INSTITUT NATIONAL POLYTECHNIQUE DE GRENOBLE

THÈSE

pour obtenir le grade de

DOCTEUR DE L'INPG

Spécialité: « INFORMATIQUE: SYSTÈMES ET COMMUNICATION »

préparée au laboratoire **Vérimag**
dans le cadre de l'**École Doctorale « MATHÉMATIQUES, SCIENCES ET TECHNOLOGIE DE L'INFORMATION »**

présentée et soutenue publiquement
par

Bertrand JEANNET

le 25 Septembre 2000

TITRE

Partitionnement Dynamique dans l'Analyse de Relations Linéaires et Application à la Vérification de Programmes Synchrones

Directeur de thèse

Nicolas Halbwachs

JURY

M. Paul Jacquet,	Président
M. Patrick Cousot,	Rapporteur
M. Andreas Podelski,	Rapporteur
M. Nicolas Halbwachs,	Directeur de thèse
M. Pascal Raymond,	Co-encadrant
M. Hans Toetenel,	Examinateur
M. Christophe Mauras,	Examinateur

Table des matières

Chapitre 1

Introduction

1.1 Contexte et motivation

La complexité des logiciels informatiques, croissant à un rythme soutenu depuis la naissance de cette discipline, pose le problème de la confiance que l'on peut avoir sur leur bon fonctionnement : satisfont-ils leurs cahiers des charges ? Ceux-ci sont-ils bien complets ? Bref, ces logiciels accomplissent-ils bien la tâche que l'on attend d'eux avec toutes les garanties de sûreté nécessaires ?

Bien sûr, les notions élaborées en théorie des langages de programmation et en génie logiciel ont permis d'accompagner cette évolution et de maîtriser la complexité des programmes. Il n'en est pas moins vrai que, d'une part les logiciels informatiques se généralisant (à quantité d'appareils domestiques par exemple), d'autre part étant de plus en plus utilisés dans des systèmes critiques (pilotage de véhicules, centrales nucléaires, ...), le besoin de méthodes rigoureuses, voire exhaustives pour s'assurer de la correction des logiciels *après leur conception* se fait sentir fortement dans l'industrie. Cette activité de *validation* présente non seulement un aspect de maîtrise des risques — des vies humaines sont clairement en jeu dans le cas de centrales nucléaires ou de métros automatiques —, mais aussi un enjeu économique très important.

Les deux grandes techniques de validation sont le *test* et la *vérification*. Le *test* consiste à soumettre un systèmes à une situation donnée et à vérifier si son comportement observé a un sens, ou d'une manière plus rigoureuse si il est conforme à ses spécifications. Il permet de détecter des erreurs ou d'établir un certain niveau de confiance dans la fiabilité du système. La *vérification* se distingue du test en ce qu'elle vérifie que le système est conforme à ses spécifications *quelle que soit la situation* à laquelle il est soumis. La vérification permet de *prouver* qu'un logiciel possède une certaine propriété, mais est bien sûr plus difficile à mettre en oeuvre, contrepartie naturelle de sa puissance. La vérification peut se faire soit par des *méthodes déductives* — mais semi-automatiques, qui permettent d'exhiber une démonstration de la propriété, soit par des *méthodes algorithmiques*, entièrement automatiques mais plus limitées dans leur champ d'application.

1.2 Sujet de la thèse

Cette thèse a pour objet la vérification par des *méthodes algorithmiques* de propriétés de sûreté de systèmes synchrones, ou plus généralement réactifs, présentant à la fois des aspects *booléens* et *numériques*. En effet on distingue généralement dans ces systèmes la structure de contrôle discrète, qui est donnée soit implicitement par des variables booléennes ou de type fini, soit explicitement par un automate, et le comportement des variables numériques.

Des méthodes éprouvées permettent de traiter ces aspects séparément :

- on peut en effet utiliser les méthodes de vérifications des sytèmes finis sur la structure de contrôle discrète, en abstrayant le comportement numérique ; naturellement, le fait d'ignorer le comportement numérique empêche la preuve de certaines propriétés, mais autorise l'utilisation de méthodes symboliques pour la partie discrète ;
- dans le cas où l'aspect booléen est petit, la structure de contrôle explicite est de taille raisonnable et on peut utiliser les techniques d'interprétation abstraite de domaines numériques, qui vont calculer en chacun des points de la structure de contrôle un sur-ensemble des valeurs numériques possibles en ce point, ce qui permet éventuellement de conclure qu'une propriété est vérifiée. Une limitation évidente de cette approche est la taille de la structure de contrôle, qui croît exponentiellement avec le nombre de variables booléennes du système vérifié.

Traiter les aspects booléens et numériques de manière *couplée* et *symbolique* est en revanche beaucoup plus délicat. Cette thèse propose une telle technique, et se distingue d'autres propositions par la conviction que les approximations sont fondamentales dans la vérification de systèmes de complexité moyenne ou élevée. Dans ce cadre, la technique proposée tente de prouver des propriétés en commençant par faire des approximations grossières, puis les affine automatiquement jusqu'à obtention de la preuve de la propriété, ou jusqu'à ce que la précision maximale soit atteinte.

1.3 Cadre théorique

Ce travail a pour cadre théorique l'Interprétation Abstraite [CC77], qui est une théorie de la résolution approchée d'équations de point fixe appliquée à l'analyse de programmes. Ce cadre théorique est très naturel lorsqu'il s'agit d'effectuer des analyses approchées et/ou de transformer des problèmes de décision indécidables en problèmes semi-décidables. De plus il garantit automatiquement la correction de nos algorithmes de vérification lorsque les hypothèses de base sont vérifiées.

Le problème de la vérification de propriétés de sûreté sur des programmes synchrones possédant des variables booléennes et numériques est indécidable. La solution pour surmonter cette première difficulté est d'user d'*approximations*, qui permettent d'obtenir un semi-algorithme, dans le sens suivant : un algorithme qui répond soit “oui”, soit “je ne sais pas”, au problème de décision. Le caractère indécidable de la vérification de propriétés de sûreté sur les programmes étudiés est donc la première motivation de l'usage d'approximations. Celles-ci sont induites par l'approximation d'ensembles d'états concrets par des valeurs abstraites manipulables algorithmiquement, ainsi que par l'emploi d'extrapolations pour assurer la terminaison des algorithmes.

La seconde motivation de l'emploi d'approximations est, comme nous l'avons dit ci-dessus, la réduction de la complexité en temps et en espace des algorithmes. Dans le cadre de l'Interprétation Abstraite, nous réglons le degré d'approximations des analyses en *partitionnant* l'espace d'états du programme analysé. Ce partitionnement permet d'approximer les ensembles d'états concrets par des *unions finies* et *de cardinal borné* de valeurs abstraites, au lieu d'utiliser une unique valeur abstraite pour ce faire. Par ailleurs, pour un programme donné il est souhaitable que la preuve d'une propriété simple soit plus facile (en terme de complexité en temps et en espace) qu'une propriété plus subtile.

Dans ce contexte, la reformulation de notre méthode de vérification consiste à débuter l'analyse d'un programme en utilisant un partitionnement très grossier de son espace d'état, fondé sur la propriété à montrer, et en cas d'échec à utiliser les informations collectées par l'analyse pour raffiner le partitionnement initial en un partitionnement plus fin et autorisant des analyses plus précises. Ce processus est répété jusqu'au succès de la preuve, ou jusqu'à ce que l'on ne puisse plus raffiner de manière raisonnable la partition obtenue.

1.4 Contributions

Les principaux résultats pratiques obtenus sont d'abord la réalisation d'un outil, permettant la vérification de propriétés non triviales sur des programmes synchrones de taille moyenne ou même élevée, possédant plusieurs dizaines de variables booléennes et un nombre de variables numériques de l'ordre de la dizaine. À notre connaissance, aucun autre outil automatique n'est capable de traiter des programmes de cette complexité, sauf peut-être les outils de vérification travaillant sur les automates temporisés ; cette dernière classe de système a toutefois une expressivité beaucoup plus réduite que celle des programmes synchrones que nous traitons : le problème de la vérification de propriétés de sûreté y est décidable, et le comportement numérique beaucoup plus simple.

La réalisation de cet outil a accessoirement nécessité la réalisation d'une bibliothèque de manipulation des polyèdres convexes plus robuste que celle auparavant utilisée, autorisant le choix de la représentation des entiers et apportant deux améliorations : mémorisation de données intermédiaires évitant des recalculs, et mise en œuvre d'heuristiques provenant d'autres bibliothèques.

D'un point de vue algorithmique, la thèse apporte les contributions suivantes :

1. elle propose une méthode originale pour calculer les *postconditions* (et *préconditions*) de valeurs abstraites par la fonction de transition induite par le programme. Cette méthode permet de faire usage d'approximations dans ces calculs, et permet un gain important en temps et en espace sans pénaliser sensiblement la précision. Elle repose sur le constat que lorsque la grossièreté de la partition de l'espace d'états du programme limite la précision des analyses, il est inutile de calculer les postconditions avec précision. La méthode proposée est très générale et est applicable à d'autres domaines de valeurs abstraites.

2. concernant les analyses d'accessibilité que nous menons sur des programmes partitionnés, qui font appel à un opérateur d'extrapolation pour garantir la convergence de l'analyse, nous étudions différentes améliorations envisageables dans la définition et l'application de cet opérateur, et examinons expérimentalement l'influence de ces perfectionnements sur la précision des analyses.

3. enfin, après avoir recensé les différents critères de raffinement de partition envisageables ou expérimentés dans d'autres travaux, nous proposons les heuristiques de raffinement implémentées dans notre outil et qui se révèlent expérimentalement très efficaces.

1.5 Plan

La partie I expose notre problématique. Nous présentons d'abord dans le chapitre 2 la théorie de l'interprétation abstraite et son application à la vérification de propriétés numériques de programmes. La lecture de ce chapitre n'est pas indispensable au lecteur au fait de cette théorie.

La thèse a été développée dans le cadre de la programmation synchrone [Hal93b], bien que ses idées s'appliquent de manière immédiate à d'autres sytèmes réactifs comme les automates hybrides linéaires [ACH+95]. Aussi nous poursuivons au chapitre 3 par une présentation de la programmation synchrone et des méthodes de vérification développées pour vérifier des propriétés de sûreté de tels programmes. Ceci nous permet au chapitre 4 d'exposer précisément les limitations des méthodes existantes, et nous présentons le principe du partitionnement dynamique, qui est l'objet de la thèse.

Le chapitre 4 décrit les idées principales de notre technique de vérification. Néanmoins, la bonne conception des algorithmes mettant en œuvre ces idées est vitale pour leur succès pratique. Les chapitres de la partie II sont donc plus techniques et décrivent en détail les structures de données et les algorithmes utilisés pour vérifier un programme.

Le chapitre 5 aborde le problème des structures de données utilisées pour représenter fonctions et formules, ainsi que le choix d'un domaine abstrait approprié à la mise en œuvre du partitionnement dynamique. Le chapitre 6 définit des opérations spécifiques sur les fonctions et les prédicats abstraits, que nous utiliserons dans les chapitres suivants.

La manière originale avec laquelle nous calculons de manière approchée les postconditions et les préconditions sur notre domaine abstrait est décrite dans le chapitre 7. Ces algorithmes prennent également une part importante dans la mise en oeuvre du raffinement de la structure de contrôle. Les analyses d'accessibilité que nous menons sur une structure de contrôle sont ensuite abordés au chapitre 8.

Armés de tous ces outils, nous pouvons enfin attaquer la description des heuristiques de raffinement dans le chapitre 9.

La partie III s'attache à présenter la réalisation pratique de notre outil et les résultats expérimentaux obtenus. Le chapitre 10 décrit l'implémentation des différents outils dont nous avons eu besoin, et des possibilités offertes par notre outil de vérification nommé NBAC, tandis que le chapitre 11 décrit l'utilisation de cet outil sur un certain nombre de programmes, dont une étude de cas d'un projet de recherche européen. Nous analysons également le comportement interne de l'outil et justifions ainsi les choix effectués dans la partie II.

Enfin le chapitre 12 conclut ce travail et en présente quelques perspectives.

Première partie

Présentation du problème et principe du partitionnement dynamique

Chapitre 2

Systèmes de transitions, interprétation abstraite et domaines numériques

L'Interprétation Abstraite est une théorie de la résolution approchée d'équations de point fixe appliquée à l'analyse de programmes. Nous en rappelons les concepts de base avant de décrire ses applications dans l'analyse de propriétés de variables numériques de programme. Nous présentons en particulier des exemples importants d'interprétation abstraite de domaines numériques.

Nous définissons dans un premier temps au §2.1 la notion de système de transitions discret. Nous rappelons au §2.2 la distinction importante entre propriété de sûreté et propriété de vivacité, la thèse ne s'intéressant qu'à la vérification du premier type de propriétés. La théorie de l'interprétation abstraite est décrite au §2.3. Enfin, nous donnons des exemples importants de domaines numériques abstraits au §2.4.

2.1 Systèmes de transitions discrets

Lorsqu'on souhaite analyser un programme, on en considère une forme abstraite. La plupart des programmes ou systèmes peuvent être formalisés par des *systèmes de transitions* (ou *systèmes dynamiques discrets*).

Définition 1 *Un* système de transitions discret *est un triplet* (Q, τ, Q_{init}) *dans lequel* Q *est ensemble d'*états*,* $\tau \subset Q \times Q$ *est la* relation de transition *entre un état et ses successeurs possibles,* $Q_{init} \subset Q$ *est l'ensemble des états* initiaux.

Un système de transitions définit un graphe orienté $G = (S, \rightarrow)$, dont les sommets sont les états du système ($S = Q$) et dont les arcs sont définis par $q \rightarrow q' \Leftrightarrow (q, q') \in \tau$; aussi la terminologie utilisée est souvent inspirée de celle de la théorie des graphes.

Définition 2 *Un* chemin *de l'état* q *à l'état* q' *est une suite* $(q_0, q_1, \ldots, q_n)$ *tel que* $q = q_0$ *et* $q' = q_n$ *et* $\forall i < n\ (q_i, q_{i+1}) \in \tau$*. Une* exécution *est un chemin éventuellement infini* $(q_0, q_1, \ldots)$ *tel que* $q_0 \in Q_{init}$ *et* $\forall i \geqslant 0\ (q_i, q_{i+1}) \in \tau$*.*

Pour tout ensemble d'états $X \subset Q$, on note respectivement

$$post(\tau)(X) = \{s' \mid \exists s \in X \wedge (s,s') \in \tau\}$$
$$pre(\tau)(X) = post(\tau^{-1})(X) = \{s \mid \exists s' \in X \wedge (s,s') \in \tau\}$$

les ensembles de *successeurs* et de *prédécesseurs* des états appartenant à X par la relation de transition. *post* et *pre* sont appelés respectivement opérateurs de *postcondition* et *précondition.*

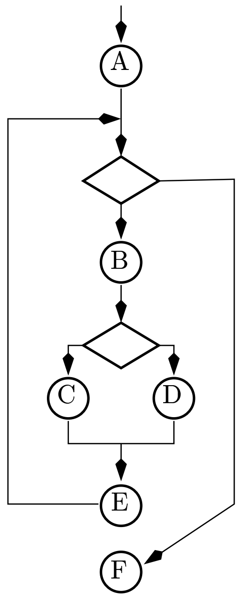
L'ensemble des états accessibles depuis un ensemble d'états $X \subset Q$ est noté $reach(\tau)(X) = post^*(\tau)(X)$, où $post^*(\tau)(X)$ est défini par

$$post^*(\tau)(X) = \cup_{n \geqslant 0} post^n(\tau)(X)$$
$$post^0(\tau)(X) = X \quad , \quad post^{n+1}(\tau)(X) = post(\tau) \circ post^n(\tau)(X)$$

On peut de même définir l'ensemble des états pouvant mener à un ensemble d'états $X \subset Q$, noté $coreach(\tau)(X) = pre^*(\tau)(X) = post^*(\tau^{-1})(X)$.

```
program PL;
  var I,J : integer;
begin
  I := 2; J := 0;
  (A)
  while I+J <= 20 do begin
    (B)
    if I >= 10 then begin
      I := I+4;
      (C)
    end else begin
      I := I+2; J := J+1;
      (D)
    end ;
    (E)
  end ;
  (F)
end.
```

(a) Programme



(b) Graphe de contrôle

FIG. 2.1 – Programme PASCAL et sa structure de contrôle

EXEMPLE. — Le programme PASCAL de la fig 2.1 définit un système dynamique avec $Q = \{A,B,C,D,E,F\} \times \mathbb{Z} \times \mathbb{Z}$, $Q_{init} = \{(A,2,0)\}$, et τ induit par la sémantique des opérations. Les valeurs possibles des variables entières au point D est par exemple

$$reach(\tau)(Q_{init}) \cap (\{D\} \times \mathbb{Z} \times \mathbb{Z}) = \Big\{(I,J) \mid I = 2 + 2J \ \wedge\ 0 \leqslant J \leqslant 4\Big\}$$

□

2.2 Systèmes de transitions discrets et propriétés de sûreté

On distingue deux grands types de propriétés sur les systèmes de transitions [Lam77], dont la vérification fait appel à des techniques différentes. Informellement,

Les propriétés de sûreté spécifient que “quelque chose de mauvais” ne se produit jamais au cours de l'exécution du système ; l'absence de dépassements arithmétiques, la garantie de service dans un délai borné sont des exemples de propriétés de sûreté ; ces propriétés sont caractérisées par le fait que leur violation est détectable par une exécution finie du système.

Les propriétés de vivacité ou d'inévitabilité spécifient que “quelque chose de bon” se produit immanquablement durant l'exécution du système ; des exemples de telles propriétés sont la terminaison ou la garantie de service ; ces propriétés sont caractérisées par le fait qu'elles portent sur un futur non borné et ne peuvent donc pas être violées par une exécution *finie* du système.

Les formules de logiques temporelles utilisées en vérification de programmes permettent de spécifier des propriétés générales, dont on montre qu'elles se décomposent en la conjonction d'une propriété de sûreté et une propriété de vivacité [AS87, BFG+91].

Nous ne nous intéressons dans cette thèse qu'à la vérification de propriétés de sûreté. Dans le contexte des systèmes de transitions, une propriété de sûreté peut toujours se ramener, par le biais d'un *observateur*, à la donnée d'un ensemble d'états *erronés* dont on doit montrer l'inaccessibilité. Nous utiliserons donc des obervateurs, et plus spécifiquement des *observateurs synchrones* (§3.3.2) pour spécifier les propriétés de sûreté.

Soit (Q, τ, Q_{init}) un système de transitions et Q_{err} une propriété. La preuve de cette propriété peut se faire de deux manières équivalentes :

- par une analyse *en avant*, en montrant que l'ensemble des états accessibles et Q_{err} sont disjoints : $reach(\tau)(Q_{init}) \cap Q_{err} = \emptyset$;
- ou par une analyse *en arrière*, en montrant que l'ensemble des états pouvant mener à un état erroné et l'ensemble des états initiaux sont disjoints : $coreach(\tau)(Q_{err}) \cap Q_{init} = \emptyset$.

Notons C le treillis complet que forme l'ensemble des parties de S. Ces analyses se réduisent à la résolution d'équations de point fixe de fonctions monotones dans le treillis C : en effet pour tout $I \in C$,

$$\begin{array}{lll} reach(\tau)(I) = post^*(\tau)(I) = lfp(F) & \text{avec} & F(X) = I \cup post(\tau)(X) \\ coreach(\tau)(I) = pre^*(\tau)(I) = lfp(F) & \text{avec} & F(X) = I \cup pre(\tau)(X) \end{array}$$

Lorsque C est un domaine trop complexe, on utilise les techniques de l'interpétation abstraite pour calculer des approximations de ces points fixes.

2.3 Interprétation Abstraite

2.3.1 Les deux principes de base

La plupart des analyses de programmes se réduisent à la résolution d'une équation de point fixe $x = F(x), x \in C$ où C est un domaine ordonné ; l'analyse en avant ou en arrière de propriété en sont des exemples.

La résolution d'une telle équation pose deux types de problèmes, pour chacun desquels l'article fondateur [CC77] propose une solution générale, et qui sont d'une part la complexité du domaine de calcul C et d'autre part la résolution itérative de l'équation de point fixe.

Complexité du domaine de calcul C

C est généralement un domaine complexe, typiquement l'ensemble 2^S des parties de l'ensemble des états du programme S ordonné par la relation d'inclusion. Les éléments de ce domaine doivent être représentés d'une manière efficace et normalisée ; les fonctions définies sur le domaine et sa relation d'ordre doivent pouvoir être calculées.

Ces exigences ne sont en pratique pas toujours satisfaisables comme le montre l'exemple suivant.

