \times P{\downarrow}I_2) \Rightarrow X \in (Q{\downarrow}I_1 \times Q{\downarrow}I_2)) \\
&\quad \Leftrightarrow\ ((X_1, X_2) \in (P{\downarrow}I_1 \times P{\downarrow}I_2) \Rightarrow (X_1, X_2) \in (Q{\downarrow}I_1 \times Q{\downarrow}I_2)) \\
&\quad \Leftrightarrow\ (X_1 \in P{\downarrow}I_1 \wedge X_2 \in P{\downarrow}I_2 \Rightarrow X_1 \in Q{\downarrow}I_1 \wedge X_2 \in Q{\downarrow}I_2) \\
&\quad \Leftrightarrow\ (X_1 \in P{\downarrow}I_1 \ \Rightarrow X_1 \in Q{\downarrow}I_1) \wedge (X_2 \in P{\downarrow}I_2 \ \Rightarrow X_2 \in Q{\downarrow}I_2) \\
&\quad \Leftrightarrow\ (P{\downarrow}I_1 \subseteq Q{\downarrow}I_1) \wedge (P{\downarrow}I_2 \subseteq Q{\downarrow}I_2)
\end{aligned}
$$

Le résultat se généralise trivialement à une factorisation en n polyèdres.
□

Exemple : La Figure 7.1 illustre le test d'inclusion de $P = \{1 \leq X_1 \leq 6, 1 \leq X_2 \leq 4\}$ dans $Q = \{2 \leq X_1 \leq 5, 1 \leq X_2 \leq 3\}$, avec $\sigma = \{\{X_1\}, \{X_2\}\}$.

7.2.3 Ajout d'une contrainte

L'ajout d'une contrainte est une opération triviale quand on ne considère pas le produit cartésien. Mais, dans le cas du produit, l'ajout d'une contrainte peut modifier la factorisation de manière arbitraire :

1. La nouvelle contrainte ne change pas la factorisation du polyèdre.

2. Le polyèdre était factorisé avant l'ajout mais ne l'est plus après. C'est le cas illustré par la figure 7.2 .
3. Le polyèdre n'était pas factorisé avant l'ajout mais le devient après. C'est le cas pour la figure 7.3 .

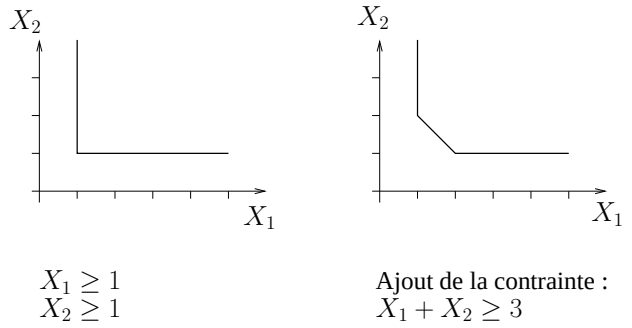


FIG. 7.2 – Perte de factorisation par l'ajout d'une contrainte

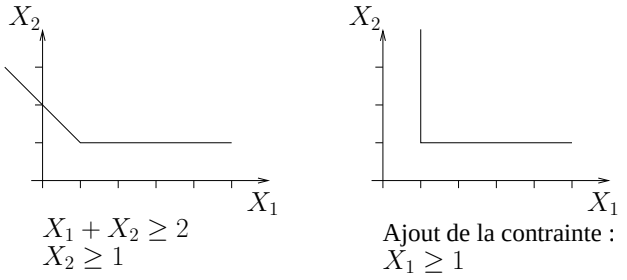


FIG. 7.3 – Factorisation par ajout d'une contrainte

Si la contrainte a tous ses coefficients non nuls dans la même classe, on ajoute la nouvelle contrainte à la classe concernée. Si la contrainte fait intervenir plusieurs classes, ces classes doivent être fusionnées afin de d'ajouter la contrainte Deux cas particuliers peuvent se produire :

- La contrainte est redondante et ne modifie pas le polyèdre. Il est donc inutile et coûteux de reconstruire le polyèdre. Par exemple, dans le cas de la figure 7.2, si au lieu de rajouter $X_1 + X_2 \geq 3$, on avait rajouté $X_1 + X_2 \geq 1$, le polyèdre n'aurait pas changé.
- Le deuxième cas intéressant est celui où la contrainte est insatisfaisable, dans ce cas le produit sera vide. Il est trivial de créer un polyèdre vide et ce quelle que soit sa taille. Sur l'exemple de la figure 7.2, cela peut s'illustrer en rajoutant une contrainte de la forme $X_1 + X_2 < 0$.

Il faut donc éviter de reconstruire le polyèdre pour s'apercevoir ensuite qu'il est factorisable, et pour cela il faut détecter que la contrainte est inutile sans reconstruire le polyèdre. La détection exacte de ce type de situation (satisfaction de la contrainte, ou de

la contrainte opposée, par tous les éléments générateurs du polyèdre) étant trop coûteuse, nous proposons une technique approchée : nous allons construire un hypercube contenant le polyèdre. à partir de ses générateurs. Nous calculons le minimum et le maximum de chaque variable comme suit :

- De l'ensemble des sommets on extrait un minimum et un maximum.
- S'il existe un rayon, avec un coefficient strictement positif (resp., strictement négatif) pour la variable, celle-ci n'a pas de maximum (resp. de minimum) fini ;
- Les variables dont les coefficients dans une droite sont non nuls, ne sont bornées ni supérieurement, ni inférieurement.

On remplace les variables de la contrainte par leur maximum ou leur minimum pour obtenir un encadrement entre m et M (éventuellement infini). On utilise cet encadrement pour faire la distinction entre 3 cas :

1. La contrainte est toujours vraie : elle est redondante et on peut la supprimer. C'est ce qui se passe pour une contrainte $\Sigma a_i X_i \geq b$ avec $m \geq b$. Sur l'exemple 7.2, on approxime le polyèdre avec pour X_1 et X_2 un minimum à 1 et pas de maximum. On peut en déduire que $X_1 + X_2 \geq 2$ donc la contrainte $X_1 + X_2 \geq 1$ est redondante.
2. La contrainte est toujours fausse : le polyèdre sera vide. C'est ce qui se passe pour une contrainte $\Sigma a_i X_i \leq b$ avec $b \leq m$. Sur l'exemple, ce serait le cas pour la contrainte $X_1 + X_2 \leq 1$.
3. Sinon il faut fusionner les sous-polyèdres.

7.2.4 Application affine

Considérons la transformation linéaire d'un polyèdre P, factorisé selon $I_1, \ldots, I_k$, par une matrice carrée F de dimension n et un vecteur G de dimension n. Pour utiliser la factorisation du polyèdre P, il faut que F vérifie :

$$\forall k_1, k_2 \in 1 \ldots k, (k_1 \neq k_2), \forall i \in I_{k_1}, \forall j \in I_{k_2},\ F_{ij} = F_{ji} = 0$$

Autrement dit, la matrice F est bloc-diagonale selon $I_1, \ldots, I_k$. Sous cette condition,

$$[F, G](P) = ([F, G](P{\downarrow}I_1 \uparrow)){\downarrow}I_1 \times \ldots \times ([F, G](P{\downarrow}I_k \uparrow)){\downarrow}I_k$$

Démonstration: (pour $k = 2$) Remarquons d'abord que le terme constant G n'intervient pas dans la factorisation. Soit $X \in P$, et soit $x_i = \sum_{j=1}^{n} F_{ij} * X_j$ la i-ième composante de FX. Alors, si $i \in I_1$, tous les facteurs F_{ij}, pour $j \in I_2$ sont nuls, par hypothèse sur F, et donc $x_i = \sum_{j \in I_1} F_{ij} * X_j$. x_i ne dépend donc que des composantes de X dont les indices sont dans I_1, ce qu'on peut écrire

$$(FX){\downarrow}I_1 = (F(X{\downarrow}I_1) \uparrow){\downarrow}I_1$$

et comme $FX = FX{\downarrow}I_1 \uparrow + FX{\downarrow}I_2 \uparrow$ le résultat suit.
□

Exemple : Soient

$$P = (1 \leq X_1 \leq 3) \wedge (1 \leq X_2 \leq 3) \text{ et } F = \begin{bmatrix} 2 & 0 \\ 0 & 1 \end{bmatrix} G = \begin{bmatrix} 0 \\ 1 \end{bmatrix}$$

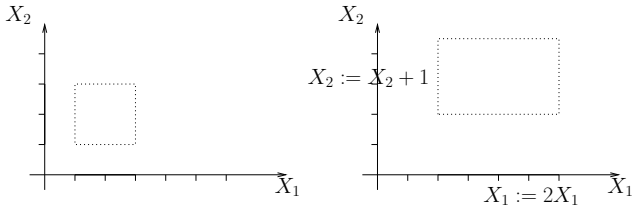


FIG. 7.4 – Transformation affine d'un produit

$$[F, G](P) = (2 \leq X_1 \leq 6) \wedge (2 \leq X_2 \leq 4)$$

7.2.5 Intersection

L'intersection de 2 polyèdres partitionnés de la même manière est facile, puisque

$$P \cap Q = (P_1 \cap Q_1) \times (P_2 \cap Q_2)$$

Démonstration: Soit $X \in P \cap Q$, alors $X = (x_1, x_2) \in (P_1 \times P_2) \cap (Q_1 \times Q_2)$, c'est-à-dire $x_1 \in P_1 \cap Q_1$ et $x_2 \in P_2 \cap Q_2$, donc $X \in (P_1 \cap Q_1) \times (P_2 \cap Q_2)$. Inversement, si $X = (x_1, x_2) \in (P_1 \cap Q_1) \times (P_2 \cap Q_2)$, alors $x_1 \in P_1 \cap Q_1$ et $x_2 \in P_2 \cap Q_2$, donc $X \in P_1$ et $X \in P_2$.
□

Exemple : Soient (cf. Fig. 7.5) $P = \{1 \leq X_1 \leq 5, 0 \leq X_2 \leq 3 = \{1 \leq X_1 \leq 5\} \times \{0 \leq X_2 \leq 3\}$, et $Q = \{2 \leq X_1 \leq 6\} \times \{0 \leq X_2 \leq 3\}$. On a $P \cap Q = (2 \leq X_1 \leq 5) \times (1 \leq X_2 \leq 3)$

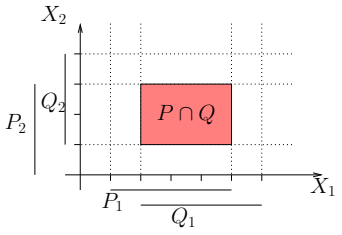


FIG. 7.5 – Intersection de 2 polyèdres factorisés

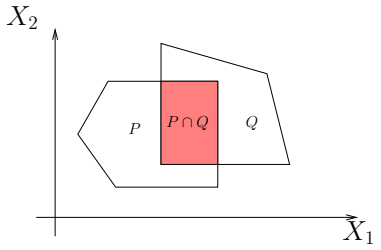


FIG. 7.6 – L'intersection peut être plus factorisée que ses opérandes

Si l'intersection se fait entre des polyèdres partitionnés en m on a :

$$P \cap Q = (P_1 \cap Q_1) \times \ldots \times (P_m \cap Q_m)$$

Il se peut que le résultat de l'intersection soit plus factorisé que ses opérandes (cf. Fig 7.6). Il faut donc faire une détection de factorisation sur le résultat.

7.2.6 L'élargissement

L'opérateur standard

Si, pour $P \bigtriangledown Q$, on prend l'opérateur standard qui consiste à éliminer de P les contraintes qui ne sont pas dans Q, on peut faire le calcul composante par composante. En effet, chaque inégalité de P est traitée indépendamment des autres inégalités. Les inégalités étant regroupées selon les variables, l'opérateur standard s'applique normalement. La factorisation du résultat peut être plus fine que celle des opérandes car on supprime des inégalités et donc on peut faire disparaître des liens entre variables.

L'opérateur réel

On a vu que, dans la pratique, l'opérateur d'élargissement est légèrement différent. Son calcul nécessite la détermination des contraintes mutuellement redondantes, ce qui nécessite de calculer les ensembles de générateurs saturant chaque contrainte. Pour que deux contraintes c de P_1 et c' de Q_1 soient mutuellement redondantes il faut qu'elles saturent les mêmes générateurs. Pour savoir si c sature un générateur il suffit de regarder la valeur des variables de I_1, les autres variables ont un coefficient nul dans c. Il suffit donc d'étudier la saturation sur les générateurs de P_1 et Q_1. On peut effectuer l'élargissement composante par composante.

Un autre avantage est la réduction du nombre de sommets à étudier. En effet si n_i est le nombre de sommets de P_i alors le nombre de sommets de P est $N = \Pi_i n_i$, il faut calculer la saturation sur ces N sommets. Par contre avec le produit le calcul se fait composante par composante, donc le calcul fait appel à $\sum_i n_i$ sommets. En général la somme est inférieure au produit.