EXEMPLE. — Considérons l'ensemble $\{(x, y, z, n) \in \mathbb{N}^4 \mid x^n + y^n = z^n \wedge n \geqslant 3\}$. La conjecture de Fermat consiste à savoir si cet ensemble est vide ou non, et n'a été résolue que récemment : il est clair que des ensembles ainsi définis ne sont pas manipulables algorithmiquement. $\square$

Afin de contourner cet obstacle une première approximation peut être faite en remplaçant le domaine C des *valeurs concrètes* par un *domaine abstrait* A plus simple, à condition que ces deux domaines soient reliés par deux fonctions α et γ formant une connexion de Galois :

Définition 3 (Connexion de Galois) *Soit $C(\sqsubseteq_C)$ et $A(\sqsubseteq_A)$ deux ensembles partiellement ordonnés, et des fonctions $\alpha : C \to A$ et $\gamma : A \to C$. (C, α, γ, A) forment une* connexion de Galois *si,*

$$\forall x \in C, \forall y \in A, \quad \alpha(x) \sqsubseteq_A y \Longleftrightarrow x \sqsubseteq_C \gamma(y)$$

α est la fonction d'*abstraction*, qui associe à une valeur concrète son approximation dans le domaine abstrait, tandis que γ est la fonction de *concrétisation* qui associe à une valeur abstraite sa signification concrète. Ce mécanisme d'abstraction se généralise aux opérateurs :

Définition 4 *Soit (C, α, γ, A) une connection de Galois et $f : C^n \to C$ une fonction monotone. On appelle* plus petite approximation supérieure de f dans A *ou simplement* abstraction standard de f dans A *la fonction*

$$\begin{array}{llll} \alpha(f) : & A^n & \to & A \\ & (x_1, \ldots, x_n) & \mapsto & \alpha\Big(f(\gamma(x_1), \ldots, \gamma(x_n))\Big) \end{array}$$

Toute fonction $g \sqsupseteq \alpha(f)$ sera appelée approximation supérieure de f dans A

Les domaines abstraits que nous allons considérer seront tous définis de la manière suivante.

CAS IMPORTANT : DOMAINE ABSTRAIT DÉFINI PAR UNE FAMILLE DE MOORE. — On se place dans le cadre de l'analyse d'un système de transitions discret, pour lequel $C = 2^S$. Une méthode répandue pour définir un treillis abstrait A est de choisir une famille de Moore de C, i.e., un sous-ensemble $A \subset C$ fermé par intersection. Lorsque $C = \mathcal{N}^n$ est un espace numérique, des exemples de familles de Moore sont les hyperrectangles ou les polyèdres convexes de C. On montre que toute famille de Moore du domaine concret C définit ainsi un domaine abstrait et une connexion de Galois entre ces domaines, avec $\gamma = id$, $\alpha(x) = \bigcap\{y \in A \mid x \subset y\}$, et $y_1 \sqcup y_2 = \alpha(y_1 \cup y_2)$: une valeur concrète est approchée par la plus petite valeur abstraite qui la contient. $\square$

Le résultat fondamental est le suivant :

Théorème 1 *Soit (C, α, γ, A) une connexion de Galois et $F : C \to C$. Si C est un treillis complet et F est monotone, alors*

$$\alpha(\mathit{lfp}(F)) \sqsubseteq_A \mathit{lfp}(\alpha(F))$$

En particulier, pour toute approximation supérieure G de F dans A, $\alpha(\mathit{lfp}(F)) \sqsubseteq_A \mathit{lfp}(G)$.

Ce théorème signifie que le plus petit point fixe d'une approximation de F dans le domaine abstrait est une approximation supérieure du plus petit point fixe de F dans le domaine concret. Le résultat le plus précis est obtenu par l'emploi de $\alpha(F)$, mais toute fonction supérieure à cette approximation peut-être utilisée, par exemple pour des raisons algorithmiques.

Résolution itérative de l'équation de point fixe

Une fois le domaine concret simplifié, demeure le problème de la résolution de l'équation de point fixe. Celle-ci se fait de manière itérative, en calculant la limite de la suite

$$x_0 = \bot_A \quad , \quad x_{n+1} = G(x_n) \tag{2.1}$$

où G est une approximation supérieure de F. Lorsque le treillis A est de profondeur finie n, la suite converge en un nombre fini d'itérations $k \leqslant n$. Cependant imposer une telle exigence au domaine abstrait est très restrictif. De meilleurs résultats peuvent être obtenus en autorisant un domaine A de profondeur infinie et en effectuant un autre type d'approximation, qui consiste à extrapoler la limite de la séquence $(x_n)_{n\geqslant 0}$ [CC76, CC92b] . Dans le cas d'une suite croissante, on utilise un opérateur d'*élargissement* (*widening* en anglais) :

Définition 5 *Un* opérateur d'élargissement *est une fonction* $\nabla : A \times A \to A$*, satisfaisant les propriétés suivantes :*

1. $\forall y_1, y_2 \in A,\ y_1 \sqcup_A y_2 \sqsubseteq_A y_1 \nabla y_2$ *;*
2. *Pour toute suite croissante* $(x_n)_{n\geqslant 0}$*, la suite croissante définie par* $y_0 = x_0$*,* $y_{n+1} = y_n \nabla x_{n+1}$ *converge en un nombre fini d'itérations.*

Pour calculer une approximation de la limite $\bar{x}$ de la suite (2.1), on peut calculer la suite $(y_n)_{n\geqslant 0}$:

$$y_0 = \bot_A \quad , \quad y_{n+1} = y_n \nabla G(y_n) \tag{2.2}$$

qui converge en un nombre fini d'itérations vers une approximation supérieure $\bar{y}$ de $\bar{x} = \mathit{lfp}(G)$.

Cette approximation peut-être améliorée, i.e., rendue plus précise, d'une part en n'appliquant l'opérateur d'élargissement qu'à partir d'un certain rang, et d'autre part en calculant les premiers termes de la suite décroissante :

$$z_0 = \bar{y} \quad , \quad z_{n+1} = G(z_n) \tag{2.3}$$

tous les termes de cette suite étant une approximation supérieure du plus petit point fixe $\bar{x}$.

2.3.2 Résolution de systèmes partitionnés

Le plus souvent le domaine concret C se décompose naturellement en une partition finie $\bigcup_{k\in K} C^{(k)}$, ou plus généralement en un recouvrement.

Exemple. — Dans le cas typique de l'analyse de programmes impératifs, l'espace des états S est de la forme $K \times S'$, où K est l'ensemble des points de contrôle, et $C = \bigcup_{k\in K} C^{(k)}$ avec $C^{(k)} = \{k\} \times \mathcal{P}(S')$. □

Par généralisation, on appellera K l'ensemble des points de contrôle engendré par le recouvrement. Pour tout $x \in C$, soit $x^{(k)} = x \cap C^{(k)}$. Clairement, $\{x^{(k)} | k \in K\}$ est un recouvrement de x. Toute équation de point fixe $x = F(x)$ peut se réécrire

$$\bigwedge_{k\in K} x^{(k)} = F^{(k)}(x^{(1)}, x^{(2)}, \ldots, x^{(|K|)})$$

avec $F^{(k)}(x^{(1)}, x^{(2)}, \ldots, x^{(|K|)}) = F(x^{(1)} \cup x^{(2)} \cup \ldots \cup x^{(|K|)}) \cap C^{(k)}$.

Exemple. — Le système d'équation de point fixe correspondant au programme de la figure 2.1 est

$$\begin{array}{lcl} A & = & \{(2,0)\} \\ B & = & (A \cup E) \cap \{(x,y) \mid x+y \leqslant 20\} \\ C & = & post(x \leftarrow x+4)(B \cap \{(x,y) \mid x \geqslant 10\}) \\ D & = & post(x \leftarrow x+2; y \leftarrow y+1)(B \cap \{(x,y) \mid x < 10\}) \\ E & = & C \cup D \\ F & = & (A \cup E) \cap \{(x,y) \mid x+y > 20\} \end{array}$$

□

Cette décomposition se transpose dans le domaine abstrait, en prenant $y^{(k)} = \alpha(x^{(k)})$, ce qui donne le système d'équation abstrait :

$$\bigwedge_{k \in K} y^{(k)} = G^{(k)}(y^{(1)}, y^{(2)}, \ldots, y^{(|K|)})$$

Nous dirons, dans un tel système, que k dépend de k' si la valeur de $G^{(k)}(y^{(1)}, y^{(2)}, \ldots, y^{(|K|)})$ peut dépendre de la valeur de $y^{(k')}$. Soit $\mathcal{R}_G$ le graphe représentant cette relation de dépendance, aussi appelé *graphe de contrôle.*

L'avantage d'une décomposition est que les fonctions composantes $G^{(k)}$ sont plus simples que G et que le système peut être résolu plus efficacement, en utilisant des itérations chaotiques et en exploitant le fait que le graphe de dépendance n'est en général pas un graphe complet.

Définition 6 (Itération chaotique) *[Cou77] Soit le système d'équations de point fixe ci-dessus. Une* stratégie d'itération chaotique *est une suite $(K_i)_{i \geqslant 0}$ de parties de K telle que $\forall k \in K$, l'ensemble $\{i \mid k \in K_i\}$ est non borné (i.e., k apparaît infiniment souvent dans la suite $(K_i)_{i \geqslant 0}$).*

La résolution du système selon la stratégie d'itération $(K_i)_{i \leqslant 0}$ consiste à calculer en parallèle à chaque étape i les nouvelles valeurs des $y^{(k)}$ pour $k \in K_i$.

Dans la pratique on utilise toujours une stratégie linéaire, c'est-à-dire qu'on effectue qu'un calcul par étape. Une itération chaotique selon une stratégie bien choisie permet de propager rapidement et avec un coût réduit les valeurs abstraites sur tout le graphe de contrôle. Ainsi, si l'on a un chemin linéaire $\boxed{1} \rightarrow \boxed{2} \rightarrow \boxed{3}$, la stratégie $(123)^*$ permet de propager une croissance de la valeur du point 1 au point 3 en deux étapes et deux calculs, alors qu'une résolution parallèle nécessite deux étapes, six calculs, et plus de mémoire.

En outre, le partitionnement de l'équation de point fixe permet d'améliorer la précision de l'analyse :

1. lorsque l'opérateur d'élargissement ∇ est nécessaire pour garantir la convergence, on peut l'appliquer de manière plus sélective. Soit K_∇ un sous-ensemble de K tel que le graphe $K \setminus K_\nabla$ ne contienne pas de cycle. Alors la convergence de la séquence approchée ascendante reste garantie si l'on n'applique l'élargissement qu'aux points appartenant à K_∇ :

$$\begin{array}{ll} \forall k \in K, & y_0^{(k)} = \bot_A \\ \forall k \in K_\nabla, & y_{n+1}^{(k)} = y_n^{(k)} \nabla G^{(k)}(y^{(1)}, y^{(2)}, \ldots, y^{(|K|)}) \\ \forall k \in K \setminus K_\nabla, & y_{n+1}^{(k)} = G^{(k)}(y^{(1)}, y^{(2)}, \ldots, y^{(|K|)}) \end{array}$$

Moins d'information est ainsi perdue par l'emploi de l'opérateur d'élargissement.

2. lorsque sur le treillis l'opération d'union perd de l'information, i.e.,

   $$\gamma(x_1) \sqcup_C \gamma(x_2) \sqsubseteq_A \gamma(x_1 \sqcup_A x_2)$$

   la décomposition permet de manipuler des unions explicites de $|K|$ valeurs abstraites au lieu de leur borne supérieure.

La remarque 2 est importante : pour éviter la perte d'information due à l'union, il a été proposé dans la littérature de remplacer le treillis A par sa *complétion disjonctive* [CC92a], qui consiste à considérer des unions explicites non redondantes. Mais dans ce cas le test d'égalité peut être très coûteux (lorsque la décomposition n'est pas unique), le nombre d'éléments de l'union n'est pas borné, et il est éventuellement nécessaire d'utiliser des opérateurs d'élargissement sophistiqués pour limiter la taille de la représentation. Nous y reviendrons.

2.4 Exemples de domaines numériques abstraits

Nous présentons dans ce paragraphe des exemples importants d'interprétation abstraite de domaines numériques de la forme $\mathcal{N}^n$ avec $\mathcal{N} \in \{\mathbb{Z}, \mathbb{Q}, \mathbb{R}\}$, en allant du plus simple au plus puissant. L'objet de ces inteprétations abstraites est de calculer, en chaque point de contrôle d'un programme, une approximation supérieure de l'ensemble des états accessibles des variables numériques du programme. Si le programme étudié comporte n variables numériques l'espace des états en chaque point de contrôle est $\mathcal{N}^n$. Nous illustrerons l'utilisation de ces domaines pour l'analyse des valeurs prises par les variables du programme de la figure 2.1, dont le domaine concret partitionné est $C = K \times \mathbb{Z}^2$ avec $K = \{A, B, C, D, E\}$.

2.4.1 Le treillis des intervalles

Définition

Supposons d'abord que $S = \mathcal{N}$ et $C = \mathcal{P}(\mathcal{N})$. Une approximation simple consiste à abstraire toute partie de $\mathcal{N}$ par le plus petit intervalle qui le contient [CC76], et donc à choisir comme domaine abstrait A le treillis $\mathcal{I}(\mathcal{N})$ des intervalles sur $\mathcal{N}$:

$$\mathcal{I} = \{\bot\} \cup \{\, [l, u] \mid l \in \mathcal{N} \cup \{-\infty\} \ \wedge\ u \in \mathcal{N} \cup \{+\infty\} \ \wedge\ l \leqslant u\}$$

ordonné par la relation

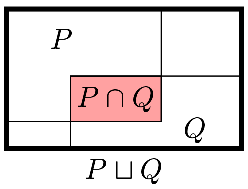
$$\begin{cases} \forall X\ \bot \sqsubseteq X \\ [l_1, u_1] \sqsubseteq [l_2, u_2] \iff l_1 \leqslant l_2 \leqslant u_2 \leqslant u_1 \end{cases}$$

Par exemple si $\mathcal{N} = \mathbb{Z}$, $\{1, 3, 6\}$ est abstrait par $[1, 6]$, qui possède la signification concrète $\{1, 2, 3, 4, 5, 6\}$.

La généralisation à un espace multidimensionnel se fait en abstrayant $\mathcal{N}^n$ par $\mathcal{I}(\mathcal{N})^n$: une partie de l'espace est abstraite par le plus petit hyperrectangle le contenant (éventuellement non borné sur certaines dimensions).

Opérations

Les opérations de bornes inférieures et supérieures découlent du choix du domaine abstrait et sont ici particulièrement simples, cf. figure ci-contre. Les opérations arithmétiques abstraites sont les opérations classiques de l'arithmétique d'intervalle.



```
program PL;
  var I,J : integer;
begin
  I := 2; J := 0;
  (A) [2,2] × [0,0]
  while I+J <= 20 do begin
    (B) [2,20] × [0,4]
    if I >= 10 then begin
      I := I+4;
      (C) [14,24] × [0,4]
    end else begin
      I := I+2; J := J+1;
      (D) [4,10] × [1,4]
    end ;
    (E) [4,24] × [0,4]
  end ;
  (F) [17,24] × [0,4]
end.
```

(a) Analyse, convergeant sans élargissement

```
program PL;
  var I,J : integer;
begin
  I := 2; J := 0;
  (A) [2,2] × [0,0]
  while I+J <= 20 do begin
    (B) [2,20] × [0,18]
    if I >= 10 then begin
      I := I+4;
      (C) [14,24] × [0,18]
    end else begin
      I := I+2; J := J+1;
      (D) [4,22] × [1,19]
    end ;
    (E) [4,+∞] × [0,+∞]
  end ;
  (F) [4,+∞] × [0,+∞]
end.
```

(b) Analyse, avec emploi de l'élargissement au point (E)

FIG. 2.2 – Analyse d'intervalle sur le programme PASCAL

$\mathcal{I}(\mathcal{N})$ étant un treillis de hauteur infinie, il est nécessaire de définir un opérateur d'élargissement. L'opérateur standard sur $\mathcal{I}$ proposé par [CC76] est le suivant :

$$\begin{aligned}
\bot \nabla X &= X \\
X \nabla \bot &= X \\
[l_1, u_1] \nabla [l_2, u_2] &= [\,\text{if}\, l_2 < l_1 \,\text{then}\, -\infty \,\text{else}\, l_1,\ \text{if}\, u_2 > u_1 \,\text{then}\, +\infty \,\text{else}\, u_1]
\end{aligned}$$

La complexité en temps comme en espace des opérations abstraites usuelles : bornes inférieure et supérieure, post- et pré-condition, croît linéairement avec le nombre de variables numériques.

EXEMPLE. — La figure 2.2 illustre les résultats de l'analyse d'intervalle sur notre programme PASCAL, et l'emploi de l'opérateur d'élargissement. La figure 2.2(a) montre les résultats d'une analyse d'accessibilité n'employant pas d'élargissement mais qui se trouve être convergeante sur ce programme, tandis que la figure 2.2(b) montre les résultats d'une analyse d'accessibilité appliquant un élargissement au point (E). On remarque l'importance de n'élargir qu'en un nombre minimum de points : si on avait élargi partout, on n'aurait pas obtenu de borne supérieure sauf en (B). Il est également clair que dans le cas où l'on a élargi, une étape de séquence décroissante permet d'obtenir une borne supérieure pour I et J aux points (E) et (F). □

En raison de sa faible complexité, le treillis des intervalles est souvent utilisé dans les analyses interprocédurales de programme impératifs ou fonctionnels, où le nombre de variables à traiter est élevé [Bou92b].

2.4.2 Le treillis des égalités linéaires

Le treillis précédent a pour avantage d'être algorithmiquement très économique, en temps comme en espace. Toutefois il n'est pas non plus très précis ; en particulier il ne permet pas de représenter un ensemble dans lequel les valeurs de variables distinctes sont liées, comme c'est le cas pour la droite du plan définie par $x + y = 0$; un treillis possédant cette propriété est dit *relationnel*.

Définition

Le premier treillis numérique relationnel utilisé en analyse de programme est le treillis des égalités linéaires [Kar76] : les éléments de ce treillis sont les sous-espaces affines de $\mathcal{N}^n$, représentables par des systèmes d'équations affines.

Opérations

L'opération de borne inférieure sur ce treillis est simple à mettre en oeuvre : il suffit de fusionner les systèmes d'équations et de normaliser le système résultant. En revanche l'opérateur de borne supérieure est plus compliqué et sa description fait l'objet de toute une partie de l'article [Kar76].

Les opérations arithmétiques linéaires se calculent aisément sur un système d'équations linéaires ; en revanche une opération non linéaire fait perdre toute l'information.

Il n'y a pas besoin d'opérateur d'élargissement, la profondeur de ce treillis étant finie et même petite, puisqu'elle est de $n + 1$: en effet dans toute suite strictement croissante, la dimension des éléments est également strictement croissante, et est bornée par la dimension de l'espace.

La complexité des opérations usuelles est $O(n^3)$ en temps (la normalisation de la représentation comporte une élimination de Gauss) et $O(n^2)$ en espace.

Exemple. — La figure 2.3 illustre les résultats de l'analyse d'intervalle sur notre programme Pascal. On constate qu'on obtient aucune informations, sauf aux points (B) et (D). En effet des opérations simples, comme l'incrémentation répétée d'une variable, font perdre toute l'information : c'est ce qui se passe au point (C). □

En fait ce treillis est intéressant lorsqu'il est combiné avec un autre treillis, incomparable du point de vue de l'inclusion, comme celui des intervalles. L'article [CC79] explique comment construire le *produit réduit* de deux treillis, qui permet de combiner ainsi deux classes de propriétés.

2.4.3 Le treillis des "zones"

Définition et représentation

Une *zone* [HNSY92] est l'ensemble vide ou une région de l'espace définie par la conjonction de *contraintes de différence*, c'est-à-dire de contraintes de la forme $x_i \leqslant d$ ou $x_i - x_j \leqslant d$ avec $d \in \mathbb{N}$. L'intersection de zones étant une zone, l'ensemble des zones forme une famille de

```
program PL;
  var I,J : integer;
begin
  I := 2; J := 0;
  (A) {I=2, J=0}
  while I+J <= 20 do begin
    (B) ⊤
    if I >= 10 then begin
      I := I+4;
      (C) ⊤
    end else begin
      I := I+2; J := J+1;
      (D) {I=2J+2}
    end ;
    (E) ⊤
  end ;
  (F) ⊤
end.
```

FIG. 2.3 – Analyse d'égalités linéaires sur le programme PASCAL

```
program PL;
  var I,J : integer;
begin
  I := 2; J := 0;
  (A) {I=2, J=0}
  while I+J <= 20 do begin
    (B) {2⩽I⩽20, 0⩽J⩽4, I−J⩾2}
    if I >= 10 then begin
      I := I+4;
      (C) {14⩽I⩽24, 0⩽J⩽4}
    end else begin
      I := I+2; J := J+1;
      (D) {4⩽I⩽10, 1⩽J⩽4, I−J⩾3}
    end ;
    (E) {4⩽I⩽24, 0⩽J⩽4, I−J⩾3}
  end ;
  (F) {17⩽I⩽24, 0⩽J⩽4}
end.
```

FIG. 2.4 – Analyse de zone sur le programme PASCAL, sans élargissement

Moore et définit donc un treillis abstrait, qui est complet. Notons que ce treillis contient le treillis des intervalles comme sous-treillis.

Les zones sont principalement utilisées dans la vérification des automates temporisés [AD90], qui sont des automates finis étendus avec des horloges évoluant de manière synchrone et continue, et pouvant être remises à zéro lorsqu'une transition est tirée. Les zones forment un ensemble fermé par les opérateurs de post- et pré- condition induits par la sémantique de ces automates, et les unions finies de zones permettent de représenter de manière exacte les valeurs accessibles des horloges pour chaque état de contrôle.

La représentation classique des zones se fait par des DBMs (Difference Bound Matrices), structure de donnée introduite dans [Dil89], qui sont des graphes dont les sommets sont étiquetés par les variables et les arêtes par des constantes : $x_i \xrightarrow{d} x_j$ représente $x_i - x_j \leqslant d$.

Opérations

La borne inférieure de deux zones consiste à fusionner les contraintes des deux zones, ce qui revient à ajouter les arêtes d'un DBM à l'autre DBM, et de normaliser le graphe résultant. Le calcul de la borne supérieure se fait en considérant pour chaque couple $x_i - x_j$ sa borne inférieure et supérieure dans les deux zones dont on calcule la borne supérieure.

Il est aisé de calculer postcondition et précondition de zones par des opérations du type $x_i \leftarrow c$ ou $x_i \leftarrow x_j + c$, éventuellement gardées par des contraintes de différence. En revanche les opérations linéaires générales ou même non linéaires font nécessairement perdre de l'information.

Un opérateur d'élargissement standard serait par exemple d'utiliser l'élargissement sur les intervalles pour chaque variable x_i et différence $x_i - x_j$. Un tel opérateur n'a encore jamais été utilisé à notre connaissance, n'étant pas nécessaire pour l'analyse d'automates temporisés.

Si n est le nombre de variables, la complexité des opérations est en $O(n^3)$, la normalisation de la représentation consistant à calculer les plus courts chemins du graphe (algorithmes de Floyd-Warshall et Bellman-Ford [CLR94]).

Exemple. — La figure 2.4 illustre l'analyse de zone sur notre programme, sans emploi d'élargissement. Les résultats obtenus sont plus précis que ceux de l'analyse d'intervalle aux points (B), (D) et (E). Le résutat obtenu au point (D) illustre le fait que le treillis des zones ne contient pas le treillis des égalités linéaires. □

2.4.4 Le treillis des polyèdres convexes

[CH78] a proposé le treillis $\mathcal{P}^{oly}$ des polyèdres convexes pour représenter les espaces numériques, ce qui permet d'inférer des relations d'*inégalité linéaire* entre variables. Ce treillis n'est pas complet : en effet on peut construire une suite de polyèdres convergeant vers un disque. Nous ne sommes donc pas dans le cadre standard de l'interprétation abstraite. On contourne cette difficulté en se plaçant dans le treillis complet des convexes, dans lequel toute partie de $\mathcal{N}^n$ est abstraite par le plus petit convexe de $\mathcal{N}^n$ qui la contient. Le fait ensuite de ne manipuler que des polyèdres et d'utiliser un opérateur d'élargissement nous garantit que nous demeurons dans le sous-treillis des polyèdres convexes.

Représentation des polyèdres

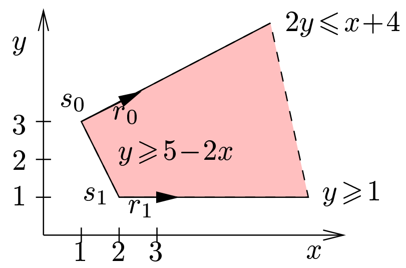


Fig. 2.5 – Les deux représentations d'un polyèdre

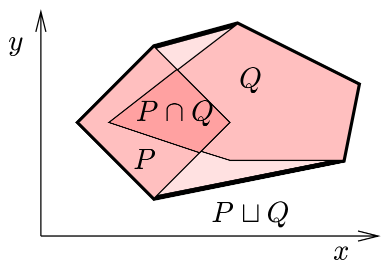


Fig. 2.6 – Intersection et enveloppe convexe de polyèdres

Un polyèdre (figure 2.5) peut alternativement être représenté par une conjonction d'inégalités linéaires :

$$P = \{x \mid \bigwedge_{1 \leqslant j \leqslant m} a_j x \geqslant b_j\} = \{x \mid Ax \geqslant B\}$$

ou par ses éléments générateurs :

$$P = \left\{ \sum_{s_i \in S} \lambda_i s_i + \sum_{r_j \in R} \mu_j r_j \mid \lambda_i \geqslant 0, \mu_j \geqslant 0, \sum_i \lambda_i = 1 \right\}$$

où S est un ensemble de *sommets* et R un ensemble de *rayons*. (A, B) et (S, R) sont respectivement un système de contraintes et un système générateur de P.

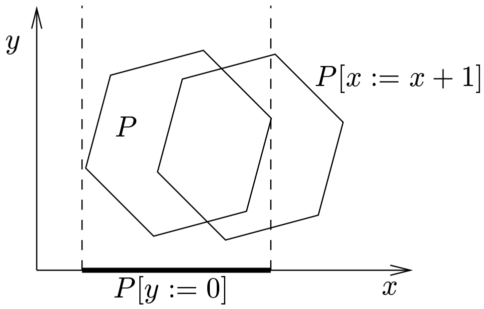


FIG. 2.7 – Transformation linéaire

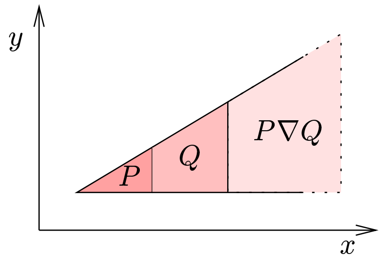


FIG. 2.8 – Élargissement de polyèdre

Opérations sur les polyèdres

Intersection et enveloppe convexe. — L'intersection de deux polyèdres P et Q est le polyèdre dont le système de contraintes est la conjonction de ceux de P et Q, fig. 2.6. Le treillis des polyèdres convexes n'étant pas fermé par union, on utilise à la place l'*enveloppe convexe* : l'enveloppe convexe de deux polyèdres P et Q est le plus petit polyèdre convexe qui les contient. Son système générateur est l'union de ceux de P et Q, cf. fig. 2.6. Cette opération est l'abstraction de l'union sur le treillis des polyèdres au sens de la définition 4.

Transformation linéaire. — Une *transformation linéaire* est un couple (M, T) avec $M \in \mathcal{N}^{n\times n}$ et $T \in \mathcal{N}^n$, définissant la fonction

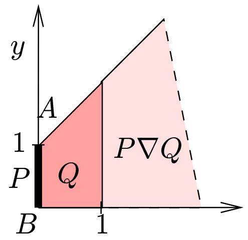
$$\begin{array}{lll} \mathcal{N}^n & \to & \mathcal{N}^n \\ x & \mapsto & Mx + T \end{array}$$

Si (S, R) est un système générateur de P, alors $(S' = \{Ms + T \mid s \in S\}, R' = \{Ar \mid r \in R\})$ est un système générateur de la postcondition de P par la transformation (M, T), figure 2.7. Si (A, B) est un système de contraintes de P, alors $(A' = AM, B' = AT + B)$ est un système de contraintes de la précondition de P par (M, T). Dans le cas où la transformation est inversible, on peut l'appliquer aux deux représentations.

Test d'inclusion et d'égalité. — Un polyèdre P de système générateur (S, R) est inclus dans un polyèdre Q défini par un système de contraintes (A, B) si et seulement si tous les générateurs de P satisfont les contraintes de Q : $\forall s \in S, As \geqslant B$ et $\forall r \in R, Ar \geqslant 0$. L'égalité de deux polyèdres est décidée par double inclusion.

Élargissement. — Tandis que les opérations de base sont données par le choix du domaine abstrait, l'élargissement est toujours fondé sur des heuristiques. L'opérateur d'élargissement standard sur le treillis des polyèdres fut proposé dans [CH78]. Soit P et Q deux polyèdres ; $P \nabla Q$ est *grosso-modo* obtenu en ôtant du système de P toutes les inégalités qui ne sont pas vérifiées par Q, figure 2.8. L'intuition est la suivante : lorsqu'une inégalité est translatée ou subit une rotation, elle peut le faire une infinité de fois et est donc enlevée. L'opérateur satisfait la propriété de l'élargissement : le résultat contient les deux opérandes et comme le système d'inégalités de $P \nabla Q$ est un sous-ensemble de celui de P, l'élargissement ne peut pas être infiniment répété sans convergence.

L'opérateur effectif est un peu plus compliqué ; en effet, appliquée tel quel, l'idée intuitive décrite ci-dessus donne des résultats hasardeux si les polyèdres contiennent des égalités ou



	Contraintes de P			Contraintes de Q			
	$x=0$	$0\leqslant y$	$y\leqslant 1$	$0\leqslant x$	$x\leqslant 1$	$0\leqslant y$	$y\leqslant x+1$
$A=(0,1)$	0	0	1	0	1	0	1
$B=(0,0)$	0	1	0	0	1	1	0

FIG. 2.9 – Élargissement effectif : polyèdres et tableau de saturation

des droites, car le système des contraintes minimal n'est alors plus canonique[1]. L'opérateur implémenté consiste donc à définir $P\nabla Q$ par l'ensemble des contraintes de Q *mutuellement redondantes* avec des contraintes de P. Étant donné un polyèdre, deux contraintes sont *mutuellement redondantes* si elles saturent les mêmes générateurs de ce polyèdre, et une contrainte *sature* un générateur si celui-ci est inclus dans l'hyperplan défini par la contrainte.

EXEMPLE. — Soient les deux polyèdres $P=\{x=0,\ 0\leqslant y\leqslant 1\}$ et $Q=\{0\leqslant x\leqslant 1, 0\leqslant y\leqslant x+1\}$ de la figure ci-contre. Si on applique "bêtement" l'élargissement, on obtient le quart de plan $\{0\leqslant x, 0\leqslant y\}$. En réécrivant P ainsi : $\{x=0, 0\leqslant y\leqslant x+1\}$, on obtiendrait le résultat souhaité de la figure : $P\nabla Q=\{0\leqslant x, 0\leqslant y\leqslant x+1\}$.

La notion de contraintes mutuellement redondante prend en compte ce phénomène, comme le montre le tableau de saturation de la figure 2.9. Les colonnes correspondent aux contraintes de P et Q, tandis que les lignes aux sommets de P. Dans ce tableau, un 0 indique que le sommet sature la contrainte. $P\nabla Q$ est défini par les contraintes de Q mutuellement redondantes avec des contraintes de P, à savoir $0\leqslant x$, $0\leqslant y$, $y\leqslant x+1$, mutuellement redondantes avec respectivement $x=0$, $0\leqslant y$ et $y\leqslant 1$ □

Complexité des opérations

Le treillis des polyèdres est algorithmiquement beaucoup plus coûteux que les treillis précédents. Tout d'abord, la taille de la représentation d'un polyèdre convexe n'est pas bornée, puisque le nombre de contraintes ou de générateurs le définissant ne l'est pas. Ensuite l'algorithme fondamental sur les polyèdres est la conversion avec minimisation d'un système de contraintes en un système de générateurs, et la conversion inverse ; cet algorithme permet d'implémenter l'intersection et l'enveloppe convexe. Or la taille du résultat d'une telle conversion est dans le pire des cas en $0(2^d)$ la taille de la représentation initiale. Cela signifie que lors d'une intersection ou d'une enveloppe convexe, la complexité en espace et en temps croît exponentiellement avec le nombre de variables.

EXEMPLE. — Appliquons l'analyse de relations linéaires à notre programme PASCAL, avec élargissement au point (E), fig 2.10. On constate l'obtention de résultats beaucoup plus précis que l'analyse d'intervalle, fig 2.2. Ici aussi, une étape de séquence décroissante améliorerait les résultats pour (E) et (F) : on obtiendrait

(E) $I\geqslant 2J+2,\ I\geqslant 4,\ J\geqslant 0,\ I+J\leqslant 24$

(F) $I\geqslant 2J+2,\ J\geqslant 0,\ 21\leqslant I+J\leqslant 24$

□

[1]on peut cependant le rendre canonique par élimination de Gauss et et en ordonnant les variables.

```
program PL;
  var I,J : integer;
begin
  I := 2; J := 0;
  (A) I = 2, J = 0
  while I+J <= 20 do begin
    (B) I ⩾ 2J+2, J ⩾ 0, I+J ⩽ 20
    if I >= 10 then begin
      I := I+4;
      (C) I ⩾ 2J+6, J ⩾ 0, I ⩾ 14, I+J ⩽ 24
    end else begin
      I := I+2; J := J+1;
      (D) I ⩾ 2J+2, J ⩾ 1, I+J ⩽ 23
    end ;
    (E) I ⩾ 2J+2, J ⩾ 0
  end ;
  (F) I ⩾ 2J+2, J ⩾ 0, I+J ⩾ 21
end.
```

FIG. 2.10 – Analyse de relations linéaires sur le programme PASCAL, avec élargissement au point (E).

Chapitre 3

Programmation synchrone et vérification

Nous présentons la programmation synchrone et un langage synchrone particulier, le langage Lustre*, ainsi que leur sémantique opérationnelle de bas niveau, utilisée par les outils de vérification existant dont nous présentons les principes*

.

Nous introduisons au §3.1 la philosophie et les caractéristiques générales des langages synchrones. Le §3.2 décrit plus précisément le langage Lustre, dans lequel la plupart des exemples de programme seront écrits. Enfin le §3.3 décrit en détail les techniques qui ont été développées pour la vérification des programmes synchrones et qui exploitent leur sémantique particulière.

3.1 Les langages synchrones

Les langages synchrones [IEE91, Hal93b], sont destinés à la programmation de systèmes interagissant avec leur environnement, réactifs et déterministes ; ces systèmes attendent des entrées, éventuellement simultanées, réagissent en calculant et en émettant des sorties, puis attendent les entrées suivantes. Leur domaine d'application induit une structuration des programmes en processus parallèles, communiquant entre eux et avec l'environnement. Il est important de remarquer qu'il s'agit ici d'un parallélisme *logique*, qui n'a rien à voir avec un parallélisme *physique* dans lequel un programme est distribué sur plusieurs unités de calcul. L'hypothèse *synchrone* stipule que les processus réagissent instantanément à leurs entrées, et que les communications entre processus se font par diffusion instantanée de signaux. La propriété de *réactivité* signifie que les processus ont un comportement défini pour toute entrée ; il n'y pas de notion de blocage. Enfin la propriété de *déterminisme* signifie que des séquences d'entrées identiques conduisent toujours à la même séquence de sortie. L'instantanéité de la réaction aux entrées est bien sûr une vue abstraite ; son intérêt est de permettre une notion simple de composition parallèle et de laisser les problèmes de génération de code aux compilateurs. Cette vue abstraite est fondée si le temps maximal de réaction aux entrées du programme compilé est inférieur à leur fréquence.

La notion de temps physique n'existe pas en programmation synchrone, mais est remplacée par un temps logique, défini par la simultanéité et la succession des événements.

Ce type de programmes s'oppose, selon la terminologie de [HP85], aux systèmes *transformationnels* dont le rôle est de terminer avec un résultat calculé à partir de données initiales (les compilateurs sont des exemples de tels programmes), et aux systèmes *interactifs* classiques qui communiquent continuellement avec leur environnement, mais à leur vitesse propre ; en particulier ils peuvent faire attendre leur environnement, ce qui est typiquement le cas des systèmes d'exploitation classiques.

Les langages synchrones parmi les plus représentatifs sont les langages impératifs ARGOS [Mar90, Mar92], ESTEREL [BG92] et SYNCCHARTS [And96], et les langages fonctionnels flots de données LUSTRE [HCRP91] et SIGNAL [LGLL91]. ARGOS est une version purement synchrone du langage graphique des STATECHARTS et est fondé sur des automates communiquants et hiérarchiques. ESTEREL est un langage textuel offrant la composition parallèle de processus, qui communiquent par des *signaux diffusés*, ainsi que la notion d'interruption. SYNCCHARTS est une version graphique d'ESTEREL et fait en quelque sorte la synthèse des deux langages prédédents. Enfin LUSTRE et SIGNAL ont pour objets de base des flots, éventuellement cadencé par des horloges qui sont elles-mêmes des flots booléens, et des fonctions sur ces flots. Ils se distinguent l'un de l'autre principalement par le traitement des horloges.

Nous ne présenterons que le langage LUSTRE, les programmes vérifiés étant pour la plupart programmés en LUSTRE.

3.2 Le langage LUSTRE

Un programme LUSTRE définit ses variables de sortie comme des fonctions de ses variables d'entrées. Une variable ou une expression `E` dénote une fonction d'un temps discret, donnant sa valeur $\mathtt{E}_n$ pour tout "instant" $n \geqslant 0$. Les variables sont définies par des équations : une équation "`X=E`" spécifie que la variable `X` est toujours égale à l'expression `E`.

Les expressions sont construites à partir des identificateurs de variables, de constantes (considérée comme des fonctions constantes), des opérateurs arithmétiques, logiques et conditionnels usuels (étendus point-à-point aux fonctions), et de seulement deux opérateurs spécifiques :

- l'opérateur de retard `pre`, qui retarde un flot d'un instant et est défini par

$$\begin{array}{lcll} (\mathtt{pre(E)})_0 & = & \mathit{nil} & \text{où } \mathit{nil} \text{ est la valeur indéfinie} \\ (\mathtt{pre(E)})_n & = & \mathtt{E}_{n-1} & \text{pour } n > 0 \end{array}$$

  Dans le cas d'une variable booléenne, cet opérateur est la version logicielle des registres dans les circuits digitaux synchrones.

- l'opérateur "suivi-de", noté `->` et défini par

$$\begin{array}{lcll} (\mathtt{E\text{-}\!\!>F})_0 & = & \mathtt{E}_0 & \\ (\mathtt{E\text{-}\!\!>F})_n & = & \mathtt{F}_n & \text{pour } n > 0 \end{array}$$

EXEMPLE. — Si `x` et `y` sont des variables, l'équation "`z = 0->(pre(x)+y)`" signifie que la valeur initiale $\mathtt{z}_0$ de `z` est 0, et que pour tout instant non initial $n > 0$, $\mathtt{z}_n = \mathtt{x}_{n-1} + \mathtt{y}_n$. □

Un programme LUSTRE est structuré en *nœuds* : un *nœud* est un sous-programme définissant ses paramètres de sortie en fonction de ses paramètres d'entrée, et faisant éventuellement intervenir des variables locales. Cette définition est donnée par un ensemble non ordonné d'équations, une par paramètre de sortie ou variable locale. Une fois déclaré, un nœud peut-être instancié dans une expression quelconque tout comme un opérateur de base. Il est bien sûr nécessaire de spécifier au compilateur le nœud principal, qui définit les entrées/sorties du programme.

EXEMPLE. — La figure 3.1 montre un nœud très simple décrivant un compteur : il reçoit deux entrées entières, `init` et `incr`, et une entrée booléenne `reset`. Il retourne une sortie entière `count`, qui se comporte ainsi : à l'instant initial ou quand `reset` est vrai, la valeur de la sortie est égale à la valeur courante de `init`, et sinon la valeur de `count` est égale à sa valeur à l'instant précédant incrémentée de `incr`. On peut ensuite utiliser ce nœud ailleurs, comme dans l'équation `mod5 = Compteur(0, 1, pre(mod5)=0);` qui instancie `Compteur` avec `0` et `1` comme constante initiale et d'incrémentation, et le réinitialise quand la valeur précédente de la sortie est 4. La variable `mod5` est ainsi la séquence cyclique des entiers modulo 5. □

```
node Compteur (init, incr: int;
               tick, reset: bool)
     returns (count: int);
let
  count = init -> if reset then
                    init
                  else
                    pre(count)+incr;
tel
```

FIG. 3.1 – Un exemple de nœud LUSTRE

À travers cette notion de nœud, LUSTRE autorise la description hiérarchique et la réutilisation de composants. Il est également possible de définir des types structurés à partir des types de base.

D'un point de vue temporel, les applications industrielles montre que différentes chaînes de calculs, cadencées à des fréquences différentes, peuvent apparaître dans le même système. LUSTRE offre la notion d'*horloge*, qui permet d'activer un nœud seulement lorsque son horloge émet un signal, plutôt que pour tout instant. Ce concept n'intervient pas en vérification et ne sera donc plus évoqué dans la suite.

3.3 Vérification des programmes synchrones

La programmation synchrone étant principalement utilisée dans le domaine des systèmes critiques embarqués, la vérification formelle des programmes synchrones présente un grand intérêt. Heureusement, elle peut tirer parti des particularités du domaine d'application et du modèle synchrones.

L'expérience montre que les propriétés critiques qui doivent être vérifiées sont généralement des propriétés de *sûreté*. Ceci permet l'utilisation d'abstractions, comme nous l'avons vu au chapitre précédent. Par ailleurs, ces propriétés sont souvent *simples*, au sens où elles font plutôt intervenir des relations logiques entre événements plutôt que des propriétés numériques fines. Ceci a pour conséquence que les propriétés peuvent souvent être vérifiées sur une abstraction finie du programme, qui est généralement l'automate de contrôle engendré par son abstraction booléenne, §3.3.5. Pour les cas où cette abstraction est trop grossière, il est nécessaire de prendre en compte l'aspect numérique, ce qui est l'objet de [Hal93a, HPR97] qui utilisent à cette fin l'analyse de relations linéaires sur l'automate de contrôle. Ce dernier travail, exposé au §3.3.6, est le point de départ de cette thèse.

L'hypothèse synchrone présente pour ces techniques utilisant l'automate de contrôle des avantages non négligeables :

– Le modèle synchrone permet de spécifier les propriétés de sûreté, ainsi que les hypothèses sur l'environnement sous lesquelles les propriétés doivent être vraies, au moyen d'*observateurs synchrones*, §3.3.2, qui sont des programmes spéciaux écrits dans le même langage que le programme vérifié.

- L'automate de contrôle du programme, obtenu par produit parallèle synchrone de ses composants, est généralement beaucoup plus petit que les modèles obtenus par produit asynchrone : il n'y a pas entrelacement des événements et par conséquent inutilité de techniques sophistiquées d'ordre partiel.
- La relation de transition est une *fonction vectorielle*, qui permet lors de l'analyse de l'abstraction booléenne l'utilisation de techniques fondées sur des BDDs particulièrement efficaces [CBM89b].