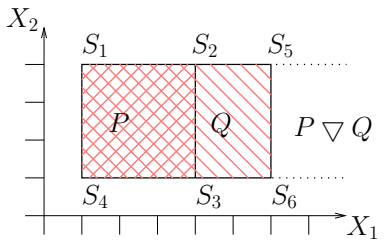


FIG. 7.7 – Elargissement d'un polyèdre factorisé

Exemple : Considérons les polyèdres P et Q de la Figure 7.7. Si on construit la matrice de saturation on obtient :

	Contraintes de P				Contraintes de Q			
	$\mathbf{1 \leq X_1}$	$X_1 \leq 4$	$\mathbf{1 \leq X_2}$	$\mathbf{X_2 \leq 4}$	$\mathbf{1 \leq X_1}$	$X_1 \leq 6$	$\mathbf{1 \leq X_2}$	$\mathbf{X_2 \leq 4}$
S_1=(1,4)	**0**	1	**1**	**0**	**0**	1	**1**	**0**
S_2=(4,4)	**1**	0	**1**	**0**	**1**	1	**1**	**0**
S_3=(4,1)	**1**	0	**0**	**1**	**1**	1	**0**	**1**
S_4=(1,1)	**0**	1	**0**	**1**	**0**	1	**0**	**1**
S_5=(6,4)	**1**	1	**1**	**0**	**1**	0	**1**	**0**
S_6=(6,1)	**1**	1	**0**	**1**	**1**	0	**0**	**1**

On garde les contraintes $1 \leq X_1$ et $1 \leq X_2 \leq 4$. On s'intéresse maintenant à la décomposition selon X_1, on a 3 sommets, on effectue le calcul de saturation sur ces sommets. On obtient les résultats suivants :

	Contraintes de P_1		Contraintes de Q_1	
	$\mathbf{1 \leq X_1}$	$X_1 \leq 4$	$\mathbf{1 \leq X_1}$	$X_1 \leq 6$
S'_1=(1)	**0**	1	**0**	1
S'_2=(4)	**1**	0	**1**	1
S'_3=(6)	**1**	0	**1**	1

On garde la même contrainte : $1 \leq X_1$

Comme pour l'opérateur standard, la suppression de certaines contraintes peut rendre la factorisation plus fine.

7.3 L'enveloppe convexe

Le calcul de l'enveloppe convexe est plus complexe car le résultat n'est pas toujours factorisé. L'enveloppe convexe est incluse dans le produit cartésien des enveloppes mais

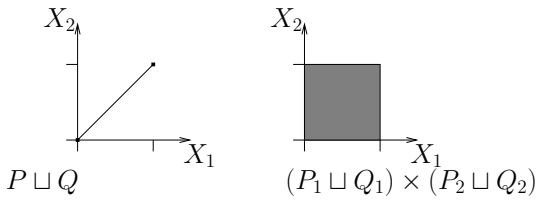


FIG. 7.8 – Contre-exemple pour l'enveloppe convexe

la réciproque n'est pas toujours vraie.

Démonstration: Soit $P = P_1 \times P_2$ et $Q = Q_1 \times Q_2$. Montrons que $P \sqcup Q \subseteq (P_1 \sqcup Q_1) \times (P_2 \sqcup Q_2)$: Soit $X \in P \sqcup Q, X = \lambda X_P + (1-\lambda)X_Q (\lambda \in [0,1])$. On a $X_P = X_P{\downarrow}I_1 \uparrow + X_P{\downarrow}I_2 \uparrow$ et $X_Q = X_Q{\downarrow}I_1 \uparrow + X_Q{\downarrow}I_2 \uparrow$. Donc $X = \lambda X_P{\downarrow}I_1 \uparrow + (1-\lambda)X_Q{\downarrow}I_1 \uparrow + \lambda X_P{\downarrow}I_2 \uparrow + (1-\lambda)X_Q{\downarrow}I_2 \uparrow$. Soit encore $X = X_1 \uparrow + X_2 \uparrow$, avec $X_1 \in X_P{\downarrow}I_1 \sqcup X_Q{\downarrow}I_1$, et $X_2 \in X_P{\downarrow}I_2 \sqcup X_Q{\downarrow}I_2$, et $X \in (P_1 \sqcup Q_1) \times (P_2 \sqcup Q_2)$.

La réciproque est fausse comme le montre le contre-exemple de la Figure 7.8. On a :

$$\begin{array}{ll} P = P_1 \times P_2 & \text{avec } P_1 = (X_1 = 0) \quad \text{et } P_2 = (X_2 = 0) \\ Q = Q_1 \times Q_2 & \text{avec } Q_1 = (X_1 = 1) \quad \text{et } Q_2 = (X_2 = 1) \end{array}$$

$P \sqcup Q = (0 \leq X_1 \leq 1 \ \wedge \ X_1 = X_2)$ est un segment alors que $(P_1 \sqcup Q_1) \times (P_2 \sqcup Q_2) = (0 \leq X_1 \leq 1) \times (0 \leq X_2 \leq 1)$ est un carré.
□

Il faut donc trouver un moyen pour effectuer le calcul sur le produit en sachant à l'avance si le résultat sera factorisé, et, s'il ne l'est pas, essayer cependant d'utiliser les calculs effectués.

Calcul de l'enveloppe convexe de polyèdres factorisés. Soient $P = P_1 \times P_2$ et $Q = Q_1 \times Q_2$ deux polyèdres factorisés selon la même partition (I_1, I_2). Soit $V' = V \cup \{\lambda\}$ l'ensemble des variables augmenté d'une variable auxiliaire (λ) et $V'_i = I_i \cup \{\lambda\}$ (pour $i = 1, 2$).Posons

$$\begin{array}{rl} & R = (P_1{\uparrow}V'_1 \cap \{\lambda = 0\}) \sqcup (Q_1{\uparrow}V'_1 \cap \{\lambda = 1\}) \\ \text{et} & S = (P_2{\uparrow}V'_2 \cap \{\lambda = 0\}) \sqcup (Q_2{\uparrow}V'_2 \cap \{\lambda = 1\}) \end{array}$$

Alors :
- si λ est borné par des expressions non constantes
  - supérieurement dans R
  - et inférieurement dans S

  ou inversement, alors l'enveloppe convexe $P \sqcup Q$ n'est pas factorisée selon (I_1, I_2), et $P \sqcup Q = (R{\uparrow}V' \cap S{\uparrow}V'){\downarrow}V$.
- sinon, $P \sqcup Q = R{\downarrow}I_1 \times S{\downarrow}I_2$.

Démonstration: Un point X de $P \sqcup Q$ est une combinaison convexe d'un point de P et d'un point de Q, soit

$$X \in \sqcup Q \Leftrightarrow \exists \lambda \in [0,1], \exists X_P \in P, \exists X_Q \in Q,\ X = \lambda X_P + (1-\lambda) X_Q$$

X est dans P pour $\lambda = 0$, dans Q pour $\lambda = 1$. Avec les mêmes notations, puisque P et Q sont factorisés selon (I_1, I_2), $X_P = (X_P{\downarrow}I_1){\uparrow}V + (X_P{\downarrow}I_2){\uparrow}V$ et $X_Q = (X_Q{\downarrow}I_1){\uparrow}V + (X_Q{\downarrow}I_2){\uparrow}V$. Soit encore,

$$\begin{aligned} X \in P \sqcup Q \Leftrightarrow\ & \exists \lambda \in [0,1], \exists X_P^{(I_1)} \in P_1, \exists X_P^{(I_2)} \in P_2 \exists X_Q^{(I_1)} \in Q_1, \exists X_Q^{(I_2)} \in Q_2, \\ & X = (\lambda X_P^{(I_1)} + (1-\lambda) X_Q^{(I_1)}){\uparrow}V + \lambda X_P^{(I_2)} + (1-\lambda) X_Q^{(I_2)}){\uparrow}V \end{aligned}$$

Ceci exprime presque que X est la somme d'un point de $P_1 \sqcup Q_1$ et d'un point de $P_2 \sqcup Q_2$, sauf que la quantification sur λ ne peut pas être distribuée sur les deux termes. Par contre, la caractérisation de X s'exprime aussi

$$X \in P \sqcup Q \Leftrightarrow \exists \lambda \in [0,1], \exists X' \in R, \exists X'' \in S, X = (X' + X''){\downarrow}V$$

L'élimination de λ (par Fourier-Motzkin) dans cette dernière expression ne peut entraîner des dépendances entre les variables de I_1 et I_2 que dans les cas suivants :

- *λ est borné supérieurement par une expression non constante, e_1, dans R, et inférieurement par une expression non constante, e_2, dans S (ce qui produira une contrainte $e_2 \leq e_1$ dans le résultat) ;*
- *λ est borné inférieurement par une expression non constante, e_1, dans R, et supérieurement par une expression non constante, e_2, dans S (ce qui produira une contrainte $e_1 \leq e_2$ dans le résultat).*

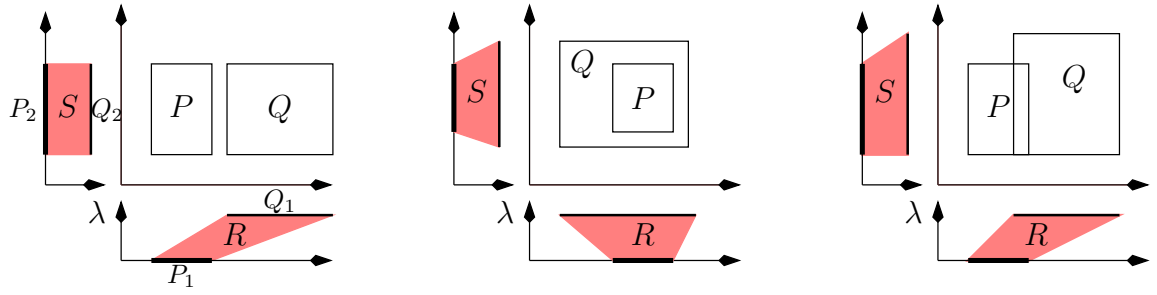
Si l'un de ces cas se présente, il faut éliminer conjointement λ dans les systèmes de R et de S, ce qui exprime que $P \sqcup Q = (R{\uparrow}V' \cap S{\uparrow}V'){\downarrow}V$. Sinon, l'élimination de λ peut être effectuée séparément, et on a $P \sqcup Q = R{\downarrow}I_1 \times S{\downarrow}I_2$.
□

La procédure d'enveloppe convexe factorisée consiste donc à calculer les deux enveloppes R et S (au prix d'une variable supplémentaire, λ, ajoutée à I_1 et I_2), puis à tester si le résultat sera partitionné ; dans l'affirmative, les projections $R{\downarrow}I_1$ et $S{\downarrow}I_2$ fournissent directement les facteurs du résultat. Sinon, il faut conjoindre les systèmes de contraintes de R et S avant d'éliminer λ.

La figure 7.9 illustre les principaux cas dans l'application de cette procédure.

Retour sur l'exemple :

On a : $P = (X_1 = 0) \times (X_2 = 0)$, $Q = (X_1 = 1) \times (X_2 = 1)$
Donc $R_1 = (0 \leq X_1 = v \leq 1)$ et $R_2 = (0 \leq X_2 = v \leq 1)$ R_1 et R_2 sont dépendants, $X_1 \leq v \leq X_2$ et on a :
$P \sqcup Q = (R_1 \cap R_2)\downarrow_v = (0 \leq X_1 = X_2 \leq 1)$



(a) λ borné seulement par des constantes dans S : résultat factorisé

(b) λ borné seulement inférieurement dans R et S : résultat factorisé

(c) λ borné supérieurement dans R et inférieurement dans S : résultat non factorisé

FIG. 7.9 – Enveloppe convexe factorisée

Par contre si on prend $Q = (X_1 = 1) \times (X_2 = 0)$ on obtient :
$R_1 = (0 \leq X_1 = v \leq 1)$ et $R_2 = (X_2 = 0 \wedge 0 \leq= v \leq 1)$
R_1 et R_2 sont indépendants et on a :
$P \sqcup Q = R_1 \downarrow_v \times R_2 \downarrow_v = (0 \leq X_1 \leq 1) \times (X_2 = 0)$

Dans le cas où la factorisation est en n sous polyèdres, c'est presque la même chose. La même variable v permet de savoir quels sont les sous-polyèdres qui seront liés. Pour chaque sous-polyèdre, on rajoute la dimension v avec $v = 0$ pour les P_i et $v = 1$ pour les Q_i. Les calculs se font sur le produit et, à la fin, on sépare les R_i en deux : groupes.

- $R_{i1}, \ldots, R_{ik}$: les sous-polyèdres où v n'introduit pas de dépendance (i.e. v n'est pas bornée par une expression non constante) ; on se contente d'éliminer v.
- $R_{j1}, \ldots, R_{jl}$: les sous-polyèdres où v introduit une dépendance. Il faut rassembler tous les sous-polyèdres avant d'éliminer la variable v.

L'enveloppe finale est alors :
$P \sqcup Q = R_{i1} \downarrow_v \times \ldots \times R_{ik} \downarrow_v \times (R_{j1} \cap \ldots \cap R_{jl}) \downarrow_v$

7.4 Conclusion

On dispose désormais de l'expression théorique de la plus part des opérations sur les polyèdres. Il est toujours possible d'utiliser la décomposition des opérandes pour simplifier les calculs même si dans certains cas, en particulier avec l'enveloppe convexe, le résultat est non décomposé.

On a également démontré que ces opérations donnent les mêmes résultats que celles

effectuées sur des polyèdres complets. Il est donc possible de faire les calculs sur des polyèdres plus petits sans perdre d'informations. Dans l'analyseur, on remplace l'utilisation des polyèdres par des produits cartésiens.

La décomposition, élément clé du gain, est dépendante de la base choisie pour les variables. Ainsi un hypercube parallèle aux axes sera décomposé en produit d'intervalle alors qu'un hypercube dont aucune face n'est parallèle aux axes ne sera pas décomposé. La chapitre suivant présente une solution pour que la décomposition soit indépendante de la base.

Chapitre 8

Factorisation par Changement de base

La décomposition en produit cartésien permet souvent de réduire significativement la dimension de l'espace dans lequel on travaille, et donc de traiter des cas de plus grande dimension. Elle est cependant évidemment très dépendante du choix des variables dans le programme initialisé : un simple changement de variable peut changer complètement la complexité de l'analyse. L'objectif de ce chapitre est de chercher à éviter ce problème, en s'autorisant des changements de variables adéquats avant d'effectuer certaines opérations sur les polyèdres.

8.1 Changement de base

8.1.1 Exemple

Nous expliquons d'abord, sur un petit exemple, l'objectif poursuivi et la technique de base pour y parvenir. Considérons le système de contraintes suivant, correspondant au polyèdre de la figure 8.1.(a) :

$$\begin{array}{rcl} 0 & \leq & 2X_1 - X_2 - 2 \\ 0 & \leq & -X_1 + 2X_2 - 2 \\ 0 & \leq & -2X_1 + X_2 + 8 \\ 0 & \leq & X_1 - 2X_2 + 8 \end{array}$$

Dans ce système, les deux variables sont liées par chaque contrainte, le polyèdre ne se factorise donc pas. Par contre, si on effectue le changement de variables

$$Y_1 = 2X_1 - X_2 \qquad Y_2 = -X_1 + 2X_2$$

on obtient le système (Fig.8.1.(b))

$$2 \leq Y_1 \leq 8 \quad , \quad 2 \leq Y_2 \leq 8$$

qui est factorisé.

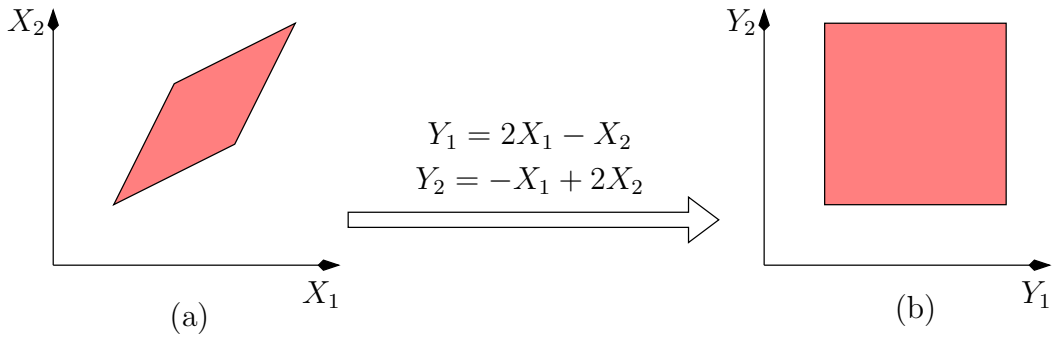


FIG. 8.1 – Factorisation par changement de base

8.1.2 Algorithme

L'exemple précédent suggère un algorithme très simple (analogue à la méthode du pivot de Gauss) pour opérer un changement de variables adéquat.

L'algorithme consiste à remplacer toutes les variables X_i par des variables Y_i obtenues par combinaison des contraintes. Il modifie progressivement le système $AX \leq B$, en introduisant de nouvelles variables Y (analogues aux "variables d'écart" de la programmation linéaire), et en construisant à mesure les équations E des variables X en fonction des Y.

```
pour i = 1 à n faire
  si il existe une variable x_j de coefficient non nul dans A_i
    Poser y_i = A_i X − B_i, en tirer l'équation de x_j, et l'ajouter à E
    Remplacer dans AX ≤ B et dans E toute occurrence de x_j par son expression
  fin si
fin pour
```

A la fin de cet algorithme, on obtient d'une part un système de contraintes $AY \leq B$, qui est "maximalement factorisable", et d'autre part un système d'équations E donnant le changement de variables effectué.

Exemple : considérons le système de contraintes suivant :

$$\begin{array}{rrrl} x_1 & +x_2 & +x_3 & \geq 0 \\ x_1 & & +x_3 & \geq 0 \\ 3x_1 & -x_2 & +x_3 & \geq 0 \\ -4x_1 & & -2x_3 & \geq -1 \end{array}$$

On introduit une première variable d'écart $y_1 = x_1 + x_2 + x_3$, équation d'où l'on tire x_1. Le système devient :

$$
\begin{array}{rrrl}
x_1 = y_1 - x_2 - x_3 & & & \\
\hline
y_1 & & & \geq 0 \\
y_1 & -x_2 & & \geq 0 \\
3y_1 & -4x_2 & -2x_3 \geq 0 & \\
-4y_1 & +4x_2 & +2x_3 & \geq -1
\end{array}
$$

La deuxième variable d'écart $y_2 = y_1 - x_2$ permet de tirer $x_2 = y_1 - y_2$, ce qui donne

$$
\begin{array}{rrrl}
x_1 = y_2 - x_3 & & & \\
x_2 = y_1 - y_2 & & & \\
\hline
y_1 & & & \geq 0 \\
 & y_2 & & \geq 0 \\
-y_1 & +4y_2 & -2x_3 & \geq 0 \\
 & -4y_2 & +2x_3 & \geq -1
\end{array}
$$

Enfin, on introduit la troisième variable d'écart $y_3 = -y_1 + 4y_2 - 2x_3$, et on obtient :

$$
\begin{array}{rrrl}
x_1 = y_1/2 - y_2 + y_3/2 & & & \\
x_2 = y_1 - y_2 & & & \\
x_3 = -y_1/2 + 2y_2 - y_3/2 & & & \\
\hline
y_1 & & & \geq 0 \\
 & y_2 & & \geq 0 \\
 & & y_3 & \geq 0 \\
-y_1 & & -y_3 & \geq -1
\end{array}
$$

Dans la nouvelle expression, la variable y_2 n'est pas liée aux 2 autres, le système est donc factorisable selon $I_1 = \{y_1, y_3\}$ et $I_2 = \{y_2\}$.

8.2 Stratégies de changement de base

On dispose désormais d'un moyen d'améliorer la décomposition d'un polyèdre. Il s'agit maintenant de savoir quand et comment l'appliquer.

8.2.1 Opérations sur un seul polyèdre

Le cas simple pour un changement de variables est celui où l'on s'intéresse à un seul polyèdre (recherche de générateurs, test du vide). Dans ce cas, on effectue d'abord une factorisation classique, celle-ci étant plus facile, puisqu'elle ne nécessite pas la réécriture des contraintes. On cherche ensuite à effectuer un changement de variables, par l'algorithme précédent, dans chaque sous-polyèdre ayant au moins 2 (ou k[1]) variables, avant

[1]Nous verrons au chapitre 9 que la factorisation n'est intéressante que pour des dimensions supérieures à un seuil, situé autour de 10. Mais ce seuil k peut-être laissé en paramètre.

de lui appliquer l'opération. Si cette opération est une recherche de système générateur, le résultat est obtenu par recomposition et changement de variables inverse.

Dans l'exemple du paragraphe précédent, on trouve tout de suite les systèmes générateurs des sous-polyèdres :

$$P^{(I_1)} = [\{(0,0),(0,1),(1,0)\},\emptyset,\emptyset] \quad , \quad P^{(I_2)} = [\{0\},\{1\},\emptyset]$$

La recomposition fournit le système générateur dans la base (y_1, y_2, y_3)

$$[\{(0,0,0),(0,0,1),(1,0,0)\},\{(0,1,0)\},\emptyset]$$

et en appliquant le changement de base inverse fourni par E, on obtient le système générateur dans la base (x_1, x_2, x_3) :

$$[\{(0,0,0),(1/2,0,-1/2),(1/2,1,-1/2)\},\{(-1,-1,2)\},\emptyset]$$

8.2.2 Opérations sur plusieurs polyèdres

Solution actuelle

En général, nous voulons trouver une factorisation commune à plusieurs polyèdres avant de leur appliquer une opération coûteuse (par exemple une enveloppe convexe). L'algorithme précédent ne peut pas être indépendamment appliqué à chaque argument, il n'y a aucune raison qu'il produise une factorisation commune.

La solution choisie consiste à appliquer l'algorithme à la *conjonction* des systèmes de contraintes (sans se préoccuper de la consistance de cette conjonction). Le résultat sera un changement de variables fournissant une factorisation commune maximale.

Optimisation possible

La solution précédente oblige à calculer un changement de variables avant chaque opération sur un gros système de contraintes. Une autre solution est de déterminer un changement de variables sur chaque opérande et de les utiliser pour en obtenir un commun. Un avantage de cette solution est de travailler sur des ensembles de contraintes plus petits. Le problème est alors d'obtenir un changement de variables commun aux deux polyèdres.

Un changement de base f_i (i=1,2) transforme la base initiale B en une base B_i. Il faut pouvoir exprimer ces deux bases dans une base commune B'. On procède en deux étapes. Dans un premier temps on exprime les B_i dans B (on utilise f_i^{-1}), on obtient un ensemble C de contraintes. Or on sait calculer une nouvelle base à partir d'un ensemble de contraintes, on utilise donc C pour obtenir B' et exprimer les P_i dans B'.

Cette solution offre plusieurs avantages, d'une part on travaille sur des ensembles de contraintes plus petits. En effets les B_i ne peuvent pas comporter plus de contraintes que les P_i, donc C comporte au plus autant de contraintes que P_1 et P_2. Un autre avantage est de pouvoir, à terme, mémoriser les changements de variables à l'intérieur de la structure de polyèdre, on pourrait ainsi manipuler des polyèdres dont la décomposition est toujours maximale.

Chapitre 9

Implémentation et Expérimentation

La base de notre expérimentation est l'analyseur décrit au chapitre 5, il nous a servi à tester les différentes solutions envisagées. Afin de tester l'efficacité du produit cartésien nous avons implémenté une bibliothèque spécifique que nous allons présenter en 9.1. Nous présenterons ensuite en 9.2 les différentes versions disponibles de l'analyseur. Puis en 9.3 nous présenterons et analyserons les résultats obtenus avec ces différentes versions.

9.1 Bibliothèque de polyèdres

Une contribution de cette thèse est la réalisation d'une bibliothèque de polyèdres capable de manipuler des produits cartésiens. Cette bibliothèque n'est en fait qu'une couche ajoutée à la "*Parma Polyhedra Library*" [PPL] (PPL).

Nous avons montré qu'il est possible d'utiliser les opérations de base sur les sous-polyèdres. Nous allons présenter au §9.1.1 comment détecter qu'un polyèdre se décompose en produit cartésien. Puis nous présenterons au §9.1.2 les champs spécifiques que doit posséder la structure de données pour pouvoir manipuler ces produits. Au §9.1.3 nous décrirons les actions spécifiques au produit, les opérations générales (intersection, enveloppe, ...) étant présentées au chapitre 7. Enfin au §9.1.4 nous indiquerons comment appeler notre bibliothèque.

9.1.1 Détection d'un produit cartésien

La décomposition d'un polyèdre en produit cartésien est détectée sur sa matrice de contraintes A. On définit la relation de dépendance $\leftrightarrow_A$ entre deux indices i et j de la manière suivante :

$i \leftrightarrow_A j$ si il existe une ligne l telle que $A_{li} \neq 0$ et $A_{lj} \neq 0$.

Cette relation est trivialement réflexive et symétrique, mais elle n'est pas transitive. On considère alors $\leftrightarrow_A^*$ sa fermeture transitive pour obtenir une relation d'équivalence. On dira que deux indices i et j sont dans la même classe si ils sont tels que $i \leftrightarrow_A^* j$.