Dans le cas des langages LUSTRE et SIGNAL, le modèle flot de donnée facilite en outre l'analyse du *cône d'influence* de la propriété, qui est l'ensemble des variables dont dépend la propriété. Cette analyse permet de diminuer le nombre de variables prises en compte dans les algorithmes de vérification.

3.3.1 Programme illustratif

Nous allons illustrer ce qui suit avec un exemple extrait d'un système de régulation de vitesse d'un métro. Le programme détermine si un train est en avance ou en retard, de la manière suivante : le train détecte et compte

- des balises, disposées le long de la voie ;
- des signaux, diffusés par radio par une horloge centrale.

Notons Δ la différence entre le nombre de balises détectées et le nombre de signaux reçus. Idéalement un train devrait rencontrer exactement une balise pour chaque réception du signal ; il est donc "à l'heure" lorsque $0 \leqslant \Delta \leqslant 1$. Cependant, pour éviter des changements d'allure trop fréquents et brutaux, une hystérésis est appliquée :

- Le train n'est considéré "en avance" que lorsque Δ devient supérieur à 10, et le reste jusqu'à ce que Δ redevienne plus petit ou égal à 1 ;
- Le train n'est considéré "en retard" que lorsque Δ devient inférieur à -10, et le reste jusqu'à ce que Δ redevienne positif ou nul.

Ce petit système est décrit avec le programme LUSTRE suivant — en supposant pour simplifier que les entrées `B` (pour "balise") et `S` (pour "signal") sont exclusives :

```
node Controleur(B,S: bool) returns (Avance, Retard: bool)
var D: int;
let
  assert(not (B and S));
  D = 0 ->
          if B then pre(D) + 1
          else if S then pre(D) - 1
          else pre(D);
  Avance = false ->
           if pre(Avance) then D > 1
           else D >= 10;
  Retard = false ->
           if pre(Retard) then D < 0
           else D <= -10;
tel
```

3.3.2 Observateurs synchrones et schéma de vérification

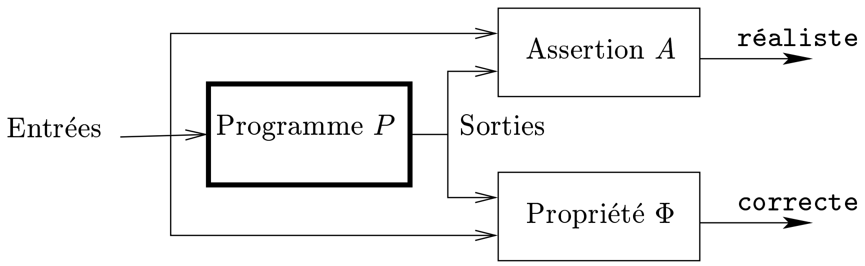


FIG. 3.2 – Schéma de vérification d'un programme synchrone

Un grand nombre d'outils de vérifications et de tests, et en particulier ceux construits autour de LUSTRE, utilisent la notion d'*observateurs synchrones* [HLR93], qui permet à la fois de spécifier la propriété à vérifier et les hypothèses sur l'environnement sous lesquelles la propriété est censée être satisfaite. L'observateur d'une propriété de sûreté est un programme prenant en entrée les entrées/sorties du programme vérifié, et qui décide à chaque instant si la propriété est violée ou non en positionnant une alarme. En composant en parallèle le programme P, un observateur Φ de la propriété à prouver, et un observateur A des hypothèses faites sur l'environnement, il suffit juste de vérifier que soit le signal d'alarme de Φ n'est jamais émis (propriété satisfaite), soit le signal d'alarme de A est émis (hypothèses violées), ce qui peut se faire par un simple parcours des états accessibles du programme composé. La propriété de sûreté spécifiée par Φ sous l'hypothèse spécifiée par A est donc transformée en une propriété d'invariance sur la composition du programme et de ces deux observateurs.

EXEMPLE. — Dans le programme précédent, l'instruction `assert(not (B and S))` correspond un observateur d'assertion qui émet une alarme si `B` et `S` sont présents au même instant.

Supposons que l'on veuille montrer que le train ne peut pas être à la fois en avance et en retard ; l'observateur correspondant est

```
node Prop1(Avance, Retard: bool) returns (OK: bool)
let
  OK = not (Avance and Retard);
tel
```

Si maintenant on veut montrer qu'on ne peut pas passer en un seul cycle de l'état "en retard" à l'état "en avance", propriété due à l'hystérésis, on peut écrire:

```
node Prop2(Avance, Retard: bool) returns (OK: bool)
let
  OK = true -> not (Avance and pre(Retard));
tel
```

□

Il peut paraître surprenant au premier abord que les hypothèses sur l'environnement puissent faire intervenir des sorties du programme. En fait l'environnement n'est pas forcément passif. Dans le cas d'un contrôleur d'ascenseur par exemple, l'environnement est l'ascenseur lui-même, dont on suppose qu'il va fermer ses portes lorsqu'il en reçoit l'ordre (signal de sortie du contrôleur). Une hypothèse raisonnable est qu'il cesse alors d'indiquer au contrôleur que la porte est ouverte.

Outre le fait de disposer d'un langage unique pour les programmes et les propriétés, le fait que les observateurs soient des programmes permet de les exécuter et de les tester, ce qui n'est pas le cas de propriétés spécifiées dans des formalismes tels que les logiques temporelles. Par ailleurs [HFB93] montre que, dans le cadre fini, un observateur LUSTRE a le pouvoir d'expression des propriétés de sûreté régulières.

Un autre intérêt de cette technique est que lorsque l'on considère la composition du programme avec l'observateur de l'environnmement et celui de la propriété, seul le positionnement des signaux d'alarmes des observateurs nous intéresse. Dans le cas des langages flots de données LUSTRE et SIGNAL, il est alors particulièrement facile de mener une analyse de *cône d'influence* qui détermine quelles sont les variables ayant une influence sur ces alarmes et permet d'éliminer les autres ; on peut ainsi simplifier parfois considérablement le problème de vérification par une analyse statique très économique.

3.3.3 Représentation symbolique d'un programme sous vérification

Les outils de vérification se branchent sur les premières passes des compilateurs, après qu'ils aient mis "à plat" le programme et résolu les aspects de synchronisation interne. Tout programme synchrone composé avec un observateur de propriété et un observateur d'assertion peut ainsi se compiler dans le formalisme suivant, qui est, en gros, celui des systèmes dynamiques discrets, augmenté des notions d'assertion et d'invariant.

Syntaxe

Un programme sous vérification est défini par :

- la déclaration de variables booléennes d'état $(b_i)_{i=1\ldots m}$ et d'entrées $(\beta_j)_{j=1\ldots n}$, et la déclaration de variables numériques d'état $(x_k)_{k=1\ldots p}$ et d'entrée $(\xi_\ell)_{\ell=1\ldots q}$.
- un ensemble d'équations $b'_i = \phi_i(\vec{b},\vec{\beta},\vec{x},\vec{\xi}), i = 1\ldots m$ et $x'_k = \psi_k(\vec{b},\vec{\beta},\vec{x},\vec{\xi}), k = 1\ldots p$ qui donnent pour chaque variable d'état sa valeur à l'instant suivant en fonction de la valeur courante des variables d'état et d'entrée. Dans ces équations les expressions ϕ_i et ψ_k sont des expressions bien-formées, combinant expressions booléennes et numériques. Les atomes des expressions booléennes sont soit des variables booléennes, soit des contraintes numériques.
- un état ou un ensemble d'état initiaux, donné par une expression booléenne $init(\vec{b},\vec{x})$.
- une *assertion* modélisant l'environnement, qui est une expression booléenne de la forme $assert(\vec{b},\vec{\beta},\vec{x},\vec{\xi})$.
- et enfin une expression booléenne $inv(\vec{b},\vec{x})$ définissant la propriété à vérifier.

L'assertion et l'invariant sont en fait les deux sorties du programme sous vérification.

EXEMPLE. — Après la partie frontale du compilateur LUSTRE V4, qui va composer l'assertion et la propriété spécifiée par l'observateur `Prop1`, on obtient la représentation symbolique suivante :

```
state Init,Avance,Retard : bool;
      D: int;
input B,S : bool;
transition
  Init'   = false;
  D'      = if B then D+1 else if S then D-1 else D;
```

```
    Avance' = if init then false
              else if Avance then
                     D + (if B then 1 else if S then -1 else 0) <= 1
                   else
                     D + (if B then 1 else if S then -1 else 0) >= 10;
    Retard' =  if init then false
              else if Retard then
                     D + (if B then 1 else if S then -1 else 0) >= 0
                   else
                     D + (if B then 1 else if S then -1 else 0) <= -10;
  initial init;
  assertion not (B and S);
  invariant not (Avance and Retard);
```

Par souci de simplicité, on a en fait retardé d'un cycle la détection d'une violation de la propriété (la propriété correspondant à l'observateur `Prop1` est en fait `not (Avance' and Retard')`). □

Sémantique et but de vérification

La sémantique d'un tel programme est donnée par un système de transition discret (Q, τ, Q_{init}), avec

- $Q = \mathbb{B}^m \times \mathcal{N}^p$: un état est défini par la valeur des variables d'état ;
- la relation de transition est définie par

$$\tau\Big((\vec{b},\vec{x})\,,\,(\vec{b}',\vec{x}')\Big) \iff \exists(\vec{\beta},\vec{\xi}) \in \mathbb{B}^n \times \mathcal{N}^q\ assert(\vec{b},\vec{\beta},\vec{x},\vec{\xi}) \ \wedge\ (\vec{b},\vec{x}) \xrightarrow{(\vec{\beta},\vec{\xi})} (\vec{b}',\vec{x}')$$

  avec

$$(\vec{b},\vec{x}) \xrightarrow{(\vec{\beta},\vec{\xi})} (\vec{b}',\vec{x}') \iff \begin{cases} b'_i = \phi_i(\vec{b},\vec{\beta},\vec{x},\vec{\xi}) & i = 1\ldots m \\ x'_k = \psi_k(\vec{b},\vec{\beta},\vec{x},\vec{\xi})) & k = 1\ldots p \end{cases}$$

- $Q_{init} = \{(\vec{b},\vec{x}) \mid init(\vec{b},\vec{x})\}$.

Le but de la vérification est de montrer que $reach(\tau)(Q_{init}) \cap Q_{err} = \emptyset$, avec $Q_{err} = \{(\vec{b},\vec{x}) \mid \neg inv(\vec{b},\vec{x})\}$.

Restriction

Dans la suite, nous nous restreindrons à des programmes dans lesquels les expressions numériques sont *linéaires*. Il est certes toujours possible d'abstraire des expressions plus compliquées dans le treillis des polyèdres : lorsqu'on a une affectation `y:=x*x` et que $0 \leqslant a \leqslant x \leqslant b$, on peut en déduire qu'après l'affectation $0 \leqslant a^2 \leqslant y \leqslant b^2$.

Nous avons fait ce choix, par souci simplicité, et parce qu'il correspond à ce qui est actuellement implémenté dans notre outil de vérification.

3.3.4 Synthèse d'une structure de contrôle : évaluation partielle et automate interprété

Deux solutions extrêmes sont possibles :

- On peut ne pas partitionner du tout l'espace d'état du programme, lequel est considéré comme une boucle simple itérant l'application de la fonction de transition à l'état courant. Cette solution produit des résultats très grossiers et en général inutilisables.
- Une autre solution est d'utiliser le graphe associé au système de transition, cf. §2.1, mais ceci requiert que l'espace des états soit fini.

Dans la représentation symbolique d'un programme, la structure de contrôle est donnée implicitement par la valeur des variables booléennes d'états, et par la valeur de vérité des contraintes sur la valeur des variables numériques d'état. Le compilateur Lustre [Ray91] est capable, à partir de cette représentation de construire un automate de contrôle en effectuant une évaluation partielle de tous les variables booléennes d'état et en minimisant à la volée l'automate produit par bisimulation. On obtient en sortie un automate interprété, que nous appellerons *automate de contrôle booléen*, dont les transitions sont étiquetées par des tests sur les entrées booléennes et les variables numériques, ainsi que par des affectations sur les variables d'état numériques. La figure 3.3 donne l'automate interprété du régulateur de métro.

3.3.5 Vérification de propriétés booléennes

Analyse booléenne

Plusieurs outils développés dans le cadre des langages synchrones (Lesar [Rat92, RHR91], Sigali [LDBL93], TempEst[JPV95] et Xeve [Bou98]) permettent de vérifier des propriétés de sûreté logiques (ne portant que sur des variables booléennes) par exploration de l'espace d'état du programme, en faisant abstraction du comportement des variables numériques. Les tests sur ces variables y sont traités comme des choix indéterministes. Par conséquent, la vérification peut échouer car un surensemble des comportements possibles des variables booléennes du programme est considéré, comme l'illustre bien la figure 3.3.

À titre d'exemple, Lesar propose deux modes de vérification :

Énumératif : l'automate de contrôle défini précédemment est exploré à la volée, état par état. La propriété est violée si on rencontre un état ne la vérifiant pas ;

Symbolique : ce mode utilise les diagrammes de décision binaires pour représenter et manipuler des ensembles d'états, ainsi que la fonction de transition. Les deux techniques d'analyse, avant et arrière, sont implémentées.

Prise en compte des gardes numériquement insatisfaisables

Les méthodes décrites précédemment échouent lorsque la satisfaction dépend directement ou indirectement de la satisfaction de tests numériques. Il est cependant possible de les améliorer en éliminant les transitions *numériquement insatisfaisables* avant la vérification proprement dite.

Lesar met en oeuvre ce type de méthode dans le cas de contraintes *linéaires*, en testant la satisfaisabilité des conjonctions de contraintes linéaires apparaissant dans la fonction de transition, problème pour lequel des procédures de décision efficaces sont disponibles (programmation linéaire [Dan55], polyèdres [Che68, Wil93, FP96]). Il est à noter que même en cas d'échec, ce type d'analyse permet de restreindre l'espace des états booléens à considérer avant d'appliquer une méthode plus puissante — et plus coûteuse.

Par exemple l'application de cette technique à l'automate interprété du régulateur de train permet de supprimer les transitions menant à l'état dans lequel le train est à la fois en retard

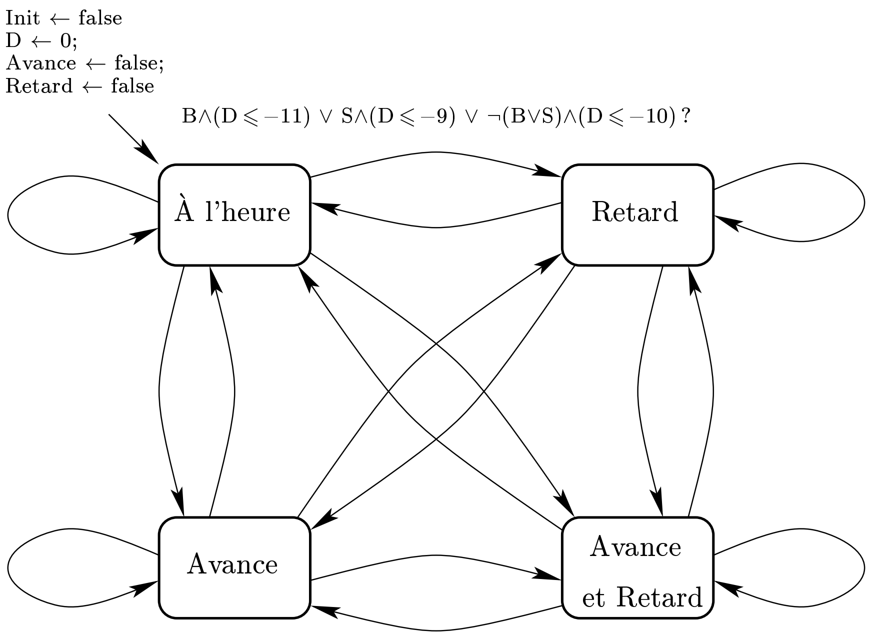


FIG. 3.3 – Automate interprété du régulateur de vitesse du métro. *On a détaillé la garde de la transition "À l'heure"→ "Retard"*

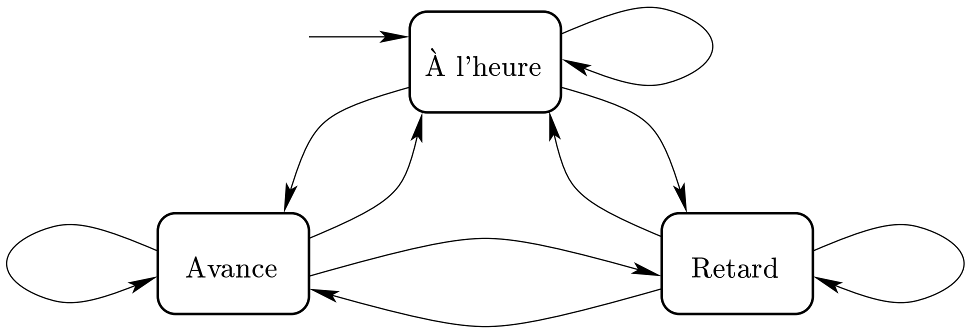


FIG. 3.4 – Automate du métro sans transitions statiquement impossibles.

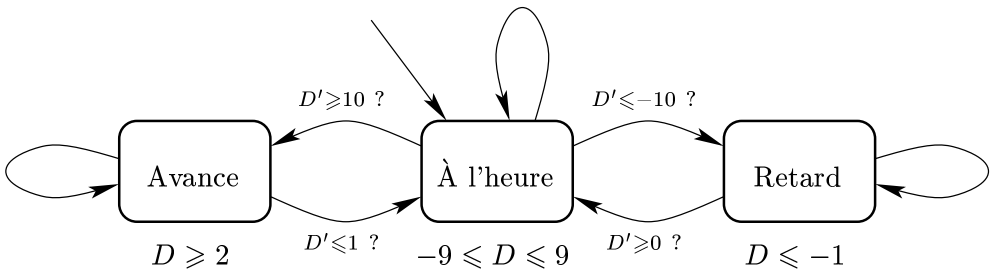


FIG. 3.5 – Automate du métro sans transitions dynamiquement impossibles.

et en avance, ce qui donne après analyse subséquente d'accessibilité booléenne l'automate de la figure 3.4 . Toutefois, le comportement dynamique des variables numériques est toujours ignoré, ce qui empêche par exemple de montrer qu'on ne peut passer directement de l'état "Retard" à l'état "Avance".

3.3.6 Analyse de relations linéaires

Pour montrer cette dernière propriété, il est nécessaire de prendre en compte le comportement numérique du programme. Nous avons vu discuté au §2.3.2 de l'intérêt d'utiliser une structure de contrôle dans les calculs approchés de point fixe sur lesquels sont fondés la vérification de propriétés. [HPR97] propose d'appliquer l'analyse de relations linéaires décrite au §2.4.4 en utilisant comme structure de contrôle l'automate de contrôle booléen d'un programme. À chaque état de cet automate va être associé un polyèdre convexe qui représente, pour la valuation des variables booléennes correspondantes, un surensemble des valeurs accessibles des variables numériques (analyse en avant) ou un surensemble des valeurs des variables numériques pouvant mener à une violation de la propriété (analyse en arrière). La fonction de transition du programme induit un système d'équations de point fixe sur ces polyèdres, système résolu de manière itérative en utilisant l'opérateur d'élargissement standard.

Les résultats d'une telle analyse sur le régulateur de train sont montrés figure 3.5. Comme il n'y a qu'une variable numérique, l'analyse de relations linéaires est ici équivalente à une analyse d'intervalle. Les intervalles admissibles pour la variable D sont données sous les états de contrôle. En particulier, l'analyse montre qu'on ne peut pas passer directement de l'état "Retard" à l'état "Avance".

Cette technique se révèle puissante, car elle permet de découvrir des nouvelles relations entre variables numériques n'apparaissant pas dans le source du programme [Hal79, CC92b].

Toutefois ses inconvénients principaux sont la taille de l'automate de contrôle booléen (problème de l'explosion d'états, bien connue en vérification des systèmes finis), et dans certains cas l'approximation induite par l'abstraction de l'union ensembliste sur les polyèdres par leur enveloppe convexe. La thèse a pour objet de traiter ces deux problèmes, qui sont en fait liés, en jouant sur le compromis entre coût et précision des analyses.

Le chapitre suivant étudie en détail ces deux problèmes, avant de proposer comme solution le *partitionnement dynamique* et de décrire son principe.

Chapitre 4

Principe du partitionnement dynamique

Après avoir examiné les limites de l'analyse de relations linéaires, nous présentons et justifions une solution pour y pallier, fondée sur le partitionnement dynamique d'une structure de contrôle initialement grossière. Ce chapitre constitue donc une présentation des idées de base de cette thèse.

Nous allons d'abord analyser dans un premier temps au §4.1 les problèmes et limites de l'analyse de relations linéaires effectuée sur l'automate de contrôle booléen, en nous aidant de deux exemples caractéristiques ; nous expliquons ensuite au §4.2 pourquoi la précision et la complexité d'une analyse sont liées à la structure de contrôle utilisée.

Nous généralisons dans un second temps au §4.3 la notion de structure de contrôle et distinguons deux types d'analyse que l'on peut effectuer sur les structures de contrôle ainsi définies.