Un polyèdre est décomposé en produit cartésien selon $\leftrightarrow_A^*$, on crée une composante par classe d'équivalence.

Algorithme de détection

On détecte les classes d'équivalence sur la matrice A de taille NbColonnes $\times$ NbLignes. On renvoie une partition qui contient les classes d'équivalences sous la forme des ensembles d'indices qui constituent chaque classe. Le calcul de la partition se fait de la manière suivante :

```
décomposition(A) retourne partition
partition = init_decomposition(NbColonnes)
pour iterL de 1 à NbLignes
    pos=1
    tant que A[iterL][pos] = 0
       pos = pos +1
    pour pos2=pos+1 à NbColonnes
       si A[iterL][pos2] ≠ 0
          a = find(pos, partition)
          b = find(pos2, partition)
          union(a,b, partition)
Retourner partition
```

La structure de données représentant les classes et les opérations **union** and **find** sont classiques [AHU74]. Les fonctions internes sont :

- **init_decomposition**(nbVar) : créer une décomposition où toutes les variables sont indépendantes.
- **find**(pos,décomposition) : trouver l'indice de la classe pos dans décomposition.
- **union**(ind1, ind2, partition) : faire l'union des classes d'indices ind1 et ind2 dans la partition courante. En sortie une des deux classes a été supprimée et le nombre de classes dans de la partition a été mis à jour.

Remarque. La détection se faisant sur la matrice de contraintes, si celle-ci n'est pas minimale on risque de considérer des variables liées alors qu'elles ne le sont pas. Par exemple, dans le polyèdre représenté Fig. 9.1, les variables X_1 et X_2 sont indépendantes mais la décomposition ne peut être détectée que si l'on supprime la contrainte redondante $X_1 + X_2 \geq 2$.

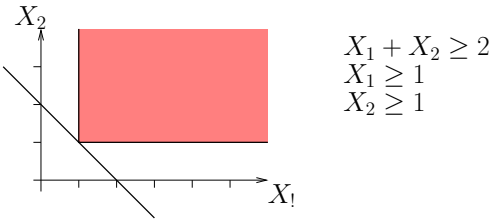


FIG. 9.1 – Système de contraintes sous forme non minimale

9.1.2 Spécificités de la structure

Une fois la décomposition détectée il faut pouvoir la conserver et pour cela :
- Savoir si le polyèdre se décompose et si oui en combien de sous-classes.
- Faire le lien entre une classe et les variables qui la composent.
- Avoir une représentation accessible de chaque élément du produit.

Comme il a été dit plus haut, notre implémentation consiste en une couche au dessus de la PPL [PPL] ; nous nous sommes basés sur la version 0.3. Nous explicitons ici les changements dans la structure de données représentant les polyèdres.

Le premier champ à ajouter est un champ qui indique si le polyèdre est décomposé ou non. Un moyen simple est de rajouter le nombre de classes.

Pour ce qui est du lien entre une classe et ses variables, on utilise des vecteurs d'entiers. Un premier vecteur contient pour chaque variable l'indice de la sous-classe la contenant. De plus pour éviter de systématiquement parcourir ce vecteur pour connaître tous les éléments d'une classe on rajoute un vecteur par classe qui contient les indices des variables contenu dans la classe.

Pour représenter les sous-polyèdres on utilise un tableau de pointeurs vers les sous-polyèdres.

La structure ressemble à :

Nom	Fonction
Dim	la dimension du polyèdre
NbPoly	le nombre de sous-polyèdres
MatC	la matrice de contraintes
MatG	la matrice de générateurs
Classes	les sous-classes de variables
Product	un tableau de sous-polyèdres

Si il y a plusieurs classes, les matrices de contraintes et de générateurs n'ont pas de sens. Si il y a une seule classe les sous-polyèdres n'ont pas de sens.

9.1.3 Opérations spécifiques au produit

Dans cette partie nous présentons en pseudo-code des opérations spécifiques au produit cartésien. Les champs d'un polyèdre sont ceux définis précédemment.

- *Classes* : ce sont les ($NbPoly$ +1) vecteurs d'entiers qui représentent la partition du polyèdre. Le premier vecteur (*Classes*[1]) indique la classe de chaque variable, les $NbPoly$ suivants donnent pour chaque classe les indices des variables de la classe. *Classes*$[i][j]$ pour $i > 1$ indique l'indice de la j^{eme} variable de la classe $i - 1$.
- *Product* : un tableau de $NbPoly$ pointeurs vers les différents polyèdres du produit. *Product*$[i]$ désigne le i^{eme} sous polyèdre, c'est un polyèdre classique.

Décomposition commune des polyèdres

On note :

- *vecteurE* : le type vecteur d'entiers.
- *partition* : les ($NbPoly$ +1) vecteurs d'une partition.

Le but est de partir de deux polyèdres ayant des décompositions quelconques pour finir avec une décomposition commune aux deux polyèdres. On procède en obtenant les mêmes classes dans les deux décompositions. Pour cela on cherche si une classe c de la première décomposition est égale à une de la seconde, donc si les vecteurs d'indices sont identiques. Pour cela on utilise la fonction **is_class_in**.

```
booléen is_class_in(vecteurE c, partition classes)
entier indice
indice = classes[c[1]]
si égale(c, classes[indice+1])
    retourner(vrai)
sinon
    retourner(faux)
```

La fonction égale renvoie vrai si le premier vecteur est égal au deuxième.

Si la classe c_1 d'indice i dans la partition $Classes_1$ est égale à une classe c_2 d'indice j dans la partition $Classes_2$ il faut juste s'assurer qu'elles ont le même indice, au besoin on permute. Si il n'existe pas une telle classe c_2 il faut fusionner les classes de $Classes_2$ ayant une intersection non vide avec c_1 puis chercher cette classe dans $Classes_1$.

La fusion de deux sous-composantes d'un polyèdre se fait en fusionnant les classes et les matrices de contraintes. La fusion des classes est faite par **fusion_classes** et la fusion des matrices par **fusion_matrices**. La fonction *taille* d'un vecteur renvoi sa taille, la fonction $ajouter$ lui ajoute (concaténation triée) les éléments d'un autre vecteur.

partition **fusion_classes**(*partition* classes, int pos1, int pos2, int *NbPoly*)
(*on suppose pos1<pos2 et NbPoly > 1*)
Pour col de 1 à classes[pos2+1].taille()
    classes[1][classes[pos2+1][col]] = pos1
classes[pos1+1].ajouter(classes[pos2+1])
pour ligne de pos2 à NbPoly -1
        classes[ligne] = classes[ligne+1]
Retourner classes

matrice **fusion_matrices**(*matriceE* m1, *matriceE* m2, *matriceB* classes, int pos1 , int pos2, int *Dim*)
matriceE nmat
copie1 = **matrices_augmente**(m1, classes, pos1, pos2, *Dim*)
copie2 = **matrices_augmente**(m2, classes, pos2, pos1, *Dim*)
nmat = **matrices_merge**(copie1,copie2)
Retourner(nmat) ;

La fonction **matrices_augmente**(m,classes,p1,p2,dim) agrandit la matrice en rajoutant des colonnes de zéro pour les indices de la classe p2. La fonction **matrices_merge**(m1,m2) fusionne 2 matrices de même taille en ajoutant les lignes de m2 à m1.

Avant toute opération sur des polyèdres décomposés on doit déterminer une partition commune, c'est le rôle de la fonction **même_décomposition**. Une opération, autre que l'enveloppe convexe, sur deux opérandes *op*1 et *op*2 se déroule de la manière suivante. On appelle **même_décomposition** pour *op*1 et *op*2, puis on effectue l'opération sur chaque sous-polyèdre, enfin on utilise le résultat de chaque sous opération pour obtenir le résultat final. Par exemple sur l'intersection on a :

intersection (*op*1, *op*2 polyèdre)
même_décomposition(*op*1, *op*2)
Résultat : polyèdre de taille *Dim*
Pour iter = 1 à *NbPoly* faire
    Résultat.Product[iter] = *intersection* (op1.Product[iter], op2.Product[iter])

Dans le cas de l'enveloppe il y a en plus l'ajout d'une dimension v pour détecter la décomposition du résultat.

Redécoupage

Après une opération pouvant augmenter la décomposition, comme par exemple dans le cas de l'ajout d'une contrainte, il peut être intéressant de rechercher une nouvelle décomposition plus fine.

Pour chaque sous-composante P_i du polyèdre initial on applique la fonction **polyOfConstraints**. Cette fonction prend une matrices de contraintes et crée un polyèdre, éventuellement décomposé, qui lui correspond. C'est la seule fonction qui détecte une décomposition. Si le résultat a plusieurs sous-polyèdres on remplace P_i par ses sous-polyèdres.

9.1.4 Interfaces de la bibliothèque

Interface en C

Dans un premier temps nous avons essayé d'utiliser les produits cartésiens dans Nbac(qui utilise une bibliothèque en C). Pour y parvenir, nous avons réalisé une interface en C qui permet d'appeler depuis Nbac notre bibliothèque. Mais, comme il a été dit plus haut, il est rapidement apparu que les traitements complexes effectués par Nbac sur la structure de contrôle ont tendance à masquer les variations de performances des opérations sur les polyèdres. Nbac ne permettant pas une expérimentation fine de nos opérations, nous avons développé notre propre analyseur (celui du chapitre 5), en C++ comme la PPL.

Interface en C++

Nous avons réalisé dans l'analyseur un module spécifique permettant de manipuler des ensembles numériques, que ce soient des polyèdres simples, des produits cartésiens ou des égalités. L'interface de notre bibliothèque a été conçue de manière à pouvoir se substituer à la PPL de la façon la plus transparente possible. On a donc repris l'interface de la bibliothèque initiale en conservant le noms et les paramètres des fonctions. La seule différence est le type des objets manipulés, des "*Polyhedron*" pour la version standard et des "*PolyhedronProduct*" pour manipuler des produits cartésiens. Il est donc possible de passer d'une version de la bibliothèque à l'autre simplement en remplaçant un type par l'autre.

9.2 Versions de l'anayseur

L'expérimentation des différents moyens envisagés pour simplifier les calculs se fait au moyen de l'analyseur décrit au chapitre 5. Pour cela nous en avons réalisé différentes versions, que nous présentons maintenant.

9.2.1 Version de base

La version simple correspond à un analyseur classique, sans aucune des optimisations que l'on cherche à tester. L'algorithme principal est le suivant :

Transformer l'automate Oc en automate au format interne.
Détecter les sous-composantes fortement connexes et en déduire un ordre de parcours.
Parcourir l'ensemble des états pour déterminer les contraintes du up to.
Créer un polyèdre par état de l'automate.
Tant que l'ensemble des polyèdres n'est pas stable faire :
    Parcourir tous les états et mettre les polyèdres à jour.

L'analyseur est structuré de la manière suivante :

- Un module **test** : qui gère la création et le parcours de l'automate, il correspond à l'algorithme principal.
- Un module **ocstruct** : qui gère l'interface entre l'automate en Oc et le code de l'analyseur.
- Un module **auto** : qui gère la création et l'accès à l'automate interne.
- Un module **state** : qui code un état de l'analyseur comme défini dans le paragraphe 5.3.1.
- Un module **action** : qui est chargé de mémoriser les actions qui seront associées à un état.
- Un module **numSet** : qui gère tout l'aspect lié à la manipulation des ensembles numériques.

Le traitement d'un état de l'automate est fait de la manière suivante :

Si P_{in} est vide fin du traitement.
Parcourir la liste des actions associées à l'état.
Mettre à jour les P_{succ_i}.
Transmettre les P_{succ_i} et mettre à jour
- P_{in} dans les états successeurs non visités
- P_{widen} dans les successeurs déjà visités.

Dans la version de base, **numSet** manipule des polyèdres et **test** lance une seule analyse. C'est cette version qui nous servira de base pour les comparaisons. Les versions suivantes reprennent cette structure, on ne donne que les différences par rapport à cette version de base.

9.2.2 Version avec détection des égalités

Il existe deux versions utilisant des égalités, celle qui effectue un pré-traitement (application de la méthode de Karr) et celle qui calcule les égalités pas à pas.

Version avec pré traitement

Cette version implique plusieurs modifications. La première consiste à modifier **numSet** pour rajouter un deuxième type d'ensembles numériques qui correspond aux égalités représentées et manipulées de la façon décrite dans le chapitre 6.

Il faut également modifier **test** pour ajouter, après la détection des contraintes utilisées dans le up to, une analyse du programme avec des égalités. Cette analyse implique de rajouter la possibilité de mémoriser des égalités dans un état, on modifie le module **state** pour mettre à jour la structure d'un état.

Enfin il faut modifier **numSet** afin d'utiliser les égalités trouvées lors de la première passe. Si avant une opération on connaît l'état qui va contenir le résultat, c'est le cas lorsque l'on transmet P_{succ_i} on dispose des égalités qui seront vérifiées par le résultat. On utilise ces égalités afin de simplifier les opérandes en éliminant une variable par égalité.

Version pas à pas

Cette version est très proche de la version de base, seul le module **numSet** à besoin d'être modifié : cette version pourrait être juste une couche ajoutée à la bibliothèque de polyèdres. Deux modifications sont nécessaires : la première est l'ajout d'un type servant à manipuler les égalités ; l'autre modification se situe au niveau des opérations : avant une opération d'intersection ou d'enveloppe on va chercher à simplifier les opérandes en détectant des égalités. Dans la version de base la fonction de **numSet** sert principalement à appeler la fonction adéquate de la bibliothèque de polyèdres. Dans cette version on a en plus le traitement des égalités. Nous illustrons le principe sur le cas de l'enveloppe :

Trouver les égalités de l'opérande 1 et les mémorisées dans Eq_1
Trouver les égalités de l'opérande 2 et les mémorisées dans Eq_2
Calculer Eq l'union de Eq_1 et Eq_2
Utiliser Eq pour simplifier les opérandes
Faire le calcul de l'enveloppe
Rajouter les égalités de Eq

9.2.3 Version avec produits cartésiens

Les 3 versions précédentes ont été étendues pour manipuler des polyèdres comme des produits cartésiens. Ainsi, par exemple, la version “base” étendue avec des produits

cartésiens sera appelée “base-produit”.

Pour passer d'une version simple à une version produit il suffit de changer le type des objets manipulés. On passe donc d'une version à l'autre en modifiant le module **numSet**.

9.2.4 Version avec changement de base

Pour cette dernière version on applique l'idée d'un changement de base décrite dans le chapitre 8. Ce changement de base servant à améliorer la décomposition en produit cartésien il n'existe pas de version manipulant de simples polyèdres.

Chaque version étendue avec les produits cartésiens est étendue à son tour avec le changement de base : la version “produit” ainsi étendue s'appelle “chgt-base”.

On part de la version simple avec produit cartésien et on rajoute avant l'opération d'enveloppe le changement de base. Cette modification s'effectue dans le module **numSet** où on remplace l'appel à la fonction d'enveloppe convexe par l'appel à une fonction d'enveloppe avec changement de base. Cette fonction a été rajouté directement dans la bibliothèque de polyèdres.

9.3 Expérimentations

9.3.1 Les exemples

Pour évaluer l'influence des techniques présentées dans cette thèse, il nous fallait expérimenter nos outils sur des exemples présentant un grand nombre de variables. De tels exemples sont rares dans la littérature, justement parce qu'ils sont difficiles à traiter ! Il est intéressant aussi de traiter des exemples génériques, sur lesquels on peut faire croître le nombre des variables, et étudier l'évolution des performances des différentes techniques.

Nous avons mené deux types d'expérimentation :

- Des expérimentations au niveau des opérations sur les polyèdres : il s'agit d'effectuer les opérations sur des opérandes variés.
- Des expérimentations de notre analyseur, appliqué à différents programmes.

Exemples d'application d'opérations

Pour examiner le comportement des différentes versions de la bibliothèque, nous avons appliqué les opérations les plus coûteuses (conversion d'une représentation dans l'autre, enveloppe convexe), d'une part sur certaines familles de polyèdres, et d'autre part sur un échantillon fourni par des partenaires extérieurs.

Les familles de polyèdres considérées sont soit des cas particulièrement favorables pour certaines techniques, soit au contraire, des cas défavorables :

- Deux familles de points $Z_n = \{x_i = 0, i = 1..n\}$, $U_n = \{x_i = 1, i = 1..n\}$, en dimension n, supposées particulièrement favorables à l'utilisation des équations, puisque Z_n et U_n satisfont n équations, et que $Z_n \sqcup U_n$ en satisfait encore $n - 1$.

- On définit la famille des hypercubes $H_n(a,b) = \{a \leq x_i \leq b, i = 1..n\}$, qui présente, bien sûr, une factorisation maximale. On s'intéressera en particulier à l'enveloppe convexe $H_n(0,2) \sqcup H_n(0,3)$, dont le résultat est encore factorisé, et à l'enveloppe $H_n(0,2) \sqcup H_n(1,3)$, dont le résultat ne l'est plus.
- On définit de même la famille $K_n(a,b) = \{\bigwedge_{j=1}^{n}(a \leq \Sigma_{i=1}^{j} x_i \leq b)\}$, qui sont des polyèdres transformables en hypercubes factorisés par changement de base. On s'intéressera aux enveloppes $K_n(0,1) \sqcup K_n(0,2)$ et $K_n(0,1) \sqcup K_n(1,2)$, la première ayant un résultat factorisé contrairement à la seconde.
- Enfin, la famille des simplexes $S_n(a,b) = \{x_i \geq a, i = 1..n, \sum_i x_i \leq b\}$ sera utilisée comme cas défavorable, puisque ces polyèdres ne se factorisent pas.

Par ailleurs, nous traiterons un gros "benchmark" d'opérations sur les polyèdres [IN04], collectées au cours d'expérimentations avec l'analyseur PIPS [IJT91]. Ces exemples ont la particularité de comporter souvent un très grand nombre de variables, mais d'être aussi très "creux", au sens où les variables sont peu liées entre elles. Cette caractéristique est due au fait que PIPS procède par analyse relationnelle ascendante (i.e., par extraction et composition des relations calculées par des fragments de programme de plus en plus gros).

Exemples de programmes analysés

La plupart des exemples de programmes analysés précédemment, en particulier avec NBAC, ne présentaient pas un nombre de variables suffisant pour être intéressants pour notre expérimentation. Par ailleurs, notre analyseur étant basé sur OC, pour être analysables, les exemples doivent comporter une structure de contrôle raisonnable (la compilation de programmes LUSTRE ou ESTEREL en OC peut produire des automates énormes, en nombre d'états de contrôle).

Une première approche pour obtenir des exemples satisfaisants a été de combiner entre eux des exemples individuellement trop simples. Evidemment, cette approche n'est pas complètement honnête, puisque dans les programmes obtenus, les variables des sous-exemples sont indépendantes entre elles, et donc ces programmes sont généralement favorables à la factorisation.

Une deuxième source d'exemples a été la thèse de Jérôme Leroux [Ler03], qui donne un ensemble de programmes, principalement des protocoles de communication, qu'il a analysé avec sa méthode. Le code de ces exemple peut être trouvé sur la page web de l'outil *FAST* développé au Laboratoire de Spécification et Ve"rification (voir [lsv]).

Enfin, nous traiterons deux exemples plus réalistes, et qu'il a été possible de complexifier selon nos besoins : le contrôleur de vitesse de métro, déjà utilisé dans [HPR97], et un dispositif d'acquisition de données gyroscopiques tolérant aux fautes, extrait d'un système avionique réel.

Le contrôleur de vitesse de métro. Un dispositif embarqué de contrôle de vitesse détecte d'une part des plots disposés le long de la voie, et d'autre part des signaux "seconde" émis par radio par une horloge centrale (commune à toutes les rames. Soient

nbp et nbs, respectivement, le nombre de plots et le nombre de secondes reçus. La vitesse idéale d'une rame est d'un plot à la seconde. Le contrôleur est en charge d'assurer cet objectif, mais pour éviter les secousses, une hystérésis est appliquée :

- Lorsque $\mathsf{nbp} \geq \mathsf{nbs} + 10$ la rame est en avance. Le contrôleur commande alors le freinage jusqu'à ce que $\mathsf{nbp} \leq \mathsf{nbs}$. Un freinage continu entraîne l'arrêt du train sur un parcours de 9 plots.
- Lorsque $\mathsf{nbp} \leq \mathsf{nbs} - 10$ la rame est en retard. Le contrôleur signale ce retard à l'horloge centrale, jusqu'à ce que $\mathsf{nbp} \geq \mathsf{nbs}$. Lorsqu'un train signale un retard, l'horloge centrale n'émet plus de signaux "seconde" (le temps logique s'arrête !)

L'analyse de ce système peut concerner plusieurs propriétés, et en particulier l'absence de collision. Il est possible d'augmenter le nombre de rames, mais au delà de 3, la structure de contrôle explose.

Le dispositif d'acquisition de données gyroscopiques. Le gyroscope est une étude de cas provenant de l'Agence Indienne pour de Développement de l'Avionique (ADA).

La position dans l'espace de l'avion est décrite selon trois axes qui correspondent au roulis, au tangage et au lacet. Le gyroscope possède un nombre nb de capteurs par axe et se sert des différentes valeurs obtenues afin de renvoyer une unique valeur par axe. L'aspect de tolérance aux fautes vient du fait qu'un capteur peut tomber en panne et dans ce cas le système doit détecter la faute, ignorer la valeur erronée et malgré tout renvoyer une valeur correcte.

Pour les tests nous n'utilisons pas la version complète du gyroscope car trop complexe. Pour être plus précis elle explose lors de la transformation d'un programme Lustre vers un automate Oc. Dans le programme que nous utilisons il y a au plus un capteur défaillant, dans la réalité plusieurs capteurs peuvent être défaillant en même temps mais cela provoque une explosion d'états lors du passage en Oc.

Nous transformons le programme de plusieurs façons. On peut soit changer le nombre de capteurs utilisés sur chaque axe, soit considérer qu'un capteur est défaillant ou défaillant selon un axe. Nous considérerons les programmes suivants :

- 3/4/1 : le système traite les 3 axes, comporte 4 capteurs par axe, et le comportement défaillant d'un capteur est le même sur les 3 axes.
- 3/4/2 : le système traite les 3 axes, comporte 4 capteurs par axe, le comportement défaillant d'un capteur varie selon l'axe considéré (un axe distingué des 2 autres).
- 3/5/1, 3/6/1, 3/7/1 : cinq, six, sept : le système traite les 3 axes, comporte 5, 6 ou 7 capteurs par axe, un capteur est soit correct sur les 3 axes soit défaillant.

L'écart entre la vraie valeur et celle renvoyée par un capteur est en général une variable interne. Lors du passage en Oc cette variable est remplacée par la formule utilisée. Si on force la compilation à garder ces variables, par exemple en les déclarant comme sorties, on peut augmenter le nombre de variables numériques manipulées. C'est ce qui se passe dans les deux exemples suivants :

- 2/4/2bis : le système traite 2 axes, comporte 4 capteurs par axe, et comportement correct/défaillant des capteurs dépend de l'axe ; les écarts sont déclarés comme

sorties.
- 3/4/1bis : c'est le programme 3/4/1 dans lequel les écarts sont déclarés comme sorties.

Enfin si on accepte d'avoir plus d'un capteur défaillant on obtient un autre ensemble de versions. Nous considérerons le programme 1/4/ ?, où le système traite 1 axe avec 4 capteurs, et éventuellement plusieurs capteurs défaillants.

9.3.2 Expérimentations avec les équations linéaires

Au chapitre 6, nous avons présenté deux techniques d'utilisation des équations linéaires :

La première consiste à calculer d'abord les équations invariantes, par la méthode de Mike Karr, puis à utiliser ces équations pour éliminer des variables. Nous avons déjà indiqué que cette approche ne donne pas d'amélioration mesurable, au contraire. La première passe ne détecte en général que très peu d'équations globalement invariantes, et son coût surpasse presque toujours les économies qu'elle permet lors de la deuxième analyse.

La deuxième technique (§6.4) consiste à utiliser les équations temporairement satisfaites, en utilisant, lors de chaque opération coûteuse, les équations satisfaites par les opérandes, pour précalculer les équations satisfaites par le résultat.

La table 9.1 donne les temps d'exécution (en secondes) de la version de base, de la version avec calcul préalable des équations, et de la version avec utilisation des équations temporaires.

Cette table montre que les deux techniques basées sur les équations donnent des résultats généralement mauvais. Pour analyser plus finement le comportement de la technique "pas-à-pas", nous l'avons appliquée au calcul de l'enveloppe convexe dans une famille de cas particulièrement favorables : le calcul de $Z_n \sqcup U_n$. Les résultats sont donnés par la table ci-contre.

dim	base	temp_eq
10	0.01	0.01
20	0.02	0.01
30	0.03	0.02
40	0.05	0.03
50	0.07	0.05

Ainsi, même dans un cas aussi favorable, l'exploitation des équations ne donne que des résultats très décevants. L'explication tient au fait que, dans les bonnes implémentations de l'algorithme de Chernikova, les équations sont plus ou moins implicitement exploitées. L'utilisation des équations affines conduit donc à une impasse. Mais même les échecs méritent d'être rapportés, ne fût-ce que pour éviter à d'autres d'explorer cette impasse [1].