Dans un dernier temps, nous présentons au §4.4 les principes du partitionnement dynamique et les idées générales qui nous guideront dans les chapitres suivants, et dont la mise en œuvre et la démonstration expérimentale de leur efficacité sont la contribution essentielle de cette thèse.

Les idées de chapitre ont pour la plupart été exposées dans [JHR99].

4.1 Limitations de l'analyse de Relation Linéaire et motivations

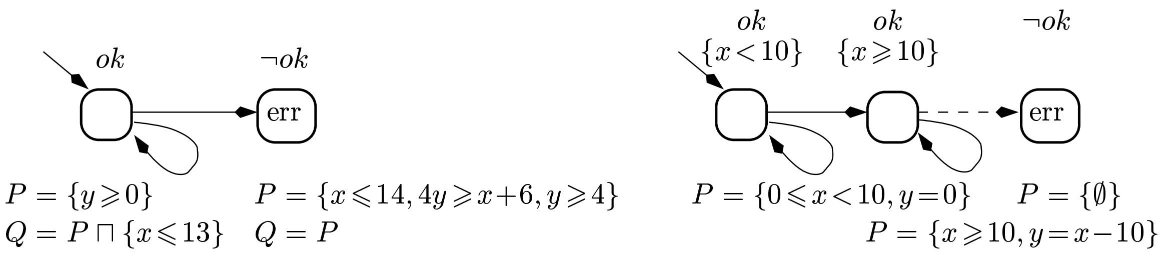
Analysons la technique consistant à appliquer l'analyse de relations linéaires à l'automate de contrôle booléen du programme à vérifier.

4.1.1 Sources d'approximation

Les sources d'approximations de cette technique sont

1. l'abstraction de l'union ensembliste des polyèdres par leur enveloppe convexe ;
2. l'emploi de l'opérateur d'élargissement pour assurer la convergence de l'analyse.

L'approximation engendrée par l'enveloppe convexe se révèle expérimentalement souvent rédhibitoire pour la preuve d'une propriété, en particulier en présence de contrôle numérique,



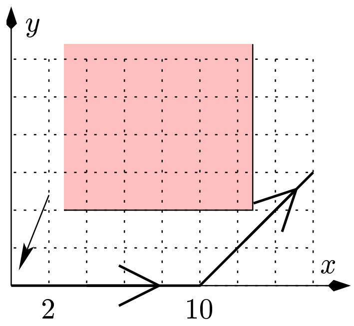
(a) (b)

FIG. 4.1 – Analyse de l'exemple avec contrôle numérique

où il est parfois nécessaire de diviser un état de contrôle selon la satisfaction d'une contrainte numérique, comme l'illustre l'exemple suivant.

EXEMPLE. — Considérons le programme suivant, muni d'une propriété et qui ne possède deux états de contrôle booléen.

```
state ok: bool; x,y: int;
transition
  x' = x+1;
  y' = if x>=10 then y+1 else y;
  ok' = ok and not (x<=13 and y>=4);
initial ok and x=0 and y=0;
invariant ok;
```

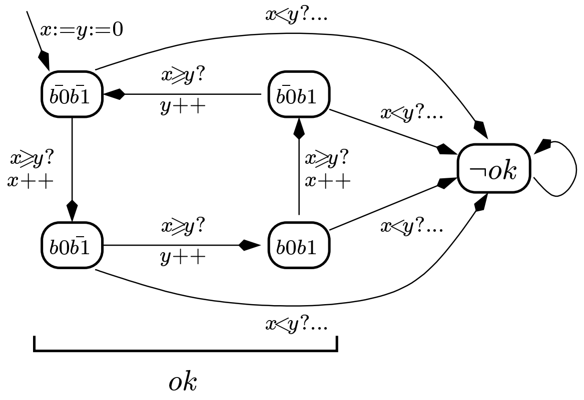


La figure ci-dessus illustre le comportement de ce programme. Initialement x et y sont nuls. Ensuite x est incrémenté à chaque pas. Tant que x est strictement inférieur à 10, y reste nul, et est ensuite incrémenté à chaque pas en même temps que x.

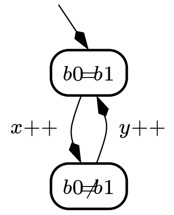
Sur l'automate de contrôle booléen, figure 4.1(a), l'analyse en avant donne les polyèdres P, qui ne permettent pas de conclure. Notons que même si l'élargissement ne perdait pas d'information, à cause de l'enveloppe convexe la place définie par $ok = faux$ semblerait accessible. L'analyse arrière avec intersection avec les états accessibles donne les polyèdres Q. L'état de contrôle défini par $ok = faux$ semble donc accessible depuis l'état initial, et vice-versa.

Maintenant, si on distingue dans l'état de contrôle défini par $ok = vrai$ les valuations numériques suivant qu'elles satisfont ou non la contrainte $x \geqslant 10$, on obtient l'automate de la figure 4.1(b), sur lequel l'analyse avant permet de conclure. □

L'approximation engendrée par l'élargissement est plus rarement gênante, en particulier lorsque l'on utilise les versions raffinées de l'opérateur d'élargissement standard décrites au chapitre 8. Nous n'aborderons pas directement le contrôle de ces approximations, mais typiquement la division d'un état de contrôle suivant une contrainte numérique peut améliorer considérablement l'élargissement. Dans l'exemple précédent, l'élargissement sur l'automate de contrôle booléen a fait perdre l'information $x \geqslant 0$ à l'état de contrôle défini par ok, mais la division de cet état permet d'obtenir un résultat exact, même avec utilisation de l'élargissement.



(a) Automate de contrôle booléen



(b) Une structure plus simple

FIG. 4.2 – Deux structures de contrôle pour le même programme

4.1.2 Sources de complexité

Les sources de complexité sont quant à elles les suivantes :

1. le nombre de variables numériques ;
2. le nombre d'états de contrôle.

Le nombre de variables numériques a une grande influence sur le coût de l'analyse : en effet les opérations d'enveloppe convexe et d'intersection de deux polyèdres ont une complexité potentiellement exponentielle en le nombre de variables, cf. §2.4.4. Certaines possibilités d'amélioration existent cependant, comme celle consistant à représenter les polyèdres par des produits cartésiens de polyèdres de dimension plus petite lorsque cela est possible. Néanmoins, nous n'aborderons pas ce problème ici.

Le nombre d'états de contrôle pose également problème. En effet il est exponentiel en le nombre de variables d'état booléennes, et celui-ci est couramment de l'ordre de plusieurs dizaines dans les programmes de taille moyenne. Or il est fréquent que certains états d'une certaine manière "équivalents" par rapport à la propriété comme le montre l'exemple ci-dessous.

EXEMPLE. — Considérons le programme suivant :

```
state b0,b1,ok: bool; x,y: int;
transition
  b0' = not b1;
  b1' = b0;
  x' = if b0=b1 then x+1 else x;
  y' = if b0 xor b1 then y+1 else y;
  ok' = ok and (x>=y);
initial ok and not b0 and not b1 and x=0 and y=0;
invariant ok;
```

Le comportement de ce programme apparaît plus clairement sur son automate de contrôle booléen, représenté figure 4.2(a). Initialement les deux variables numériques x et y sont nulles.

Ensuite x est incrémenté, puis y, puis x, et enfin y. À la fin d'un tel cycle, on retombe sur l'état de contrôle initial et on recommence un nouveau cycle. Il est donc clair que la propriété $x \geqslant y$ est vérifiée par ce programme.

L'analyse en avant donne les résultats du tableau ci-dessous :

	$\overline{b0}\,\overline{b1}$	$b0\overline{b1}$	$b0b1$	$\overline{b0}b1$	$\neg ok$
P	$\{x = y, x \geqslant 0\}$	$\{x = y+1, x \geqslant 1\}$	$\{x = y, x \geqslant 1\}$	$\{x = y+1, x \geqslant 2\}$	$\perp$

et permet la preuve de la propriété. Mais la structure à deux points de contrôle de la figure 4.2(b) aurait suffi à cette preuve, car les états à partir desquels on incrémente x sont "équivalents" du point de vue de la propriété, et de même pour les états à partir desquels on incrémente y. On aimerait donc pouvoir fusionner ces états booléens, c'est-à-dire ne pas les distinguer dans la structure de contrôle de façon à ce que l'analyse leur associe le même polyèdre. □

Certes, une notion de *bisimulation* adaptée permettrait d'améliorer les choses. Cette bisimulation devrait capturer, sur l'automate booléen, la sémantique des gardes sur les entrées booléennes et les contraintes numériques, ainsi que les affectations sur les variables numériques. [CGS91] définit une bisimulation sur des processus booléens, qui sont de systèmes de transition dont les étiquettes sont des formules booléennes et ont donc une sémantique. Avec cette bisimulation, les deux automates de la figure 4.3 sont équivalents.

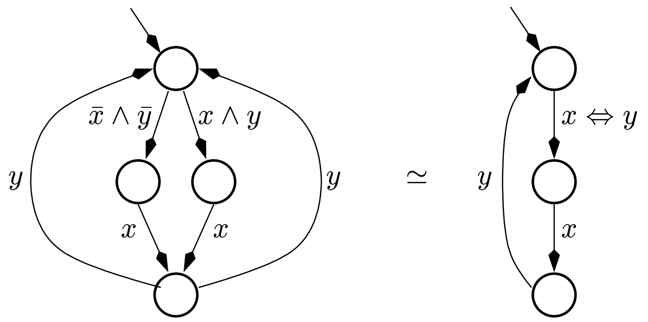


FIG. 4.3 – Bisimulation sur des processus booléens

On peut utiliser cette bisimulation dans notre contexte, en considérant les contraintes numériques comme des entrées et les affectations comme des étiquettes supplémentaires. L'outil OCMIN développé dans l'équipe LUSTRE implémente la minimisation d'un automate booléen en utilisant *grosso-modo* une telle bisimulation. Cependant cette minimisation est coûteuse en elle-même. En outre,

- un grand nombre de ces états se révéleront inatteignables après l'analyse ; il n'est donc pas judicieux de construire un énorme automate interprété, construction coûteuse en temps comme en espace ;
- beaucoup des états de contrôle sont équivalents du point de vue de la propriété — sans être pour autant bisimilaires, comme l'a illustré l'exemple ci-dessus ; il est pourtant souhaitable de pouvoir fusionner ces états de contrôle non bisimilaires ;
- comme nous l'avons vu avec le premier exemple, il est quelquefois nécessaire de séparer des états de contrôle selon une contrainte numérique.

4.2 Le problème de la structure de contrôle

Les phénomènes précédents sont liés à la structure de contrôle. En fait, l'utilisation d'une structure de contrôle pour la résolution d'une équation de point fixe revient à manipuler des unions explicites de valeurs abstraites au lieu d'une seule valeur abstraite. Le grand intérêt d'utiliser une telle structure est le contrôle de la taille de l'union explicite, puisqu'on ne peut avoir qu'un seule valeur abstraite par point de contrôle. Ainsi la structure de contrôle permet de jouer sur la précision et la complexité de l'analyse : une structure de contrôle détaillée permet une meilleure précision, mais au prix d'une complexité plus élevée.

Pour résumer la discussion précédente, dans notre contexte, la structure de contrôle booléenne est selon le cas trop fine ou trop grossière.

1. Trop fine dans le cas du programme de la fig 4.2, où la structure de contrôle de la fig 4.2(b) suffit à montrer la propriété souhaitée, au lieu de l'automate de contrôle booléen de la fig 4.2(a). Dans la fig 4.2(b), on a simplement fusionné des états booléens, si bien que le même polyèdre sera associé à plusieurs états de l'automate de contrôle booléen.
2. Trop grossière dans le cas du programme de la fig 4.1. Dans ce cas, la séparation d'un état selon la satisfaction d'une contrainte numérique consiste à associer à un état de contrôle booléen une union de deux polyèdres, séparés par la contrainte.

En fait, même après une analyse de cône d'influence, les différentes parties du programme n'influent pas de la même façon sur la satisfaction de la propriété. Certaines parties nécessitent d'être analysées finement, par exemple en utilisant des unions explicites de polyèdre, tandis que des approximations grossières sur d'autres parties ne sont pas rédhibitoires, lesquelles peuvent consister à associer le même polyèdre à un grand nombre d'états de contrôle booléens.

La structure de contrôle idéale ne peut cependant pas être trouvée par simple inspection du source du programme, car elle dépend de sa dynamique. Le principe du partitionnement dynamique, décrit au paragraphe suivant, consiste à démarrer avec une structure de contrôle grossière et à la raffiner progressivement d'après le résultat des analyses, jusqu'au succès de la preuve (si celle-ci est possible).

4.3 Structure de contrôle et différents types d'analyse

4.3.1 Structure de contrôle

Soit un système de transition (Q, τ, Q_{init}) et une propriété spécifiée par un ensemble d'états erronés $Q_{err} \subset Q$.

On suppose que l'on dispose d'un treillis abstrait A de prédicats sur l'espace d'état Q, dont les éléments forment une famille de Moore de 2^Q, et dans lequel la borne supérieure fait perdre de l'information.

Soit $\bigcup_{0 \leqslant i \leqslant n} Q_i$ une couverture de l'espace d'état Q. Soient $P_i = \alpha(Q_i)$ les prédicats abstraits associés aux Q_i. *La structure de contrôle engendrée par la couverture de* Q est un graphe à n sommets, chaque prédicat abstrait P_i définissant un sommet q_i. Nous appellerons *places* les sommets de ce graphe et *arcs* les arêtes reliant les places ; il existe un arc $q_i \rightsquigarrow q_j$ entre deux places q_i et q_j dès qu'un état concret appartenant à Q_i a un successeur appartenant à Q_j par la fonction de transition. Dans ce graphe une place est *initiale* si elle contient un état initial, *finale* si elle contient un état erroné, et *correcte* si elle ne contient aucun état erroné.

Ce graphe est une abstraction du programme et représente un surensemble de ses exécutions possibles.

Avant toute analyse, le graphe est complet, puisqu'on n'a pas encore testé, pour chaque couple (i, j), si Q_i contient des états tels que leurs successeurs intersectent Q_j.

4.3.2 Analyse "statique" des transitions

Un premier type d'analyse consiste à tester pour chaque arc $q_i \rightsquigarrow q_j$ du graphe, par une procédure de décision appropriée, si $post(P_i) \sqcap P_j = \perp$. Si l'on montre cette propriété, on peut ôter l'arc car celui-ci ne correspond à aucune transition concrète.

Après cette opération, effectuée sur chaque arc du graphe, on peut mener une analyse d'accessibilité avant — au sens de la théorie des graphes — depuis les places initiales, qui permet éventuellement de montrer qu'aucune place finale n'est accessible et que donc il n'existe aucune exécution menant à un état erroné. On peut de manière similaire mener une analyse en arrière depuis les places finales.

L'analyse booléenne prenant en compte la satisfaisabilité des contraintes numériques, décrite au §3.3.5, correspond à ce type d'analyse menée sur l'automate de contrôle booléen, c'est-à-dire la structure de contrôle définie par la couverture de Q par l'ensemble des monômes complets sur les variables d'états booléennes.

Les travaux cherchant à combiner vérification de modèle et preuve [SUM96, GS97, BLO98, CU98] s'inscrivent également dans ce schéma, pour leur principe de base : on génère un graphe de contrôle à partir d'une partition de l'espace d'état, on utilise des démonstrateurs automatiques pour éliminer le plus possible d'arcs du graphe de contrôle et éventuellement générer des invariants auxiliaires, puis on emploie des techniques de vérification de modèle pour vérifier si le graphe résultant est un modèle de la formule de logique temporelle spécifiant la propriété. Bien sûr, le choix de la structure de contrôle est d'une grande importance, puisqu'elle définit à elle seule le treillis abstrait utilisé pour l'analyse. [SUM96] propose de prouver des formules de logique temporelle, en utilisant une méthode de tableau dans laquelle les formules sont partitionnées dynamiquement. [CU98] demande à l'utilisateur de spécifier une base de prédicats $B = \{b_1, \dots b_n\}$ puis énumère leurs valeurs de vérité, tandis que dans [GS97, BLO98] la base est construite automatiquement à partir des gardes apparaissant dans le programme.

4.3.3 Analyse "dynamique"

Comme nous l'avons vu au §3.3.6, il est quelquefois nécessaire d'analyser la dynamique du programme pour prouver certaines propriétés. Nous entendons par *analyse dynamique* une analyse d'accessibilité, avant ou arrière, procédant par calcul de plus petit point fixe, et calculant un sous-ensemble des états concrets représentés par une place, tel que décrit au §2.3, et appliqué dans les articles référencés au §2.4.

Ce type d'analyse donne des résultats beaucoup plus précis qu'une analyse statique. En effet, dans une analyse statique, il n'y a de propagation d'information que d'une place à une autre, qui permet éventuellement de supprimer un arc, tandis que au cours d'une analyse dynamique la propagation d'information est itérée sur tout le graphe.

Un autre point de vue est qu'on ne se limite plus à l'abstraction finie du programme donnée par la structure de contrôle. En particulier les analyses dynamiques permettent de découvrir automatiquement des *invariants auxiliaires*. Or il est bien connu en démonstration automatique, où même lors de preuves manuelles, qu'il est souvent nécessaire d'utiliser de tels invariants pour parvenir au but. Par exemple, l'analyse de relations linéaires permet la découverte de relations n'apparaissant pas dans le source du programme, et qui peuvent être considérées comme des invariants auxiliaires.

Bien sûr, si les opérations du treillis abstrait ont une complexité élevée, comme c'est le cas des polyèdres convexes, les analyses dynamiques peuvent être beaucoup plus coûteuses que les analyses statiques.

Par exemple [TBL99] utilise des analyses dynamiques pour vérifier des formules CTL (Computation Tree Logic [CES86]) sur des programmes comportant des entiers non bornés. L'arithmétique de Presburger est utilisée pour représenter de manière exacte les ensembles numériques. La vérification d'une formule CTL consiste à calculer des plus petits et plus grands points fixes, éventuellement imbriqués. Les calculs peuvent être exacts ou approchés, mais dans les deux cas la convergence n'est pas garantie ; en particulier l'opérateur d'élargissement utilisé pour les analyses approchées n'en est pas au sens de [CC77], car il ne garantit pas la convergence[1].

Les travaux combinant vérification de modèle et preuve font également usage d'analyses dynamiques — mais de manière plus limitée — en utilisant par exemple des techniques de compilation : propagation de constantes, analyse de vivacité, etc... , afin d'inférer des invariants auxiliaires et de réduire l'espace des états à considérer [BBM97, BLS96]. En revanche, comme le domaine des propriétés manipulées est très complexe dans ces travaux, on ne dispose pas d'opérateur d'élargissement, et il n'est pas possible de mener une analyse d'accessibilité par calcul de plus petit point fixe sur ce domaine.

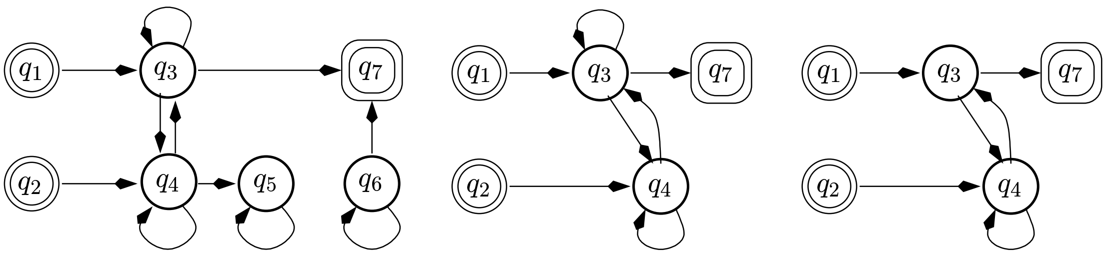
4.4 Partitionnement dynamique d'une structure de contrôle

4.4.1 Combinaison des analyses avant et arrière : états dangereux et contre-exemples

Nous allons reformuler le principe classique de la combinaison des analyses avant et arrière sur des structures de contrôle, en utilisant la notion de *contre-exemple* et d'état *dangereux*. Une exécution est un *contre-exemple* si elle contient un état erroné. Un état est *dangereux* s'il appartient à un contre-exemple, ce qui équivalent au fait qu'il est à la fois accessible (depuis un état initial) et coaccessible (depuis un état final ou erroné). Une place sera dangereuse si elle contient un état dangereux.

Une première idée, permettant de simplifier la structure de contrôle, est de ne considérer que les états dangereux, de manière à ce que le graphe de contrôle représente l'ensemble des exécutions qui sont potentiellement des contre-exemples à la propriété. "Potentiellement" signifie ici que ces exécutions apparaissent comme des contre-exemples soit parce qu'elles en sont réellement, soit à cause des approximations effectuées. Cette simplification, outre le gain évident en temps et en espace pour les analyses, permet aussi une meilleure précision lors des analyses dynamiques : comme les transitions ayant pour extrémité une place non dangereuse sont supprimées, l'opérateur de borne supérieure s'applique en chaque place à un nombre plus réduit de valeurs abstraites.

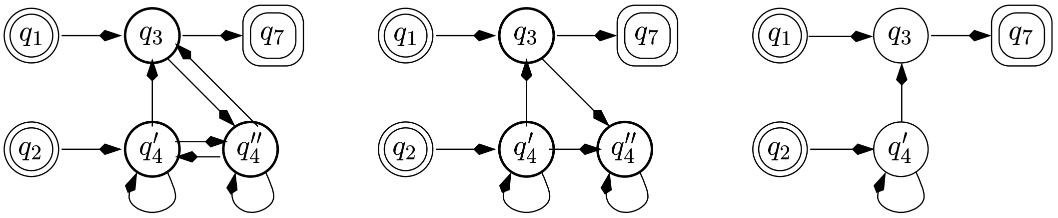
[1][TBL99] mentionne bien sûr le problème de la convergence, mais n'évoque aucune preuve de terminaison lorsque l'élargissement qu'il définit est utilisé



(a) Analyse statique sur le graphe initial

(b) Élimination des places non dangereuses

(c) Analyse dynamique



(d) Division de q_4

(e) Analyse statique

(f) Élimination des places non dangereuses

FIG. 4.4 – Graphe de contrôle. Deux cercles concentriques désignent une place initiale, deux carrés arrondis une place finale, et un cercle en trait épais une place correcte.

4.4.2 Analyse statique et élimination des places non dangereuses

Soit un graphe de contrôle, dans laquelle on a éliminé des arcs par une analyse statique. La figure 4.4(a) montre un tel graphe. À l'évidence, si dans ce graphe il n'existe aucun chemin reliant une place initiale à une place finale, la propriété est vérifiée : il est impossible d'extraire un contre-exemple de la propriété parmi le surensemble des exécutions possibles que représente le graphe.

Sinon, on peut éventuellement simplifier le graphe, en supprimant les places non dangereuses, c'est-à-dire n'appartenant pas à un chemin reliant une place initiale à une place finale. Considérons une telle place q_i dont le prédicat associé est P_i. Deux cas se présentent : soit il n'existe pas de chemin depuis une place initiale jusqu'à cette place (cas de la place q_6 fig 4.4(a)) : cela signifie qu'aucune exécution ne peut atteindre l'ensemble d'états P_i, et que donc aucun de ces états ne peut appartenir à un contre-exemple de la propriété. Soit il n'existe pas de chemin depuis cette place jusqu'à une place finale (cas de la place q_4 fig 4.4(a)), et dans ce cas les états de P_i peuvent appartenir à une exécution possible, mais pas à un contre-exemple. Dans les deux cas, on peut donc supprimer la place, sans supprimer aucun des contre-exemples potentiels, car celle-ci ne contient aucun état dangereux. Dans le cas du graphe figure 4.4(a), on obtient le graphe de la figure 4.4(b).

Cette simplification permet un gain en espace, et réduit également le coût des analyses dynamiques subséquentes.

4.4.3 Analyse dynamique et réduction des définitions

Si après la simplification du graphe selon la méthode précédente, il existe encore des contre-exemples, on peut mener une analyse (dynamique) en avant. Celle-ci va calculer des prédicats $P'_i \sqsubset P_i$, qui représentent une surapproximation de la partie accessible de P_i. Les états représentés par $P_i \setminus P'_i$ n'étant pas accessibles, ils ne sont pas dangereux et n'appartiennent à aucun contre-exemple. On peut les supprimer dans la structure de contrôle, en remplaçant les P_i par les P'_i, ce qui peut le cas échéant permettre de supprimer des arcs dans le graphe, fig 4.4(c).

On peut de même mener une analyse (dynamique) en arrière, qui va calculer des prédicats $P''_i \sqsubset P_i$, qui représentent une surapproximation des états pouvant mener à une violation de la propriété. Les états appartenant à $P_i \setminus P''_i$ n'appartiennent à aucun contre-exemple et peuvent être supprimés de la structure de contrôle en remplaçant les P_i par les P''_i, réduction qui peut induire des suppressions d'arcs dans le graphe.

On peut ainsi combiner ces analyses jusqu'au succès de la preuve, ou jusqu'à convergence.

Comme nous l'avons mentionné au §4.3.3, l'analyse statique obtient des résultats strictement moins précis que l'analyse dynamique, mais elle est aussi nettement moins coûteuse dans notre cas. Il est donc intéressant de précéder les analyses dynamiques par une analyse statique en avant suivie d'une analyse statique en arrière.

4.4.4 Division de place et raffinement de structure de contrôle

Si les analyses ont convergé, sans succès pour la preuve de la propriété, il existe deux possibilités : la propriété est fausse, ou les approximations sont trop grossières. Nous allons nous placer dans le second cas. Supposons qu'après convergence on obtiennent le graphe de la figure 4.4(c). Le principe du raffinement est de remarquer que les états représentés par P_4 par exemple ont des origines différentes : certains proviennent de l'arc $q_2 \rightsquigarrow q_4$ tandis que d'autres proviennent de l'arc $q_3 \rightsquigarrow q_4$ ou de la boucle sur q_4. Suivant leur origine, il est possible que le futur de ces états soit différent, mais la structure de contrôle ne distingue pas ces différentes catégories d'état.

Divisons par exemple q_4 en q'_4 et q''_4 suivant les prédécesseurs q_2 et q_3. Avant toute analyse, tous les arcs entrants et sortant de q_4 sont dupliqués, cf. fig 4.4(d). Une analyse statique peut être effectuée, fig 4.4(e), qui permet de mieux distinguer les comportements des états dans q'_4 et q''_4, et de montrer que la place q''_4 n'est plus dangereuse et peut être éliminée. La division de q_4 a ainsi permis de restreindre localement l'espace des états dangereux, et cette restriction se propage ensuite au reste du graphe lors d'une analyse dynamique.

Même si l'analyse statique ne permet pas d'éliminer un des deux états issus de la division, l'analyse dynamique sera plus précise, car les états dangereux considérés dans la place q_4, auparavant représentés par une seule valeur abstraite P_4, sont maintenant représentés par une union explicite de deux valeurs abstraites $P'_4 \cup P''_4$.

4.4.5 Utilisation de l'évaluation partielle

Comme nous l'avons mentionné, l'intérêt de ne considérer que les états dangereux est de simplifier la structure de contrôle, ce qui implique une réduction du coût en temps et en espace ainsi qu'une meilleure précision des calculs. De plus, ceci permet l'utilisation de techniques d'*évaluation partielle* pour simplifier la fonction de transition, celle-ci n'étant appliquée qu'à un sous-ensemble de l'espace des états.

Le raffinement de la structure de contrôle a en réalité deux conséquences sur l'efficacité de l'évaluation partielle de la fonction de transition :

- de manière *globale*, il restreint l'espace des états dangereux, puisque la précision des analyses augmente ;
- *localement*, comme en fait on spécialise la fonction de transition sur chaque place, plus celles-ci sont nombreuses, plus le sous-ensemble d'états à considérer pour chacune d'elle est petit, et plus l'évaluation partielle est efficace.

Ainsi, au fûr et à mesure que l'on raffine la structure de contrôle, celle-ci a tendance à grossir, mais les calculs de postconditions et préconditions de valeurs abstraites deviennent plus économiques. Expérimentalement, dans le treillis abstrait utilisé pour la vérification de programmes synchrones, défini au §5.5, il a été constaté que les analyses deviennent de moins en moins chères, l'augmentation de la taille de la structure de contrôle au cours du raffinement étant plus que compensée par la simplification de la fonction de transition.

4.4.6 Approximation des opérateurs de post- et pré- conditions et lien avec l'évaluation partielle

Conceptuellement, et parfois algorithmiquement, il est plus simple d'utiliser la meilleure approximation $\alpha(f) = \alpha \circ f \circ \gamma$ d'un opérateur concret f, que de définir et d'utiliser une approximation moins précise mais restant raisonnable. Dans le schéma esquissé ci-dessus, la structure de contrôle initiale, ainsi que celles obtenue après les premières divisions de places, restent grossières et les bornes supérieures effectuées sur chaque place lors des analyses perdent beaucoup d'information. Dans ce contexte, il n'est pas forcément judicieux de calculer avec grande précision post- et pré- conditions, sachant que leur résultat va être fusionné de manière imprécise avec d'autres valeurs abstraites.

Il apparaît donc raisonnable de lier la précision avec laquelle sont effectués les calculs de post- et pré- conditions à la précision engendrée par la structure de contrôle utilisée.

Dans notre cadre d'application, nous avons utilisé une méthode issue des techniques de compilation du langage Lustre, [Ray91], pour effectuer ces calculs de manière approchée. Dans cette méthode, les transitions sont en fait des graphes acycliques, dont les sommets sont appelés *micro-états* et représentent des étapes intermédiaires dans le calcul de l'état successeur d'un état concret, figure 4.5. Lorsqu'on applique cette technique de compilation pour calculer des postconditions (ou préconditions) de manière symbolique, des bornes supérieures doivent être appliquées sur certains de ces micro-états pour fusionner les résultats de leurs branches entrantes, comme le montre la figure 4.5.

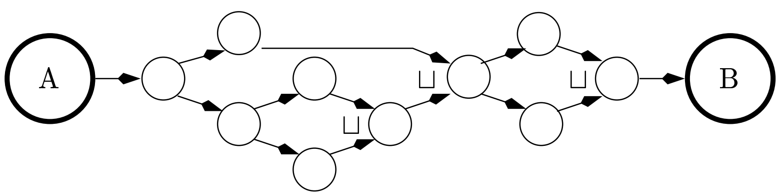


Fig. 4.5 – Calcul de postcondition permettant de passer de la place A à la place B

Fondamentalement, cette méthode consiste donc à appliquer des bornes supérieures au cours des calculs intermédiaires permettant de passer d'une place du graphe à une autre, en plus des bornes supérieures appliquées sur les places elles-mêmes, et elle est généralisable à d'autres cadres d'application.

Maintenant, au cours du raffinement de la structure de contrôle, la fonction de transition devient de plus en plus simple à cause de l'évaluation partielle ; une très bonne propriété de notre technique est que plus la fonction de transition est simple, moins les approximations effectuées dans les calculs de post- et pré- conditions sont nombreuses

Ainsi, la combinaison du raffinement de la structure de contrôle, de l'évaluation partielle et de notre technique d'approximation des opérateurs de post- et pré- condition permet de régler le degré d'approximation des transformateurs de prédicats à la finesse de la structure de contrôle, ce qui est le but souhaité.

4.5 Travaux reliés

Précision induite par l'opérateur de borne supérieure. — Le problème de l'approximation induite par l'opération de borne supérieure sur un treillis abstrait L est bien connu. Une solution naturelle est d'utiliser la *complétion disjonctive* de L [CC92a], qui consiste essentiellement à manipuler des unions non redondantes d'éléments de L. [Bou92a] propose une version raffinée de cette solution, dans laquelle le nombre de disjonctions est limité par l'application d'opérateurs d'élargissement adéquats, et l'applique au calcul de graphes de fonction minimaux. Le terme “partitionnment dynamique” est employé dans ce contexte. Il est cependant douteux qu'une telle démarche soit efficace dans notre cas : utiliser la décomposition en nœuds[2] des programmes synchrones et appliquer des analyses interprocédurales mène à des résultats trop grossiers. Et surtout, cette solution est utilisée dans le seul but de calculer une bonne abstraction d'éléments concrets : les calculs ne sont pas guidés par une propriété à prouver.

Utilisation d'une structure de contrôle en vérification. — De nombreux travaux utilisent une structure de contrôle pour régler le degré de précision des analyses. C'est le cas des travaux qui cherchent à vérifier des systèmes infinis en appliquant des méthodes de vérification de modèle sur des abstractions finies des systèmes étudiés. [CGL94] en présente le principe. Suivant que l'abstraction est fine ou grossière, le graphe fini qui est généré est plus ou moins détaillée, et les propriétés pouvant être prouvées seront plus ou moins subtiles. Les abstractions les plus classiques sont par exemple l'abstraction d'un entier par sa parité, par son signe, par une congruence modulo n ... En général l'abstraction est définie manuellement.

Les travaux combinant preuve et vérification de modèle cités au §4.3.2 ([GS97, BLO98, CU98]) utilisent également une structure de contrôle pour contrôler la précision des analyses. Par comparaison aux travaux évoqués ci-dessus, ils peuvent être vus comme une application plus subtile — et plus coûteuse — des techniques de vérification de modèle aux systèmes infinis.

[TBL99], que nous avons cité au §4.3.2, utilise également un partitionnement de l'espace d'état, mais plus pour simplifier les calculs que pour régler la précision de l'analyse. En effet, par défaut il ne fait pas usage d'approximation. Toutefois, lorsqu'il utilise une analyse approximée, le partitionnement utilisé a de l'influence sur la précision.

[2] qui sont des fonctions flot de donnée, c.f. §3.2

Raffinement automatique d'une structure de contrôle. — Les travaux que nous venons de citer utilisent une structure de contrôle *fixée*. En cas d'échec, l'utilisateur doit spécifier manuellement une nouvelle structure de contrôle. Le principe de trouver automatiquement un compromis entre précision et complexité des analyses de programme en jouant sur la structure de contrôle apparaît dans [SUM96] et [DWT95].

[SUM96] cherche à prouver des propriétés de logique temporelle (pouvant combiner sûreté et vivacité) sur des systèmes de transitions avec contraintes d'équité. Pour cela, à partir d'une propriété ϕ, on construit un *tableau* $\Phi_{\neg\phi}$, qui représente l'ensemble des contre-modèles de la formule. La procédure classique, valable dans le cas de systèmes finis, est de construire le *graphe produit* de $\Phi_{\neg\phi}$ et du système de transition, qui représente l'ensemble des exécutions du système de transition qui sont des contre-exemples de la propriété. Il suffit alors de vérifier que le graphe ne contient aucune exécution équitable. L'idée de [SUM96], qui vérifie des sytèmes infinis, est de démarrer la vérification avec le tableau $\Phi_{\neg\phi}$ et de prendre en compte progressivement le système de transition. Lorsqu'un contre-exemple apparaît, on essaie de raffiner des nœuds suivant la postcondition d'un arc entrant ou la précondition d'un arc sortant.

[WT94, DWT95] s'intéresse à la vérification de propriétés de sûreté sur des automates temporisés, et constitue sans doute le travail qui partage le plus d'idées en commun avec le nôtre. Il exploite la combinaison d'analyses avant et arrière, et repose sur l'usage conjoint de sous- et sur- approximations pour le raffinement. L'idée fondamentale consiste à remarquer que lorsque par exemple une surapproximation des états accessibles intersecte une sous-approximation des états menant à une violation de la propriété, la surapproximation est trop grossière et il convient de raffiner la structure de contrôle. Le chapitre 9 présentera plus en détail ce processus de raffinement. Il est également fait usage dans ce travail de transformateurs de prédicats approchés, le chapitre 7 y fait référence de manière détaillée.

Deuxième partie

Mise en oeuvre du partitionnement dynamique

Chapitre 5

Fonctions, formules et domaine abstrait

Après avoir exposé au chapitre précédent les motivations et le principe du partitionnement dynamique, nous discutons du choix d'un domaine abstrait permettant sa mise en œuvre dans le cadre de la vérification de programmes synchrones booléens et numériques. Ce choix est lié à la représentation des fonctions de transition et des formules du programme, c'est pourquoi nous abordons également cette question.

L'ESPACE d'état d'un programme à analyser étant de la forme $S = \mathbb{B}^n \times \mathcal{N}^p$, la fonction de transition globale du programme se décompose donc en n fonctions booléennes et p fonctions numériques, donnant chacune la valeur d'une variable d'état dans l'instant suivant en fonction de l'état courant et de l'entrée, cf §3.3.3. Ces *fonctions* sont initialement spécifiées, suivant leur type, soit par des formules booléennes dont les atomes sont des variables booléennes ou des contraintes linéaires, soit par des affectations affines gardées par de telles formules. Nous utiliserons de nombreuses opérations sur ces fonctions : substitution, évaluation partielle, calcul d'image, etc.

Un programme à vérifier est spécifié non seulement par sa dynamique (i.e., sa fonction de transition globale) mais aussi par des *formules* définissant l'espace d'états initiaux, ainsi que la propriété et l'assertion sous laquelle la propriété est supposée être vérifiée. Ces formules sont en fait des fonctions booléennes de même signature que les fonctions de transition booléennes.

Une fonction booléenne de signature $S \to \mathbb{B}$ pouvant être interprétée comme la fonction caractéristique d'un sous-ensemble de S, la représentation des formules peut être aussi utilisée comme domaine abstrait de prédicats, et c'est pourquoi nous abordons dans le même chapitre la représentation des fonctions de transition et le choix du domaine abstrait. Une autre raison est que le calcul de post- et pré- condition nécessite la manipulation conjointe des deux représentations.

Fonctions de transition et formules. — Concernant les fonctions booléennes de signature $\mathbb{B}^n \to \mathbb{B}$, les *diagrammes de décisions binaires* ou BDDs [Bry86] constituent une représentation reconnue comme particulièrement efficace. Une solution pour représenter les fonctions de transition booléennes ou certains sous-ensembles de $\mathbb{B}^n \times \mathcal{N}^p$ consiste donc à généraliser ces diagrammes en prenant des décisions non seulement sur la valeur de vérité d'une variable booléenne, mais aussi sur la satisfaction d'une contrainte numérique. Nous appellerons de tels diagrammes *diagrammes interprétés*. Par ailleurs la généralisation des BDDs aux MTBDDs (Multi-Terminal Decision Diagram) [CMZ+93, BFG+93, Ray91] permet la représentation des fonctions de transition numériques.

Domaine abstrait. — Étant donné l'espace d'état $S = \mathbb{B}^n \times \mathcal{N}^p$, la mise en œuvre du partitionnement dynamique nécessite le choix d'une famille de Moore de $C = 2^S$ définissant un domaine abstrait de prédicats sur S, sur lequel les tests d'inclusion et d'égalité, les opérations de borne inférieure et supérieure, ainsi que les transformateurs de prédicats puissent être implémentés efficacement. Il faut également disposer d'un opérateur d'élargissement.

Le problème de la représentation symbolique de sous-ensembles de $\mathbb{B}^n \times \mathcal{N}^p$ a déjà fait l'objet de propositions dans la littérature. La difficulté essentielle pour représenter de tels sous-ensembles réside dans le fait qu'il est souhaitable de pouvoir établir des relations entre partie booléenne et partie numérique. Considérons par exemple le sous-ensemble défini par la formule $b \Leftrightarrow x > 0$, où b est une variable booléenne et x une variable numérique. Dans ce sous-ensemble, parties booléenne et numérique ne sont pas *indépendantes*, contrairement à ce qui se passe avec $b \wedge x > 0$ où l'appartenance d'une valeur concrète se décide *séparément* sur la partie booléenne et sur la partie numérique.

Une première famille de solutions consiste à utiliser les BDDs interprétés évoqués ci-dessus comme prédicats abstraits [MLAH99, BLP+99, Mau96]. Ceci permet de lier finement les parties booléenne et numérique, mais présente également des inconvénients, dont l'un des plus pénalisants du point de vue de la complexité est l'absence d'approximations induites par la borne supérieure. Ceci nous amène à proposer une autre famille de solutions, nettement plus simples, qui renoncent à la possibilité de combiner finement propriétés booléennes et numériques avec une seule valeur abstraite, et qui se reposent sur la structure de contrôle et son raffinement pour établir des connexions précises entre les deux types de propriétés.

Plan du chapitre. — Après avoir introduit les BDDs au §5.1, le §5.2 présente leur généralisation dans laquelle les décisions se prennent à la fois sur la valeur de variables booléennes et la satisfaction de contraintes numériques. Le principal effet de l'introduction de contraintes est que les décisions ne sont plus nécessairement indépendantes entre elles.

Le §5.3 décrit l'utilisation des diagrammes interprétés pour représenter les fonctions de transition. Le §5.4 discute de l'utilisation de ces diagrammes comme treillis de prédicats abstraits, et explique pourquoi nous avons opté pour un autre type de solution présenté au §5.5

5.1 Les diagrammes de décision classiques

Les BDDs constituent une représentation compacte des fonctions booléennes de $\mathbb{B}^n$ dans $\mathbb{B}$. Cet espace fonctionnel est fini, mais de taille exponentielle 2^n. De telles fonctions permettent de représenter des fonctions de transition booléennes, mais peuvent aussi être interprétée comme des fonctions caractéristiques de sous-ensembles de B^n, et plus généralement d'ensembles finis codés à l'aide de variables booléennes. C'est pourquoi ils sont à la base de la plupart des méthodes symboliques d'analyse de systèmes finis.

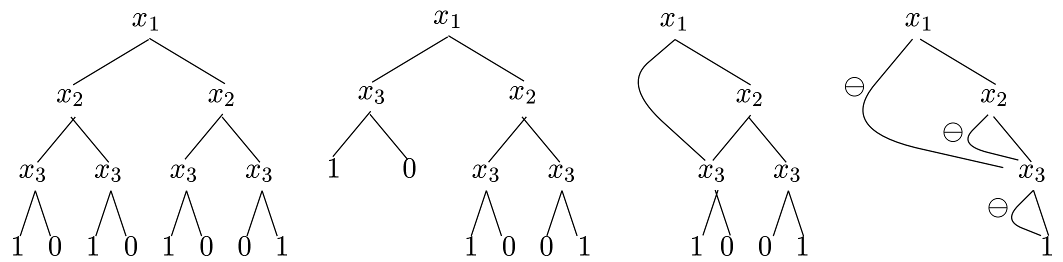
5.1.1 Principes des BDDs

Soit $x_1, x_2, \ldots, x_n$ un ensemble de variables booléennes et f une expression booléenne sur ces variables, définissant donc une fonction $\mathbb{B}^n$ dans $\mathbb{B}$.