[1] "L'homme sage apprend de ses erreurs, l'homme plus sage apprend des erreurs des autres." (Confucius)

	dim	base	pre_eq	temp_eq
voiture	3	0.02	0.03	0.03
centralserv	13	1.44	1.58	2.17
firefly	4	0.68	0.87	1.56
lamport	11	1.63	1.77	3.05
lifo	7	0.57	0.67	1.06
rtp	9	0.54	0.61	0.79
swimingpool	9	0.30	0.33	0.59
ttp2	9	0.80	0.91	1.34
barber	8	1.82	1.57	2.43
consistency	12	3.97	5.86	5.88
consprodjava	18	3.55	4.07	6.27
readwrite	13	4.82	5.25	7.72
csm	13	11.33	12.16	15.61
fms	22	21.54	23.99	41.18
kanban	16	11.11	12.85	14.10
multipoll	17	16.14	18.59	22.69
peterson	14	69.28	69.61	57.77
barber efm	14	24.38	22.95	27.89
barber moesi	12	10.20	12.45	14.22
barber synapse	11	6.69	6.50	10.24
firefly moesi	8	11.42	12.84	11.55
barber mesi moesi	16	184.93	140.80	399.42
barber mesi synapse	15	57.20	122.81	82.67

TAB. 9.1 – Utilisation des équations sur les exemples simples

9.3.3 Expérimentations avec les produits cartésiens et le changement de base

Nous expérimentons d'abord nos techniques sur diverses familles d'opérations. Pour chacune d'entre elles, nous donnerons la dimension, le temps d'exécution avec la version de base, la version avec produit cartésien, et la version avec changement de base. La table 9.2 concerne :

(a) le calcul du système générateur des hypercubes $H_n(0,1)$, cas le plus favorable pour la factorisation. L'expérience confirme la réduction logarithmique du temps de calcul, et montre aussi que la tentative de changement de base, évidemment inutile dans ce cas, n'introduit pas de surcoût perceptible.

(b) le calcul des enveloppes convexes $H_n(0,1) \sqcup H_n(0,3)$, cas très favorables encore, puisque les opérandes et le résultat sont factorisés.

(c) le calcul des enveloppes convexes $H_n(0,1) \sqcup H_n(1,3)$, cas moins favorable, le résultat n'étant pas factorisé du tout. De façon inattendue, on constate cependant que le calcul factorisé est quand même bien plus efficace.

(d) le calcul des enveloppes convexes $K_n(0,1) \sqcup K_n(0,2)$. Les opérandes sont factorisables en hypercubes par changement de base, et le résultat est alors encore factorisé.

(e) le calcul des enveloppes convexes $K_n(0,1) \sqcup K_n(1,2)$, cas analogue mais où le résultat n'est pas factorisé du tout.

(f) le calcul des enveloppes convexes $S_n(0,1) \sqcup S_n(1,2)$, cas très défavorable aux deux méthodes, puisqu'aucune factorisation n'est possible : on ne constate cependant aucun surcoût mesurable.

Pour tester les performances des opérations sur des exemples moins ad-hoc, nous avons calculé toutes les enveloppes convexes de la base d'opérations venant de PIPS. La table 9.3 donne, pour chaque dizaine de dimensions, le nombre d'enveloppes effectuées, et le rapport "temps de la version de base" / "temps de la version avec factorisation".

Considérons maintenant les exemples d'analyse de programmes : la table 9.4 donne les résultats de l'analyse des exemples simples déjà considérés. Sans surprise, cette table montre que les techniques de factorisation ne sont intéressantes qu'à partir d'une certaine dimension, mais qu'elles peuvent alors entraîner des réductions du temps de calcul tout-à-fait significatives. Dans le cas contraire, le surcoût qu'elles provoquent est rarement important (à quelques notables exceptions près : "consistency", "consprodjava").

Pour l'exemple du contrôleur de vitesse de métro, nous n'avons pas pu mener nos expérimentations au delà de 4 trains, en raison de l'explosion de la taille de l'automate de contrôle. Les résultats sont donnés dans la table 9.5. C'est un exemple qui montre bien l'intérêt du changement de base.

Enfin, nous donnons dans la table 9.6 les performances de l'analyse du système d'acquisition de données gyroscopique, en faisant varier le nombre d'axes (ce qui n'a évidemment pas de sens physique !), le nombre de capteurs, et en considérant les différentes version décrites précédemment.

Dimension	10	11	12	13	14	15	16	17
base	0.18	0.66	4.09	24.21	112.79	477.71	>600	>600
prod	0.06	0.14	0.26	0.56	1.27	2.51	5.55	11.33
chgt-base	0.07	0.15	0.30	0.55	1.24	2.61	5.55	11.33

(a) Systèmes générateurs des hypercubes $H_n(0,1)$

Dimension	12	13	14	15	50
base	18.99	102.94	516.54	>600	>600
prod	0.01	0.02	0.02	0.02	0.03
chgt-base	0.01	0.02	0.02	0.02	0.03

(b) Enveloppes convexes $H_n(0,1) \sqcup H_n(0,3)$, résultat factorisé

Dimension	8	9	10	11	12
base	0.70	3.81	11.58	84.96	662.99
prod	0.30	1.26	2.50	13.03	77.02
chgt-base	0.33	1.25	3.53	12.86	77.74

(c) Enveloppes convexes $H_n(0,1) \sqcup H_n(1,3)$, résultat non factorisé

Dimension	19	20	21	22	23	24	25
base	1.96	10.72	45.56	196.82	>600	>600	>600
prod	2.12	11.19	46.77	196.79	>600	>600	>600
chgt-base	0.12	0.28	0.77	3.15	23.78	108.53	442.63

(d) Enveloppes convexes $K_n(0,1) \sqcup K_n(0,2)$, factorisables par changement de base, résultat factorisable

Dimension	19	20	21	22	23	24	25
base	3.49	13.62	60.32	235.67	>600	>600	>600
prod	5.51	18.07	65.09	245.04	>600	>600	>600
chgt-base	0.13	0.28	0.77	3.22	23.59	109.28	445.17

(e) Enveloppes convexes $K_n(0,1) \sqcup K_n(1,2)$, factorisables par changement de base, résultat non factorisable

Dimension	10	20	30	40	50
base	0.01	0.03	0.02	0.06	0.13
prod	0.00	0.03	0.03	0.10	0.17
chgt-base	0.02	0.03	0.05	0.09	0.10

(f) Enveloppes convexes $S_n(0,1) \sqcup S_n(1,2)$ de simplexes (non factorisables)

TAB. 9.2 – Calcul de familles d'opérations, avec factorisation et changement de base

dimension	40-49	50-59	60-69	70-79	80-89	90-99	100-9	110-9	120-9	130-9	150-9
nb env	36	7	26	12	13	16	37	6	32	33	3
base/prod	1.8	1.9	2.1	2.1	4.0	4.4	4.7	5.7	6.0	6.0	7.6

TAB. 9.3 – Les résultat sur le “benchmark” issu de PIPS

	dim	base	prod	chgt-base
voiture	3	0.02	0.03	0.04
centralserv	13	1.44	2.09	2.84
firefly	4	0.68	1.50	2.11
lamport	11	1.63	1.78	1.97
lifo	7	0.57	1.11	1.37
rtp	9	0.54	0.62	0.71
swimingpool	9	0.30	0.48	0.59
ttp2	9	0.80	1.13	1.00
barber	8	1.82	2.29	1.75
consistency	12	3.97	10.99	27.33
consprodjava	18	3.55	6.08	7.56
readwrite	13	4.82	3.63	4.22
csm	13	11.33	5.04	5.55
fms	22	21.54	17.29	20.69
kanban	16	11.11	6.27	8.38
multipoll	17	16.14	4.11	4.78
peterson	14	69.28	36.76	38.20
barber efm	14	24.38	5.08	13.08
barber moesi	12	10.20	3.83	5.95
barber synapse	11	6.69	3.45	4.19
firefly moesi	8	11.42	3.75	4.79
barber mesi moesi	16	184.93	3.88	3.88
barber mesi synapse	15	57.20	4.11	12.40

TAB. 9.4 – Résultats de l’analyse des exemples simples, avec factorisation et changement de base

La figure 9.2 représente le gain en fonction du temps de calcul nécessaire à l'analyse pour les exemples de protocoles, du métro et du gyroscope. Ce graphique montre clairement que l'utilisation du produit cartésien ainsi que du changement de base est plus intéressant sur de gros exemples.

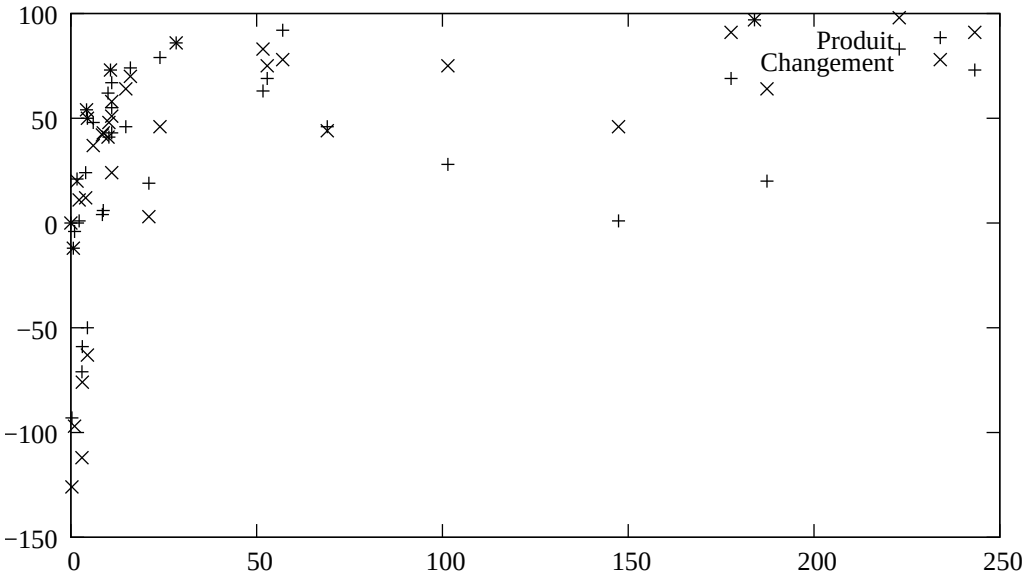


FIG. 9.2 – Réduction en pourcetage du temps d'execution

Le gain dépend du temps d'exécution mais il dépend aussi, et surtout, du nombre de variables manipulées. Les figures 9.3 et 9.4 représentent le gain en fonction de la dimension avec le produit cartésien et le changement de base. On peut remarquer une valeur étrange pour = 18, il s'agit d'un protocole dont la résolution se fait en moins de 4 secondes. Si lùon excepte ce cas particulier on voit clairement le croitre le gain avec la dimension.