Représentation. — La *décomposition de Shannon* de f selon la variable x_k est le couple unique de formules

$$f_{\neg x_k} = f[faux/x_k] \quad , \quad f_{x_k} = f[vrai/x_k]$$

On a $f = (\neg x_k \wedge f_{\neg x_k}) \vee (x_k \wedge f_{x_k})$. Cette décomposition commute avec la substitution et les opérations booléennes. L'*arbre de Shannon* de f (figure 5.1.(a)) est la décomposition successive de f selon les variables $x_1, x_2, \ldots, x_n$ et est unique pour un ordre fixé des variables. Les noeuds d'un arbre de Shannon sont étiquetés par les variables et les feuilles par des booléens indiquant la valeur de vérité de la formule évaluée sur le chemin correspondant. Pour un nœud étiqueté par x_k, le sous-arbre gauche correspond à x_k faux et le sous-arbre droit à x_k vrai. L'*arbre réduit de Shannon* est un arbre de Shannon dans lequel on a remplacé les noeuds dont les deux sous-arbres sont égaux par l'un des sous-arbres (figure 5.1.(b)). Le *graphe binaire de décision* de f est obtenu à partir de son arbre de Shannon réduit en partageant les sous-arbres identiques (figure 5.1.(c)). Les BDDs permettent donc une représentation canonique et compacte des fonctions booléennes, en partageant les chemins menant de la racine aux feuilles dans l'arbre de Shannon réduit.



(a) Arbre de Shannon

(b) Arbre de Shannon réduit

(c) BDD

(d) TDG

FIG. 5.1 – Représentations de $f = x_1 \wedge (x_2 \equiv x_3) \vee \neg x_1 \wedge \neg x_3$

Implémentation des opérations. — Grâce à des implémentations astucieuses, le partage *en mémoire* procuré par la représentation s'étend au partage des *calculs*.

En effet les opérations logiques s'implémentent par un parcours postfixé et en parallèle des paramètres ; elles construisent le résultat de bas en haut. Comme les implémentations veillent à ce que deux BDDs identiques soient toujours stockés à la même adresse physique et que le test d'égalité est donc trivial :

- le résultat des opérations est produit directement sous forme de BDD réduit ;
- il est possible d'utiliser des techniques de cache peu coûteuses pour mémoriser les résultats des opérations déjà calculées ; lorsque les caches sont de taille suffisante, le coût des opérations est alors fonction du nombre de *nœuds* plutôt que du nombre de *chemins* des BDDs arguments.

La plupart des opérations non logiques sur les BDDs peuvent également bénéficier de la technique des caches.

Le problème de l'ordre des variables. — La taille de la représentation d'une fonction dépend beaucoup de l'ordre des variables ; certaines fonctions de n variables ont une représentation de taille linéaire en n pour un ordre, et exponentiel pour un autre ; par exemple la fonction $\bigwedge_{1 \leqslant i \leqslant n}(x_{2i-1} \Leftrightarrow x_{2i})$ est représentable par un BDD dont la taille est en $O(n)$ pour l'ordre $x_1 < x_2 < \ldots < x_{2n}$ et en $O(2^n)$ pour l'ordre $x_1 < x_3 < \ldots < x_{2n-1} < x_2 < x_4 < \ldots < x_{2n}$. Le problème de trouver l'ordre optimal pour l' ensemble des fonctions présents dans la table de hachage, ou pour une fonction unique, est NP-complet [THY93, BW96]. Par ailleurs certaines fonctions ont une représentation exponentielle quel que soit l'ordre choisi [Bry91]. Cependant dans la pratique les BDDs sont une bonne représentation des fonctions rencontrées lors de l'analyse de systèmes finis et ont historiquement permis un saut quantitatif, voire qualitatif à ce niveau, dans la vérification de ces sytèmes. De plus les implémentations modernes ont des algorithmes heuristiques de *réordonnancement de variables* qui se révèlent efficaces, comme celle du "*sifting*" [Rud93] que nous utilisons dans notre outil.

Les TDGs. — L'opération de négation d'un BDD nécessite son parcours en profondeur. Pour éviter ce parcours, il est possible d'affecter un signe à chaque arc du graphe, indiquant si le graphe doit être interprété tel quel ou comme sa négation (figure 5.1.(d)). Une règle supplémentaire permet de maintenir la canonicité de cette représentation. Théoriquement, on peut obtenir jusqu'à un gain de 50% en espace en utilisant des TDGs à la place de BDDs, mais [MT98] rapporte qu'en pratique le gain en espace n'est que de l'ordre de 10%. En revanche, le fait de pouvoir complémenter une fonction en temps constant permet un gain de 50% en temps dans les opérations logiques, d'après la même référence.

5.1.2 Opérations sur les BDDs et TDGs

Opérations logiques. — Outre la panoplie des opérations logiques classiques : négation, disjonction, conjonction, si-alors-sinon (noté $ite(f, g, h)$) etc, on dispose également des quantifications *universelles* ($\forall$) et *existentielles* ($\exists$) de BDDs sur un ensemble de variables. Le *cofacteur* d'un BDD par une conjonction de littéraux (i.e. une variable ou sa négation) est défini par :

$$\begin{array}{lcl} f_x & = & f[vrai/x] \\ f_{\neg x} & = & f[faux/x] \\ f_{(l_1 \wedge \ldots \wedge l_n)} & = & (f_{l_1})_{(l_2 \wedge \ldots \wedge l_n)} \end{array}$$

Cofacteur généralisé. — Un *cofacteur généralisé* d'un BDD f par un BDD g est un BDD h tel que $g \Rightarrow (f \equiv h)$. L'objectif est que h soit plus petit que f. Toute fonction h vérifiant $f \wedge g \Rightarrow h \Rightarrow f \vee \neg g$ convient. La notion de cofacteur généralisé généralise celle de cofacteur d'un BDD par une conjonction de littéraux. Cette simplification d'un BDD f sachant qu'un BDD g est vérifié peut aussi être vue comme une opération de *spécialisation* ou d'*évaluation partielle* de f par g. En général le cofacteur généralisé d'un BDD par un autre n'est pas unique. Nous présenterons différents opérateurs possibles au §6.2.1.

5.2 Diagrammes de décision interprétés

Nous présentons ici les BDDs *interprétés*, dans lesquels certaines variables booléennes sont interprétées par la satisfaction de contraintes numériques. Nous appellerons ces variables *pseudo-variables*, pour les différencier des “vraies” variables booléennes. Nous étudions d'abord les diagrammes de décision purement numériques, c'est-à-dire qui ne contiennent que des pseudo-variables, avant de discuter des choix à effectuer lorsqu'on mélange variables booléennes et pseudo-variables. En général, nous adopterons la vision ensembliste des BDDs, dans laquelle ceux-ci sont vus comme des fonctions caractéristiques de sous-ensembles.

5.2.1 Diagrammes de décisions purement numériques

Dans le but de représenter des polyèdres généraux, qui sont des unions finies de polyèdres convexes, [Mau96] propose d'associer aux pseudo-variables des contraintes linéaires et appelle de tels diagrammes des DDCs (Diagrammes de Décision avec Contraintes). La figure ci-contre représente un DDC correspondant à $x \geqslant 10 \wedge y \geqslant 0 \;\vee\; x \leqslant 0$. Les DDCs sont utilisés dans l'outil d'exécution symbolique de programmes synchrones YOYO.

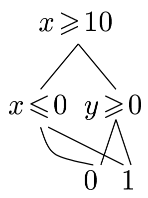


FIG. 5.2 – Un DDC

Afin de représenter des unions finies de zones, [MLAH99] associe quand-à-lui aux pseudo-variables des contraintes de différence de la forme $x_i - x_j \leqslant d$, et appelle de tels diagrammes DDDs (Difference Decision Diagramm). [BLP$^+$99] se distingue de cette première solution par l'utilisation de nœuds n-aires qui permettent d'effectuer des choix multiples sur la valeur d'une différence $x_i - x_j$, ce qui permet de réduire le nombre de nœuds mais sans doute aussi leur partage. Les diagrammes obtenus sont appelés CDDs (Clock Difference Diagrams) par leurs auteurs.

De manière générale, les représentations obtenues ne sont plus canoniques, comme le sont les BDDs :

- D'une part, parce qu'il n'existe généralement pas de décomposition unique d'ensembles numériques non convexes en union finie d'ensembles convexes ;
- D'autre part, parce que les pseudo-variables ne sont pas indépendantes entre elles et que les opérations logiques ne prennent pas en compte la sémantique des contraintes ; de ce fait leur résultat peut contenir des informations inutiles ou redondantes.

Ceci a pour conséquence que les tests d'inclusion ou d'égalité de tels diagrammes sont coûteux, et qu'il est nécessaire de simplifier fréquemment les diagrammes issus d'opérations de type BDD pour limiter leur taille.

Nous allons présenter les algorithmes et propriétés qui sont communs à ces représentations dans le contexte des DDCs, puisque ce sont les seuls diagrammes aptes à représenter les fonctions de transition de nos programmes.

Définition et sémantique des DDCs

La sémantique d'un DDC numérique représentant une partie de $\mathcal{N}^p$ est donnée par la fonction d'interprétation $\mathcal{S}$ suivante :

$$
\begin{array}{lcl}
\mathcal{S}(vrai) & = & \mathcal{N}^p \\
\mathcal{S}(faux) & = & \emptyset \\
\mathcal{S}(\Delta(c,G,D)) & = & \{\vec{x} \in \mathcal{S}(G) \mid \neg c(\vec{x})\} \cup \{\vec{x} \in \mathcal{S}(D) \mid c(\vec{x})\}
\end{array}
$$

(a) Chemin insatisfaisable (b) Contrainte redondante

FIG. 5.3 – BDDs simplifiables

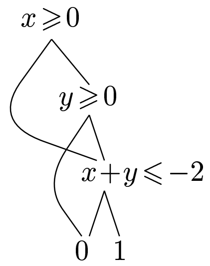


FIG. 5.4 – Dépendance du contexte dans la satisfaisabilité du nœud $x+y \leqslant -2$

Il est clair que $\mathcal{S}$ commute avec toutes les opérations logiques. Nous ne considérons pas de contraintes d'*égalités* dans les diagrammes car celles-ci ne séparent pas l'espace numérique en deux polyèdres convexes. Elles peuvent de toute façon être représentées par la conjonction de deux inégalités.

Outre le fait que, comme nous l'avons mentionné, la décomposition d'un polyèdre non convexe en union de polyèdres convexes n'est pas unique, deux autres phénomènes font que cette représentation n'est pas optimale : d'une part, les DDCs peuvent contenir des *chemins insatisfaisables*, figure 5.3(a) ; d'autre part des *contraintes redondantes*, figure 5.3(b). La simplification d'un DDC peut se faire en deux phases, qui traitent respectivement des deux phénomènes.

Simplification des DDCs

Suppression des chemins insatisfaisables. — Cette opération, outre son objectif de simplification du diagramme passé en paramètre, implémente aussi le test de vacuité.

La suppression des chemins insatisfaisables ne peut se faire sur des critères uniquement syntaxiques sur la forme des contraintes appartenant à un chemin. Il est clair qu'on peut se contenter de ce type de critère pour éliminer les contraintes parallèles incompatibles, comme c'est le cas pour le diagramme de la figure 5.3(a). Mais la détection de l'insatisfaisabilité du chemin $x \geqslant 0 \wedge y \geqslant 0 \wedge x + y \leqslant 2$ du diagramme de la figure 5.4 requiert une procédure de décision plus sophistiquée. Dans tous les cas, il est nécessaire de parcourir *tous les chemins* du BDD à simplifier, et non de parcourir seulement l'ensemble de ses nœuds comme pour les opérations logiques. En effet le fait qu'un nœud puisse être éliminé dépend de son contexte, et plus précisément des nœuds rencontrés depuis la racine, comme l'illustre la figure 5.4. Cela signifie que le partage opéré par la représentation ne s'étend pas au partage des calculs.

L'algorithme de suppression consiste à le considérer comme un arbre de Shannon réduit, à effectuer un parcours descendant de cet arbre en construisant les polyèdres attaché à chaque nœud de manière incrémentale. Dès que le polyèdre associé à un nœud est vide, celui-ci est supprimé, algorithme 5.1.

Le fait de construire incrémentalement les polyèdres est important, car d'une part, dès que le polyèdre associé à la racine d'un sous-arbre est vide, le sous-arbre entier est supprimé, et d'autre part l'ajout d'une contrainte à un polyèdre est incrémental. Mentionnons qu'il est possible que l'utilisation d'algorithmes de type programmation linéaire soit préférable à l'emploi des algorithmes sur les polyèdres décrits au §2.4.4 — à condition d'être incrémentaux —, car il ne s'agit ici que de tester si une conjonction de contraintes est satisfaisable, et non

Algorithme 5.1 : Suppression des chemins insatisfaisables

```
Input: un BDD f
Output: un BDD f' équivalent sans chemins insatisfaisables
  f' := Supp(f, ⊤)
  where Supp(f, p) ≡   { p polyèdre attaché au nœud courant }
    if f = vrai or f = faux then return f
    else
      let Δ(c, G, D) = f in
      pG := p ⊓ ¬c
      pD := p ⊓ c
      if pG = ⊥ then return Supp(D, pD)
      else if pD = ⊥ then return Supp(G, pG)
      else return Δ(c, Supp(G, pG), Supp(D, pD))
      end if
    end if
```

d'obtenir la représentation duale du polyèdre correspondant.

Comme dans le cas de la manipulation des polyèdres, si les contraintes peuvent être partitionnées en ensemble de contraintes indépendantes, i.e. portant sur des ensembles de variables disjointes, la région de l'espace associée est un produit cartésien, et la simplification peut être appliquée séparément sur chaque sous-espace, ce qui permet un gain important en temps, et en espace si l'ordre sur les contraintes prend également en compte ce fait.

La complexité de l'algorithme est fonction de la taille de l'arbre réduit de Shannon et du nombre de chemins insatisfaisables. Le pire des cas se rencontre lorsque l'arbre réduit de Shannon est un arbre complet et qu'aucun chemin n'est insatisfaisable, et sa complexité est en $O(2^n)$ avec n le nombre de contraintes. Cependant ce cas est peu susceptible de se produire et implique que les n contraintes soient orthogonales 2 à 2.

Élimination des contraintes redondantes. — [Mau96] propose un algorithme plus complexe pour détecter et enlever les contraintes redondantes, qui procède par un parcours ascendant et est donné en figure 5.2. Il repose sur le théorème suivant 2.

Théorème 2 *Soit* $\Delta(c, G, D)$ *un noeud de* BDD *numérique. Si* $\mathcal{S}(c \to (G \equiv D)) = universe$, *alors* $\Delta(c, G, D) = D$.

Preuve: Prenons $\vec{x} \in \mathcal{S}(\Delta(c, G, D))$ et montrons que $\vec{x} \in \mathcal{S}(D)$. Si $\neg c(\vec{x})$, la conclusion est immédiate. Si $c(\vec{x})$, alors $x \in \mathcal{S}(G)$. Par ailleurs, d'après l'hypothèse $x \in \mathcal{S}(G \equiv D)$. Donc $x \in \mathcal{S}(G) \cap \mathcal{S}(D) \subset \mathcal{S}(D)$. □

Cet algorithme est plus cher que le précédent, car il effectue un double parcours récursif : à chaque fois que l'on cherche à reconstruire un noeud, il faut effectuer le test $\mathcal{S}(c \wedge (G \equiv D)) = \emptyset$ qui nécessite l'emploi de l'algorithme précédent d'élimination des chemins insatisfaisables. En revanche il autorise l'emploi d'un cache au niveau de la fonction principale et des tests de vacuité.

Algorithme 5.2 : Suppression des contraintes redondantes

Input: un BDD f
Output: un BDD f' sans contraintes redondantes
  f' := $Elimine(f)$
  where $Elimine(f) \equiv$
    if $f = vrai$ or $f = faux$ **then return** f
    else
      let $\Delta(c, G, D) = f$ **in**
      G' := $Elimine(G)$
      D' := $Elimine(D)$
      if $\mathcal{S}(c \Rightarrow (G \Leftrightarrow D)) = \top$ **then return** G'
      else if $\mathcal{S}(\neg c \Rightarrow (G \Leftrightarrow D)) = \top$ **then return** D'
      else return $\Delta(c, G', D')$
      end if
    end if

Le cas des Ddds et des Cdds

Ces diagrammes se restreignent à la représentation d'union finie de zones, avec des contraintes beaucoup plus simples. Cela rend-il plus efficace les algorithmes de simplification ? En fait non, car hormis le fait que les procédures de décision (inclusion, vacuité) sont moins coûteuses, ils demeurent fondés sur le parcours de tous les chemins du diagramme à simplifier.

5.2.2 Diagrammes de décision mixtes et ordre des variables

Nous appelons *mixtes* les diagrammes mélangeant variables booléennes et pseudo-variables. Le principal problème qui se pose ici est de décider de l'ordre relatif des deux types de variables. Trois solutions sont possibles : mélanger les noeuds de manière quelconque, mettre les contraintes au dessus des variables booléennes, ou les mettre en dessous.

Mélanger les noeuds dans un ordre quelconque est une mauvaise idée : cela complique les algorithmes de simplifications et engendre un surcoût important. La simplification d'un BDD purement booléen va impliquer sa reconstruction, et on perd le bénéfice des techniques de cache dès que l'on se trouve sous une contrainte.

Placer les contraintes sous les variables booléennes est déjà plus simple ; on peut par exemple employer un cache externe pour les algorithmes de simplification, qui sera exploité lorsque deux chemins booléens distincts mènent au même diagramme numérique. Cependant les mêmes inconvénients, atténués, demeurent : il faut parcourir tout le niveau booléen avant de se mettre à simplifier, et la simplification se propage ensuite au niveau booléen.

Nous avons donc choisi de placer les contraintes au-dessus des variables booléennes. Le partage des contraintes dans les chemins se fait naturellement, puisqu'on les parcourt en premier. Dès qu'on arrive au niveau booléen, le seul changement à apporter aux algorithmes de simplification précédent est de remplacer les cas terminaux "BDDs constants" par 'BDDs booléens".

Le critère utilisé dans la discussion précédente est l'efficacité des algorithmes de simplification. Cependant, si le critère est la taille des diagrammes (en nombre de nœuds), il vaut quelquefois mieux mélanger variables et pseudo-variables. C'est le cas de la formule $\bigwedge_{1 \leqslant i \leqslant n}(b_i \Leftrightarrow x_i \leqslant 0)$, où les relations entre les deux types de nœuds sont étroites, de telle sorte qu'avec l'ordre $b_1 < \ldots < b_n < (x_1 \leqslant 0) < \ldots < (x_n \leqslant 0)$, la taille du BDD correspondant est en $O(2^n)$, alors qu'il est en $O(n)$ pour l'ordre $b_1 < (x_1 \leqslant 0) < \ldots < b_n < (x_n \leqslant 0)$.

5.3 Formules et fonctions de transitiion

Après avoir décrit les BDDs classiques et les BDDs interprétés, nous décrivons ici leur utilisation dans la représentation des formules et fonctions de transition. Les opérations spécifiques dont nous aurons besoin pour les analyses seront en revanche abordées au chapitre suivant.

5.3.1 Formules et fonctions booléennes

Les BDDs interprétés constituent une représentation exacte pour les formules et fonctions de transition booléennes, qui mélangent différents type de décision. Dans notre cas, les diagrammes utilisés sont de type DDCs, puisque les contraintes manipulées sont quelconques. L'utilisation de la mécanique des BDDs nous fournit automatiquement opérations logiques, substitutions, cofacteurs généralisés permettant la mise en œuvre de l'évaluation partielle, etc.

Les algorithmes de simplification décrits au §5.2.1 ne sont appliqué qu'une seule fois pour chaque fonction de transition, et donc leur coût n'est absolument pas un goulot d'étranglement dans notre outil.

5.3.2 Fonctions numériques

Les BDDs sont une représentation compacte d'arbres binaires dont les feuilles appartiennent à l'ensemble $\{0, 1\}$. Une extension classique est donc d'utiliser les principes des BDDs pour représenter des arbres binaires dont l'ensemble des feuilles E est quelconque, ce qui permet de représenter des fonctions de signature $\mathbb{B}^n \to E$, comme le montre la figure 5.5. La représentation obtenue est appelée MTBDD [CMZ+93] ou ADD (Algebraic Decision Diagram) [BFG+93], dans le cas où E est un ensemble numérique. Bien sûr, dans ce cas il n'y a plus de notion de signe, et cette généralisation ne s'applique qu'au "vrais" BDDs et non aux TDGs.

Nous utilisons donc des MTBDDs interprétés pour représenter nos fonctions numériques, en prenant pour E l'ensemble des expressions affines sur les variables numériques du programme.

EXEMPLE. — La figure 5.5 représente le MTBDD correspondant à l'expression suivante, pour l'ordre $b_1 < b_2 < (x \geqslant 10) < (x-y \geqslant 0)$:

```
if not b1 or not b2 then
  if x >= 10 then x+y else x
else
```

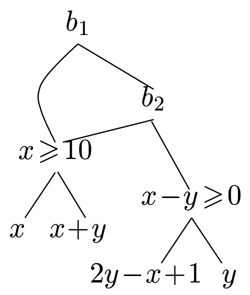


FIG. 5.5 – Un MTBDD interprété, avec E l'ensemble des expressions affines

```
if x-y >= 0 then y else 2y-x+1
```

□

La plupart des opérations non logiques sur les BDDs se transposent aux MTBDDs. On dispose ainsi du *si-alors-sinon*, noté $ite(f, g, h)$ avec f un BDD et g et h des MTBDDs, des opérations de cofacteur par un ensemble de variables et de cofacteur généralisé par un BDD. Les algorithmes de simplification des BDDs interprétés sont applicables directement aux MTBDDs interprétés.