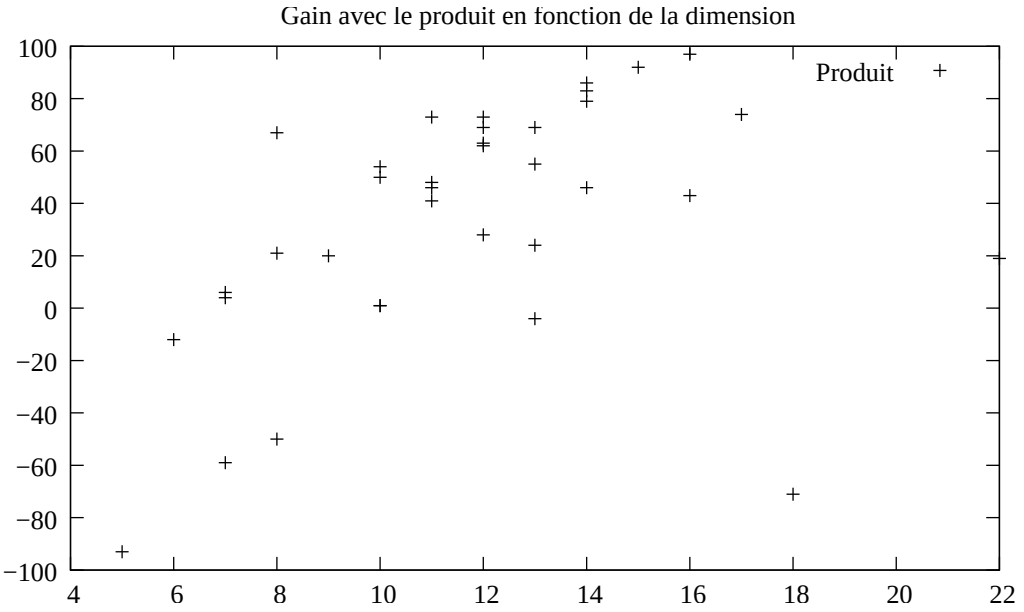


FIG. 9.3 – Réduction en fonction de la dimension avec le produit cartésien

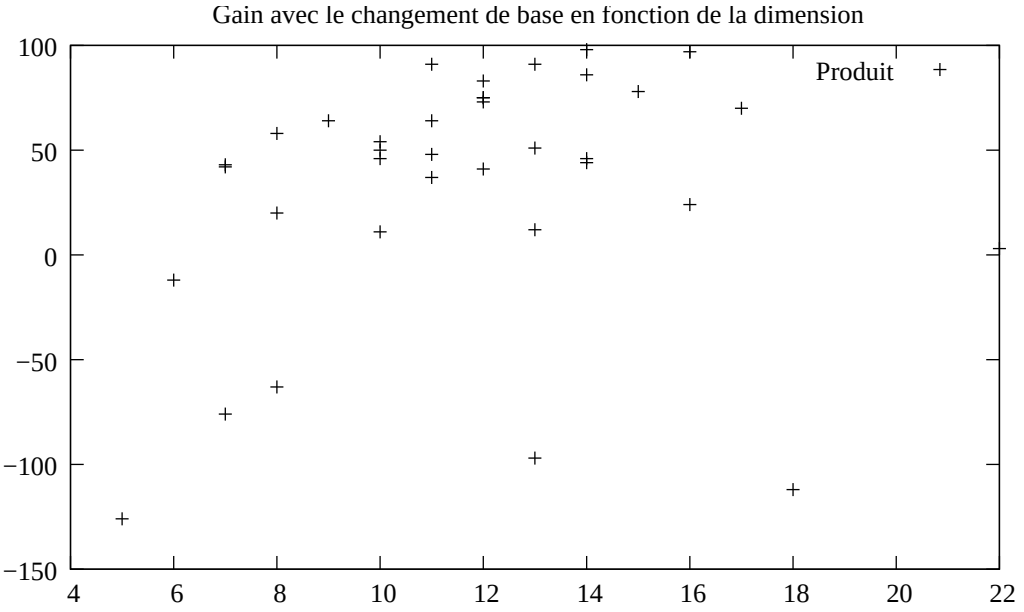


FIG. 9.4 – Réduction en fonction de la dimension avec changement de base

En conclusion, la factorisation est presque toujours intéressante à partir d'une dimension seuil qui se situe autour de 10 ou 12. En dessous de ce seuil, il est préférable de s'abstenir de factoriser. En complément de la factorisation, le changement de base est très souvent intéressant, et, dans le cas contraire, n'entraîne la plupart du temps qu'un surcoût négligeable.

	nb-trains	dim	base	prod	chgt-base
prop1	1	3	0.02	0.07	0.07
	2	5	0.30	0.58	0.68
	3	7	3.10	4.95	5.48
	4	9	187.36	151.02	65.81
prop2	1	4	0.04	0.12	0.11
	2	6	0.33	0.78	0.95
	3	8	4.45	6.71	7.27
	4	10	147.37	144.97	78.20
prop3	2	5	0.28	0.52	0.59
	3	7	8.46	8.12	4.84
	4	9	>600	>600	266.64
prop4	2	5	0.27	0.51	0.55
	3	7	8.74	8.15	4.91
	4	9	>600	>600	300.31
prop5	2	5	0.32	0.54	0.60
	3	7	8.52	8.49	5.33
	4	9	>600	>600	310.05

TAB. 9.5 – Résultats de l'analyse du métro

axes	dim	base	prod	chgt-base
1	6	0.66	0.74	0.74
2	8	1.65	1.30	1.31
3	10	4.22	1.94	1.94
4	12	10.65	2.86	2.82
5	14	28.35	3.93	3.90
6	16	>mem	5.07	5.00
7	18	>mem	6.37	6.31

(a) Variation du nombre d'axes

capteurs	dim	base	prod	chgt-base
3	9	0.55	0.78	0.67
4	10	2.22	2.18	1.96
5	11	10.25	6.02	5.23
6	12	52.86	16.05	13.18
7	13	>mem	44.74	33.76

(b) Variation du nombre de capteurs

version	dim	base	prod	chgt-base
3/4/1	10	4.44	2.20	2.22
3/4/2	14	222.92	37.15	3.54
3/5/1	11	14.81	7.89	5.28
3/6/1	12	51.69	18.95	8.74
3/7/1	13	177.66	53.89	15.03
2/4/2bis	12	101.45	72.12	25.07
3/4/1bis	22	>600	18.32	18.17
1/4/?	11	243.27	64.75	21.19

(c) Les versions décrites page 95

TAB. 9.6 – Résultats des analyses du gyroscope

Chapitre 10

Conclusion

Dans cet ouvrage, nous avons exploré quelques pistes pour optimiser les performances de l'analyse de relations linéaires, en diminuant le nombre de variables dans les opérations. L'idée d'utiliser les équations affines pour éliminer une variable parégalité et ainsi réduire le nombre de variables s'est révélée, contrairement à la décomposition en produit cartésien, sans intêret. L'approche fondée sur la factorisation cartésienne, et l'extension de son application par changement de base sont apparues tout-à-fait fructueuses. L'exepérimentation a montré qu'une fois atteint un seuil de dimensions manipulées, situé vers 10 variables, on obtient un ain important du temps d'exécution. Cette représenation des polyèdres sous forme de produits cartésiens à été implémentée dans une nouvelle bibliothèque de polyèdres. Nous avons également développé un analyseur de programmes Oc, qui a son intérêt propre, lorsqu'on s'intéresse à des programmes dont la structure de contrôle n'est pas trop explosive. Enfin, nous avons essayé dans ce mémoire d'expliciter des aspects techniques de l'analyse de relations linéaires — comme les algorithmes sur les polyèdres, l'élargissement limité, les stratégies d'élargissement — aspects qui sont rarement détaillés dans la littérature.

Concrètement, ce travail devrait être complété dans plusieurs directions :

- Concernant l'intégration des techniques proposées dans les bibliothèques existantes : comme nous l'avons dit, nous avons développé nos algorithmes comme une couche logicielle au dessus de la bibliothèque de Parme. Malheureusement, entre-temps, cette bibliothèque a subi des évolutions répétées et importantes, qu'il ne nous a pas été possible de suivre. Ainsi, nos programmes viennent-ils compléter la version 0.3 de la PPL, alors que celle-ci en est maintenant à la version 0.7, et que ces versions sont incompatibles. L'adaptation de nos programmes à la version la plus récente de la PPL est un prérequis à leur diffusion.
- Concernant l'intégration de nos programmes dans un analyseur à l'état de l'art : notre but était d'expérimenter rapidement nos algorithmes, et c'est pourquoi nous avons développé un prototype d'analyseur d'automates Oc, bien adapté à l'étude expérimentale de l'influence de nos propositions. Ceci dit, les possibilités de notre analyseur sont limitées par l'explosion de la taille de l'automate Oc. Il est donc

nécessaire, maintenant, d'interfacer nos programmes avec un analyseur comme Nbac.

– Tout au long de ce travail, nous avons développé des techniques d'optimisation *sans perte d'information*. Si l'on accepte de perdre de l'information, on peut aller plus loin dans les simplifications. Ainsi les techniques de "packing" [BCC+03] consistent à partitionner les calculs dans les analyses relationnelles, en oubliant certaines relations. Ce partitionnement peut être statique, en séparant d'emblée des classes de variables entre lesquelles on renonce à trouver des relations. Le partitionnement peut aussi être dynamique : si l'on découvre qu'il suffirait d'enlever quelques contraintes pour qu'un polyèdre très complexe se factorise, on peut choisir d'oublier ces contraintes. Enfin, le "packing" ne correspond pas nécessairement à un partitionnement : on peut effectuer des simplifications moins brutales en séparant les variables en classes non disjointes, les variables à l'intersection de deux classes permettant de garder certaines relations entre les classes.

Annexe A

Code des exemples

A.1 Code du Metro

```
const N=3;

node controleur(nB,nS:int) returns (diff:int; avance,retard:bool);
let
  diff = nB-nS;
  avance = false ->
        if not (pre avance) then diff >= 10 else diff >= 1;
  retard = false ->
        if not (pre retard) then (diff <= -10) else (diff <= (-1));
tel

node hypothese(B,S,avance,retard:bool) returns (ok:bool);
var c:int;
let
  ok = true ->
        (if pre retard then not S else true) and
        (if pre c >= 9 then not B else true);
  c = 0 ->
        if pre avance and pre (false -> pre avance) then
                if B then pre c + 1 else pre c
        else
                0;
tel

node main(B : bool^N; S : bool)
returns (ast: bool; nB : int^N; nS : int; diff:int^N; avance,retard : bool^N);
var H:bool^N;
```

```
let
  nB = 0^N -> if B then pre nB + 1^N else pre nB;
  nS = 0 -> if S then pre nS + 1 else pre nS;
  H = hypothese(B,S^N,avance,retard);
  (diff,avance,retard) = controleur(nB,nS^N);
  ast = AND(N,H);
tel
```

A.2 Code d'un gyroscope

```
const
        DELTA = 1;
        NBERR = 3;
        SAFE_COUNTER_TIME = 3;
        FAIL_SAFE_ROLL_VALUE = -100;

node abs (v : int) returns (a : int) ;
let
        a = if v >= 0 then v else -v ;
tel

node OlympicAverage (one, two, three, four : int) returns (m : int) ;
let
     m = (one + two + three + four
         - max4 (one, two, three, four)
         - min4 (one, two, three, four)) / 2 ;
tel

node MedianValue3 (a, b, c : int) returns (z : int);
let
        z = a + b + c - min2 (a, min2(b,c)) - max2 (a, max2(b,c)) ;
tel

node Average2(a, b: int) returns (z : int);
let
        z = (a+b)/2 ;
tel

node max4 (one, two, three, four : int) returns (m : int) ;
let
        m = max2 (max2 (one, two), max2 (three, four)) ;
```

```
tel

node max2 (one, two  : int) returns (m : int) ;
let
        m = if one > two then one else two ;
tel

node min4 (one, two, three, four : int) returns (m : int) ;
let
        m = min2 (min2 (one, two), min2 (three, four)) ;
tel

node min2 (one, two  : int) returns (m : int) ;
let
        m = if one < two then one else two ;
tel

node Average (x1, x2, x3, x4 : int ; f1, f2, f3, f4 : bool) returns (r : int)
-- two of the f are true, exactly
let
        r = if f1 then
              if f2 then Average2 (x3, x4)
              else if f3 then Average2 (x2, x4)
              else Average2 (x3, x2)
        else if f2 then
              if f1 then Average2 (x3, x4)
              else if f3 then Average2 (x1, x4)
              else Average2 (x3, x1)
        else if f3 then
              if f2 then Average2 (x1, x4)
              else if f4 then Average2 (x2, x1)
              else Average2 (x4, x2)
        else
              if f2 then Average2 (x3, x1)
              else if f3 then Average2 (x2, x1)
              else Average2 (x3, x2) ;
tel


node Median (x1, x2, x3, x4 : int ; f1, f2, f3, f4 : bool) returns (r : int)
-- one of the four f is true exactly
let
```

```
        r = if f1 then MedianValue3 (x2, x3, x4)
            else if f2 then MedianValue3 (x1, x3, x4)
            else if f3 then MedianValue3 (x1, x2, x4)
            else            MedianValue3 (x1, x2, x3) ;
tel


node Channel(
        val : int;
        erreur : bool)
returns(newVal : int; off : bool);
var
        nbErr : int;
let
        newVal = val + if(erreur) then 10 * DELTA else DELTA;

        nbErr = 0 -> if(pre (erreur or off)) then pre nbErr+1 else 0;
        off = (nbErr>=NBERR);
tel

node roll(
        roll : int;
        erreur1, erreur2, erreur3, erreur4 : bool;)
returns(nbOff, newRoll : int);
var
        roll1, roll2, roll3, roll4 : int;
        off1, off2, off3, off4 : bool;

let

        (roll1, off1) = Channel(roll, erreur1);
        (roll2, off2) = Channel(roll, erreur2);
        (roll3, off3) = Channel(roll, erreur3);
        (roll4, off4) = Channel(roll, erreur4);

        newRoll = if(nbOff = 0)
                then OlympicAverage (roll1, roll2, roll3, roll4)
                else(   if(nbOff = 1) then
                        Median (roll1, roll2, roll3, roll4, off1, off2, off3,
                                off4 ) else
                         Average (roll1, roll2, roll3, roll4, off1, off2, off3,
                                  off4 ) );
```

```
        nbOff = 0 + if(off1) then 1 else 0 + if(off2) then 1 else 0 + if(off3)
                then 1 else 0 + if(off4) then 1 else 0;

tel
```

Annexe B

Bibliographie

Bibliographie

[ACD93] R. Alur, C. Courcoubetis, and D.L. Dill. Model-checking in dense real-time. *Information and Computation*, 104(1) :2–34, 1993. Preliminary version appears in the Proc. of 5th LICS, 1990.

[ACIK97] C. Ancourt, F. Coelho, F. Irigoin, and R. Keryell. A linear algebra framework for static hpf code distribution. *Scientific Programming*, 6 :3–27, 1997.

[AD90] R. Alur and D. Dill. Automata for modeling real-time systems. In *ICALP'90*, 1990.

[AHU74] A. Aho, J. Hopcroft, and J. Ullman. *Design and analysis of computer Algorithms*. Addison Wesley, 1974.

[AS87] B. Alpern and F. B. Schneider. Recognizing safety and liveness. *Distributed Computing*, 2 :117–126, 1987.

[Avi98] D. Avis. Computational experience with the reverse search vertex enumeration algorithm. In *Optimization Methods and Software*, 1998.

[BBC+00] N. Bjorner, A. Browne, M. Colon, B. Finkbeiner, Z. Manna, H. Sipma, and T. Uribe. Verifying temporal properties of reactive systems : A STeP tutorial. *Formal Methods in System Design*, 16 :227–270, 2000.

[BCC+03] B. Blanchet, P. Cousot, R. Cousot, J. Feret, L. Mauborgne, A. Miné, D. Monniaux, and X. Rival. A static analyzer for large safety-critical software. In *PLDI 2003, ACM SIGPLAN SIGSOFT Conference on Programming Language Design and Implementation*, pages 196–207, San Diego (Ca.), June 2003.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model checking of infinite state systems using presburger arithmetic. In *Computer Aided Verification, CAV'97*. LNCS 1254, Springer Verlag, June 1997.

[BHRZ03] R. Bagnara, P. M. Hill, E. Ricci, and E. Zaffanella. Precise widening operators for convex polyhedra. In R. Cousot, editor, *Static Analysis : Proceedings of the 10th International Symposium*, volume 2694 of *Lecture Notes in Computer Science*, pages 337–354, San Diego, California, USA, 2003. Springer-Verlag, Berlin.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Invest : A tool for the verification of invariants. In Moshe Vardi Alan Hu, editor, *International Conference*

on Computer-Aided Verification , CAV'98, Vancouver, Canada, June 1998. LNCS vol. 1427 Springer-Verlag.

[Bou92] F. Bourdoncle. Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite. Thèse Ecole Polytechnique, Paris, 1992.

[BRdSV89] G. Boudol, V. Roy, R. de Simone, and D. Vergamini. Process calculi, from theory to practice : Verification tools. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[BRZH02] R. Bagnara, E. Ricci, E. Zaffanella, and P. M. Hill. Possibly not closed convex polyhedra and the parma polyhedra library. In M. V. Hermenegildo and G. Puebla, editors, *9th International Symposium on Static Analysis, SAS'02*, Madrid, Spain, September 2002. LNCS 2477.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *CAV'94*, Stanford (Ca.), 1994. LNCS 818, Springer Verlag.

[CC76] P. Cousot and R. Cousot. Static determination of dynamic properties of programs. In *2nd Int. Symp. on Programming*. Dunod, Paris, 1976.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : a unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *4th ACM Symposium on Principles of Programming Languages, POPL'77*, Los Angeles, January 1977.

[CC92] P. Cousot and R. Cousot. Abstract interpretation and application to logic programs. *Journal of Logic Programming*, 13(1–4) :103–179, 1992. (Also, Rapport de Recherche LIX/RR/92/08, Ecole Polytechnique).

[CC04] R. C. Clarisó and J. Cortadella. Verification of parametric timed circuits using octahedra. In *Designing correct circuits, DCC'04*, Barcelona, March 2004.

[CES86] E. M. Clarke, E. A. Emerson, and A. P. Sistla. Automatic verification of finite-state concurrent systems using temporal logic specifications. *ACM TOPLAS*, 8(2), 1986.

[CGL94] E. M. Clarke, O. Grumberg, and D. E. Long. Model checking and abstraction. *ACM TOPLAS*, 16(5) :1512–1542, 1994.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *5th ACM Symposium on Principles of Programming Languages, POPL'78*, Tucson (Arizona), January 1978.

[Che68] N. V. Chernikova. Algorithm for discovering the set of all solutions of a linear programming problem. *U.S.S.R. Computational Mathematics and Mathematical Physics*, 8(6) :282–293, 1968.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis and presburger arithmetic. In *CAV'98*, Vancouver (B.C.), 1998. LNCS 1427, Springer Verlag.

[Fer90] J.-C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2-3), May 1990.

[FP95] Komei Fukuda and Alain Prodon. Double description method revisited. In *Combinatorics and Computer Science*, pages 91–111, 1995.

[FS00] A. Finkel and G. Sutre. An algorithm constructing the semilinear post* for 2-dim reset/transfer vass. In *25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000)*, Bratislava, Slovakia, August 2000. LNCS 1893, Springer Verlag.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification and abstraction. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[gnu] The GNU Project. http ://www.gnu.org.

[Gra89] P. Granger. Static analysis of arithmetical congruences. *International Journal on Computer Mathematics*, 30 :165–190, 1989.

[Gra91] Ph. Granger. Analyses sémantiques de congruence. Phd thesis, Ecole Polytechnique, July 1991.

[Hal79a] N. Halbwachs. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Thèse de 3e cycle, Université de Grenoble, March 1979.

[Hal79b] N. Halbwachs. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Thèse de troisième cycle, Université de Grenoble, March 1979.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), July 1993. LNCS 697, Springer Verlag.

[HCRP91] N. Halbwachs, P. Caspi, P. Raymond, and D. Pilaud. The synchronous data-flow programming language LUSTRE. *Proceedings of the IEEE*, 79(9) :1305–1320, September 1991.

[HLR92] N. Halbwachs, F. Lagnier, and C. Ratel. Programming and verifying real-time systems by means of the synchronous data-flow programming language LUSTRE. *IEEE Transactions on Software Engineering, Special Issue on the Specification and Analysis of Real-Time Systems*, pages 785–793, September 1992.

[HLR93] N. Halbwachs, F. Lagnier, and P. Raymond. Synchronous observers and the verification of reactive systems. In M. Nivat, C. Rattray, T. Rus, and G. Scollo, editors, *Third Int. Conf. on Algebraic Methodology and Software Technology, AMAST'93*, Twente, June 1993. Workshops in Computing, Springer Verlag.

[HMPV03] N. Halbwachs, D. Merchat, and C. Parent-Vigouroux. Cartesian factoring of polyhedra in linear relation analysis. In *Static Analysis Symposium, SAS'03*, San Diego, June 2003. LNCS 2694, Springer Verlag.

[HNSY92] T. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. In *LICS'92*, June 1992.

[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.

[HRR91] N. Halbwachs, P. Raymond, and C. Ratel. Generating efficient code from data-flow programs. In *Third International Symposium on Programming Language Implementation and Logic Programming*, Passau (Germany), August 1991. LNCS 528, Springer Verlag.

[IJT91] F. Irigoin, P. Jouvelot, and R. Triolet. Semantical interprocedural parallelization : An overview of the PIPS project. In *ACM Int. Conf. on Supercomputing, ICS'91, Köln*, 1991.

[IN04] F. Irigoin and D. Nguyen. Private communication, 2004.

[Jea00] B. Jeannet. Partitionnement dynamique dans l'analyse de relations linéaires et application à la vérification de programmes synchrones. These, Institut National Polytechnique, Grenoble, September 2000.

[Jea03] B. Jeannet. Dynamic partitioning in linear relation analysis, application to the verification of synchronous programs. *Formal Methods in System Design*, 23(1) :5–37, July 2003.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In A. Cortesi and G. Filé, editors, *Static Analysis Symposium, SAS'99*, Venice (Italy), September 1999. LNCS 1694, Springer Verlag.

[Kar76] M. Karr. Affine relationships among variables of a program. *Acta Informatica*, 6 :133–151, 1976.

[Lam77] L. Lamport. Proving the correctness of multiprocess programs. *IEEE Transactions on Software Engineering*, SE-3(2) :125–143, 1977.

[Lam80] L. Lamport. "Sometime" is sometimes "Not Never"'. In *Seventh ACM Symposium on Principles of Programming Languages*, January 1980.

[Ler03] J. Leroux. Algorithmique de la vérification des systèmes à compteurs – approximation et accélération – implémentation dans l'outil Fast. Phd thesis, Ecole Normale Supérieure de Cachan, December 2003.

[LeV92] H. LeVerge. A note on Chernikova's algorithm. Rapport de Recherche 635, IRISA, February 1992.

[LMC02] V. Loechner, B. Meister, and Ph. Clauss. Precise data locality optimization of nested loops. *The Journal of Supercomputing*, 21(1) :37–76, January 2002.

[LMQ91] H. LeVerge, Ch. Mauras, and P. Quinton. The alpha language and its use for the design of systolic arrays. *Journal of VLSI Signal Processing Systems*, 3(3) :173–182, September 1991.

[lsv] http ://www.lsv.ens-cachan.fr/fast/.

[Mer91] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics, MFPS'91*, pages 312–325, 1991.

[Min01] A. Miné. The octagon abstract domain. In *AST 2001 in WCRE 2001*, IEEE, pages 310–319. IEEE CS Press, October 2001.

[MT53] Thompson Motzkin, Raiffa and Thrall. The double description method. *Theodore S. Motzkin : Selected Papers*, 1953.

[Muc97] S. S. Muchnick. *Advanced Compiler Design Implementation*. Morgan Kaufmann Pub., 1997.

[NIAK01] N. V. Nguyen, F. Irigoin, C. Ancourt, and R. Keryell. Efficient intraprocedural array bound checking. In *Second International Workshop on Automated Program Analysis, Testing and Verification, WAPATV'01*, Toronto, Canada, May 2001.

[NR00] Sunder Phani Kumar Nookala and Tanguy Risset. A library for Z-polyhedral operations. Technical Report 1330, Irisa, Mai 2000.

[Pod04] A. Podelski. A complete method for synthesis of linear ranking functions. In *Verification, Model Checking and Abstract Interpretation, VMCAI'04*, Venice, Italy, January 2004.

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

[PS87] J. A. Plaice and J-B. Saint. The LUSTRE-ESTEREL portable format. Rapport non publié, INRIA, Sophia Antipolis, 1987.

[QS82] J. P. Queille and J. Sifakis. Specification and verification of concurrent systems in Cesar. In *International Symposium on Programming*. LNCS 137, Springer Verlag, April 1982.

[SSM05] S. Sankaranarayanan, H. Sipma, and Z. Manna. Scalable analysis of linear systems using mathematical programming. In R. Cousot, editor, *6th International Conference on Verification, Model-checking, and Abstract Intepretation, VMCA'05*, Paris, January 2005. LNCS 3385, Springer Verlag.

[Tar72] R. E. Tarjan. Depth-first search and linear graph algorithms. *SIAM Journal on Computing*, 1 :146–160, 1972.

[Tip95] F. Tip. A survey of program slicing techniques. *Journal of Programming Languages*, 3(3) :121–189, September 1995.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Rapport de Recherche 785, IRISA, December 1993.

Résumé

Cette thèse s'inscrit dans la vérification automatique de propriétés numériques de programmes, principalement des logiciels embarqués. Lors de la vérification on doit représenter de façon finie des ensembles éventuellement infinis de valeurs, pour cela une solution possible est l'utilisation de polyèdres convexes. Cette représentation est précise mais coûteuse ce qui limite le nombre de variables qu'il est possible de manipuler. Le but de cette thèse est d'augmenter le nombre maximal de variables qu'il est possible de représenter. Deux approches ont été envisagées puis testées. Dans un premier temps on a voulu tirer profit de la présence d'équations affines pour éliminer une variable par équation. Cette approche s'est révélée, expérimentalement, assez décevante. Une autre approche, bien plus prometteuse, est l'utilisation du produit cartésien. L'idée est alors de représenter indépendamment les variables dont l'évolution n'est pas liée. Cette décomposition peut être améliorée grâce à un changement de base. Un analyseur a été réalisé afin de tester ces deux approches.

Mots-clés : Interprétation abstraite, polyèdres convexes, analyse de relations linéaires

Abstract

This thesis takes place in automatic verification of numerical properties, mainly for embeded softwares. During verification one must represents in a finished way possibly infinite sets of values, for that a solution is use of convex polyhedra. This representation is precise but expensive so maximal number of variables is limited. The goal of this thesis is to increase the maximum number of variables which it is possible to represent. Two approaches were considered then tested. First we wanted to benefit from the presence of linear equations to eliminate a variable by equation. This approach appeared, in experiments, rather disappointing. Another apprroach, more promising, is the use of cartesian product. The idea is then to represent independently the variables whose evolution is not dependent. This decomposition can be improved through a change of basis. An analyser was done in order to test these two approaches.

Keywords : Abstract interpretation, convex polyhedra, linear relation analysis