5.4 Le domaine abstrait des BDDs mixtes

Une question naturelle se pose : pourquoi ne pas utiliser les diagrammes interprétés pour représenter l'espace d'état accessible ? En effet, ils permettent des calculs exacts, puisque les unions sont exactes (puisqu'explicites). Par conséquent, plus besoin non plus de structure de contrôle pour régler le degré des approximations effectuées. Mais se pose alors le problème de la terminaison (définition d'un opérateur d'élargissement), et aussi de l'implémentation des opérateurs de post- et pré- condition.

Annonçons d'emblée que l'absence d'approximation induit par un tel domaine abstrait est aussi son principal défaut, puisque notre objectif est au contraire d'effectuer le maximum d'approximations compatibles avec la preuve de la propriété. Nous allons cependant discuter de cette solution de manière plus approfondie.

[MLAH99] fait ce choix en utilisant les DDDs, mais comme ils analysent des automates temporisés, le problème est décidable et la terminaison garantie. Il est encore trop tôt pour comparer l'efficacité de cette solution aux techniques plus classiques qui procèdent par exploration ou construction du graphe de contrôle discret. [BLP$^+$99] utilise pour le moment les CDDs uniquement pour mémoriser les états explorés ; cela signifie qu'ils n'appliquent aucun transformateur de prédicats, et que les opérations utilisées sont, outre les simplifications, l'union et le test d'inclusion. [Mau96] fait également ce choix dans notre cadre d'application (contraintes linéaires quelconques), pour lequel la terminaison n'est pas garantie. Étant donné leur objectif, qui est d'exécuter un programme de manière symbolique [CG99], l'absence de garantie de terminaison n'est pas rédhibitoire.

Complexité des transformateurs de prédicats. — Dans tous les cas, le calcul de la postcondition d'un diagramme par une fonction de transition est une opération complexe. En effet en reprenant les notations du §3.3.3 il consiste à calculer l'expression suivante :

$$\Big(post(\tau)(X)\Big)(\vec{b'}, \vec{x'}) =$$
$$\exists(\vec{b}, \vec{\beta}, \vec{x}, \vec{\xi}) \left(X(\vec{b}, \vec{x}) \wedge assert(\vec{b}, \vec{\beta}, \vec{x}, \vec{\xi}) \wedge \bigwedge_{1 \leqslant i \leqslant m} b' = \phi_i(\vec{b}, \vec{\beta}, \vec{x}, \vec{\xi}) \wedge \bigwedge_{1 \leqslant k \leqslant p} x'_k = \psi_k(\vec{b}, \vec{\beta}, \vec{x}, \vec{\xi}) \right)$$

ce qui implique introduction de nouvelles contraintes, opérations logiques, quantifications booléennes et surtout numériques, puis simplifications — coûteuses mais nécessaires, sans quoi l'itération de l'opération se heurte à une explosion de la représentation. Bien sûr, la restriction à des contraintes de différence diminue grandement la complexité de l'opération dans [MLAH99], si on la compare à l'implémentation de [Mau96].

Conception d'un élargissement. — Dans notre cadre d'application, il faudrait de plus définir un opérateur d'élargissement pour garantir la convergence en temps fini, ce qui paraît délicat. En effet, élargir sur des unions de polyèdres semble déjà difficile ; en effet l'élargissement standard sur les polyèdres convexes repose sur l'intuition que lorsqu'une contrainte est translatée à chaque itération, elle file vers l'infini et qu'on peut donc la supprimer. Cette intuition ne s'étend guère aux unions de polyèdres. [BGP99] propose certes un tel élargissement, défini comme suit :

$$\Big(\bigcup_{1\leqslant i\leqslant n} P_i\Big)\nabla\Big(\bigcup_{1\leqslant j\leqslant m} Q_i\Big) = \bigcup_{\{(i,j)\ \mid\ P_i\subseteq Q_j\}} (P_i\nabla Q_j)\ \cup \bigcup_{\{j\ \mid\ \forall i\ P_i\not\subseteq Q_j\}} Q_j$$

La complexité de l'opérateur vient du test $\forall i,j\ P_i \subseteq Q_j$, qui est quadratique en le nombre de polyèdres impliqués dans l'union explicite. De plus, le résultat est une union de taille quadratique. L'article propose à ce sujet d'appliquer des enveloppes convexes lorsque la taille dépasse un certain seuil. Enfin et surtout, l'emploi de cet opérateur *ne garantit pas* la convergence de l'itération de point fixe. En ce sens il ne s'agit pas d'un opérateur d'élargissement au sens de [CC77].

L'extension de cet élargissement compliqué à des diagrammes mixtes n'est donc absolument pas immédiate.

Une solution plus raisonnable, si l'on veut utiliser de tels diagrammes, serait par exemple de placer les contraintes sous les booléens, et de calculer, après chaque calcul de postcondition, l'enveloppe convexe du DDC purement numérique associé à chaque chemin booléen dans le DDC. Ceci rendrait la conception d'un élargissement sur les DDCs mixtes plus simple et l'opérateur obtenu serait plus économique. Mais il ne serait alors plus possible d'associer une union explicite de polyèdres à une valuation booléenne, or nous avons vu que cela pouvait être nécessaire avec l'exemple de la page 32 du chapitre précédent.

Nous venons de discuter de différents aspects qu'impliquerait l'utilisation des DDCs comme domaine abstrait de prédicats ; nous avons en particulier évoqué les difficultés techniques à résoudre pour garantir la convergence, et le coût du calcul de postconditions. En fait, le principal défaut d'un tel domaine abstrait est qu'il n'engendre *aucune approximation* — à part celles induites par un éventuel opérateur d'élargissement, et c'est pour cette raison que le problème de la structure de contrôle ne se pose plus. Le faible degré d'approximation obtenu a la contrepartie automatique d'une complexité élevée. De plus le coût de la preuve d'une propriété triviale serait identique à celui d'une propriété plus fine. Or notre objectif est d'adapter le degré de précision de l'analyse, et donc aussi son coût, en fonction de la propriété à montrer, dans l'espoir que le degré de précision exigé n'est pas trop grand et surtout qu'il n'est pas uniforme sur tout l'espace d'état du programme.

Le paragraphe suivant décrit donc le domaine abstrait que nous avons choisi, beaucoup plus simple mais permettant, en combinaison avec la structure de contrôle, de choisir la précision des analyses.

5.5 Le domaine abstrait des *états convexes*

Nous décrivons dans ce paragraphe la construction du domaine abstrait mis en œuvre dans le partionnement dynamique, qui permet de combiner le treillis booléen avec un quelconque treillis numérique.

5.5.1 Définition et représentation

Soit $S = \mathbb{B}^n \times \mathcal{N}^p$ l'espace d'état du programme à vérifier et $C = \mathcal{P}(S)$ le domaine concret correspondant. On suppose que l'on dispose d'un treillis abstrait $L_{\mathcal{N}}$ relié par une connexion de Galois $(\alpha_{\mathcal{N}}, \gamma_{\mathcal{N}})$ au treillis concret $\mathcal{P}(\mathcal{N}^p)$. $L_{\mathcal{N}}$ peut en particulier être un des treillis décrit au §2.4.

Nous proposons de prendre comme treillis abstrait $A = \mathcal{P}(\mathbb{B}^n) \times L_{\mathcal{N}}$, dont les éléments forment une famille de Moore de C. Une valeur abstraite (B, P) est donc le produit cartésien d'un sous-ensemble de l'espace d'état booléen, et d'une valeur abstraite numérique. Nous appellerons ces valeurs abstraites *états convexes*.

Afin que chaque valeur concrète ait une représentation unique, i.e. que γ soit injective, il faut quotienter A par la relation d'équivalence $\mathcal{R}(x, y) \Leftrightarrow \gamma(x) = \gamma(y)$. Il suffit ici de rendre équivalentes les valeurs abstraites $\{(\emptyset, P) \mid P \in \mathcal{P}^{oly}(\mathcal{N}^p)\} \cup \{(B, \emptyset) \mid B \in \mathbb{B}^n\}$ et de les représenter par une valeur distinguée $\perp$.

Un élément sera représenté, pour la partie numérique, par la représentation du treillis $L_{\mathcal{N}}$ sous-jacent, et pour la partie booléenne, par un BDD (§5.1). La terminologie utilisée (états convexes) s'inspire du fait que nous avons choisi de prendre pour $L_{\mathcal{N}}$ le treillis des polyèdres convexes $\mathcal{P}^{oly}(\mathcal{N}^p)$.

5.5.2 Opérations de base

Nous décrivons l'implémentation des opérations de base sur le treillis que nous venons de définir.

Test d'inclusion et d'égalité. — Le test d'inclusion se fait membre-à-membre, en utilisant les tests d'inclusion disponibles pour les treillis sous-jacents :

$$(B_1, P_1) \sqsubseteq (B_2, P_2) \Longleftrightarrow (B_1 \Rightarrow B_2) \wedge (P_1 \sqsubseteq_{\mathcal{N}} P_2)$$

Le test d'égalité procède de la même manière.

Intersection et *union convexe*. — La borne inférieure dans A se calcule terme à terme et correspond à l'intersection ensembliste :

$$\perp \sqcap (B, P) = (B, P) \sqcap \perp = \perp$$
$$(B_1, P_1) \sqcap (B_2, P_2) = (B_1 \wedge B_2, P_1 \sqcap_{\mathcal{N}} P_2)$$

La borne supérieure, que nous appellerons *union convexe*, est l'abstraction de l'union ensembliste et se calcule ainsi :

$$\perp \sqcup (B, P) = (B, P) \sqcup \perp = (B, P)$$
$$(B_1, P_1) \sqcup (B_2, P_2) = (B_1 \vee B_2, P_1 \sqcup_{\mathcal{N}} P_2)$$

Il est à noter que cette opération ne perd pas de l'information seulement à cause de l'utilisation de la borne supérieure sur la partie numérique, mais aussi parce que parties booléennes et numériques sont considérées séparément : par exemple en prenant le treillis des polyèdres pour $L_{\mathcal{N}}$ (le plus précis que nous connaissions), $(b, x > 0) \sqcup (\neg b, x \leqslant 0) = (\top, \top) = \top$ au lieu du résultat exact qui serait $(b \equiv (x > 0))$.

Une autre remarque est qu'il n'y a pas distributivité de l'intersection sur l'union convexe et vice-versa :

$$\begin{aligned} (X \sqcap Y_1) \sqcup (X \sqcap Y_2) &\sqsubseteq X \sqcap (Y_1 \sqcup Y_2) \\ X \sqcup (Y_1 \sqcap Y_2) &\sqsubseteq (X \sqcup Y_1) \sqcap (X \sqcup Y_2) \end{aligned}$$

Élargissement standard. — Dans le treillis A, la partie booléenne est de profondeur finie, il n'y a donc pas nécessité d'utiliser un élargissement sur les booléens. L'opérateur d'*élargissement standard* est donc défini comme suit :

$$(B_1, P_1)\nabla(B_2, P_2) = (B_1 \vee B_2, P_1 \nabla_{\mathcal{N}} P_2)$$

où $\nabla_{\mathcal{N}}$ est l'élargissement standard sur $L_{\mathcal{N}}$. $\mathbb{B}^n$ étant de profondeur finie 2^n, et $\nabla_{\mathcal{N}}$ ayant par hypothèse les propriétés d'un élargissement, $\nabla : A \times A \to A$ est un élargissement.

Étant donné que la profondeur du treillis booléen est grande, on pourrait envisager d'élargir également sur les booléens. Cependant un tel élargissement sur un espace fini est difficilement définissable d'une manière intuitive : on ne sait pas bien capter la régularité sur un treillis booléen. Les propositions existantes sont fondées sur une réduction de la taille des BDDs. [Mau98] propose un opérateur de surapproximation, qui prend en entrée un BDD f et un nombre de nœuds n, et retourne un BDD g de taille inférieure à n tel que $f \Rightarrow g$. Dans le même esprit la bibliothèque de BDD CUDD [Som] propose entre autre un opérateur de conjonction sous-approximé, dont on déduit un opérateur de disjonction sur-approximé. Nous n'avons pas testé ces opérateurs, mais il est possible qu'ils soient fort utiles lorsque les BDDs manipulés deviennent trop gros.

Transformation par une fonction de transition. — Le calcul de la postcondition ou de la précondition d'une valeur abstraite par une fonction de transition mélangeant expressions booléennes et numériques (c.f. paragraphe 3.3.3) est une opération complexe, qui fait l'objet du chapitre 7. L'algorithme implantant l'abstraction standard (la plus précise) n'est déjà pas si simple, et de plus il apparaît plus judicieux d'utiliser un opérateur approché, comme nous l'avons mentionné au §4.4.6.

5.5.3 Lien avec la structure de contrôle

Le treillis qui vient d'être décrit engendre des approximations très importantes, puisque propriétés booléennes et numériques ne sont pas reliées : il est en effet relationnel uniquement entre variables booléennes ou entre variables numériques. Le rôle de combinaison de propriétés va être dévolu à la structure de contrôle. Reprenons par exemple la formule $(b \Leftrightarrow (x > 0))$. Son abstraction est $\top$, mais si on utilise une structure de contrôle à deux places, l'une définie par l'état convexe $(b, \top)$ et l'autre par $(\neg b, \top)$, alors on peut représenter cette valeur abstraite sans perte d'information par $(b, x > 0) \cup (\neg, x \leqslant 0)$. On remarque que la partition $(vrai, x > 0), (vrai, x \leqslant 0)$ le permet aussi. Cela signifie qu'étant donné un programme et une propriété sur celui-ci, des partitions différentes de l'espace d'état peuvent mener à des structures de contrôle et à des résultats semblables.

Chapitre 6

Manipulation de formules et de valeurs abstraites

Le chapitre précédent a décrit la représentation des fonctions et le choix du domaine abstrait. Ce chapitre décrit maintenant les opérations spécifiques sur ces objets, qui seront utilisées dans les calculs de post- et pré- condition, les analyses, et les heuristiques de raffinement (chapitres 7, 8 et 9).

NOUS disposons de deux représentations pour les sous-ensembles $S = \mathbb{B}^n \times \mathcal{N}^p$: les BDDs mixtes et les états convexes. Nous discutons tout d'abord au §6.1 des relations entre ces deux représentations.

Le §6.2 présente les algorithmes d'évaluation partielle de fonctions. Nous verrons qu'outre leur utilisation pour simplifier les fonctions de transition en fonction de leur domaine d'application, certains des opérateurs présentés permettent d'effectuer une simplification partielle des fonctions (au sens du chapitre précédent) sans utiliser d'opérations polyédriques. Nous présentons aussi les opérations de quantifications de formules sur des variables booléennes et numériques. Typiquement, elles seront utilisées pour éliminer les variables d'entrées d'une formule.

Le §6.3 conclut en discutant du compromis à effectuer entre opérations exactes prenant en compte la sémantique des contraintes, et impliquant des calculs polyédriques, ou leur approximation obtenue par des calculs purement booléens.

6.1 Formules et valeurs abstraites

Nous disposons de deux représentations pour les sous-ensembles de $Q = \mathbb{B}^n \times \mathcal{N}^p$: les BDDs mixtes et les états convexes. Suivant les cas et les besoins, nous utiliserons l'une ou l'autre. Quand on veut mémoriser une information de manière exacte, nous manipulerons des formules, quand au contraire les approximations sont nécessaires, comme c'est le cas pour les places de la structure de contrôle, nous emploierons des états convexes.

De manière générale la possibilité offerte par les formules de représenter des ensembles non convexes, et leur représentation par des BDDs, rendent leur emploi plus aisé, bien que se posent alors les problèmes de non canonicité.

6.1.1 Contraintes apparaissant dans les formules

Les formules et fonctions que nous manipulerons sont initialement générées à partir du source du programme. Ceci implique que les contraintes qui apparaissent dans leur support apparaissent aussi dans le programme. Au cours des calculs de post- et pré- condition et des analyses (chapitres 7 et 8), de nouvelles formules seront générées, mais porteront toujours sur cet ensemble fini de contraintes, *à une exception près* : l'élimination des variables d'entrée peut introduire de nouvelles contraintes. Par exemple $\exists\xi\ (x \leqslant \xi \wedge \xi \leqslant y) = x \leqslant y$. Tous les diagrammes que nous manipulerons vérifient donc la propriété suivante :

> *Toute contrainte apparaissant dans le support d'un* BDD *ou* MTBDD *est soit une contrainte du programme, soit une contrainte résultant de l'élimination des variables d'entrées d'une conjonction de contraintes du programme.*

Ce choix présente l'avantage de limiter le nombre de contraintes apparaissant dans les diagrammes, et de pouvoir utiliser des caches avec un certain succès.

Une autre conséquence est que l'expressivité des deux représentations est incomparable :

1. d'une part les formules permettent de représenter des ensembles non convexes, comme $x \geqslant 0 \vee y \geqslant 0$;
2. d'autre part, le polyèdre défini par $\{x + 3y + 4z \geqslant 23\}$ ne peut pas être représenté de manière exacte par une formule, si la contrainte le définissant n'apparaît pas dans le programme.

Par ailleurs, comme nous l'avons déjà mentionné au chapitre précédent, nous ne manipulons que des contraintes d'*inégalité*, car la négation d'une contrainte d'*égalité* n'est pas convexe. L'égalité est toujours représentable par la conjonction de deux inégalités.

6.1.2 Formules convexes et non-convexes

Dans ce paragraphe, nous supposons que les formules sont ordonnées par la relation d'implication logique $\Rightarrow$, mais sans prendre en compte l'interprétatiobn des pseudo-variables par des inégalités numériques : on n'a par exemple pas $(x \geqslant 0) \Rightarrow (x \geqslant 1)$.

Avant de présenter les notions de formules convexes et non-convexes, nous avonsaussi besoin de définir l'opérateur *mon*. Pour une formule booléenne f, $mon(f)$ est le plus petit monôme m tel que $f \Rightarrow m$, c'est-à-dire que $mon(f)$ est la conjonction des variables qui doivent être soit toujours vraies, soit toujours fausses, pour que la formule f soit satisfaite.

Formules convexes. — Une formule f sera dite *convexe* si et seulement si

$$f = f^m \wedge f^b$$

avec f^m un monôme sur des inégalités et f^b une formule purement booléenne.

Une formule convexe peut donc être convertie en une unique valeur abstraite (f^b, P) avec P le polyèdre formé des contraintes de f^m. Notons que la définition repose sur un critère syntaxique : la formule $f = ite(x \geqslant 0, x \leqslant 5, x \geqslant -5)$[1] sera considérée comme non convexe, même si en fait elle correspond à la valeur abstraite $(vrai, \{-5 \leqslant x \leqslant 5\})$.

[1] Rappelons que l'opération $ite(f, g, h) = f \wedge g \vee \neg f \wedge h$ est l'opération si-alors-sinon, §5.1.2

Factorisation de la partie convexe d'une formule. — Lorsque une fonction f est non convexe, on peut la décomposer en la conjonction d'une formule convexe et d'un "reste" comme suit :

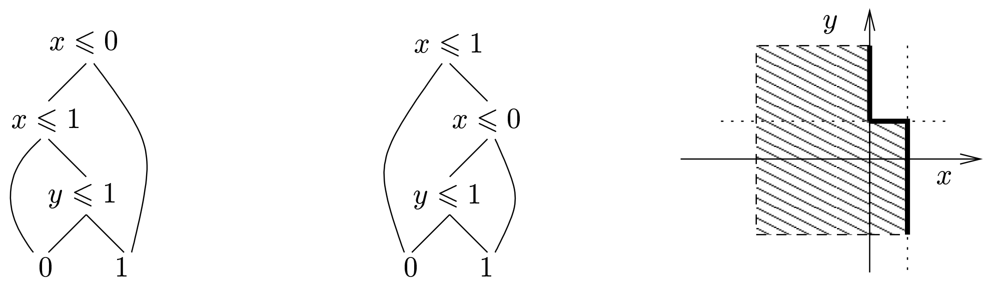
$$f = (f^m \wedge f^b) \wedge f^{nc}$$

avec, en notant $\vec{b}$ l'ensemble des variables booléennes et $\mathcal{C}$ l'ensemble des contraintes apparaissant dans f :

- $f^m = \exists\vec{b}\ mon(f)$ le plus petit monôme portant exclusivement sur des contraintes tel que $f \Rightarrow f^m$;
- $f^b = \exists\mathcal{C}\ f$ la plus petite formule purement booléenne telle que $f \Rightarrow f^b$;
- $f^{nc} = f \star (f^m \wedge f^b)$ la partie non convexe de f, où $\star$ est un opérateur de cofacteur généralisé (§5.1.2).

Par exemple, pour $f = (x > y) \wedge \Big(\big(b \wedge (x > 0)\big) \vee c\Big)$ et $\star$ un des quelconques opérateurs de cofacteur généralisé décrit au paragraphe suivant, on obtient $f^b = b \vee c$, $f^m = (x > y)$ et $f^{nc} = (c \vee (x > 0))$.

Idéalement on souhaiterait que f^m représente sous forme de formule le plus petit polyèdre convexe contenant f et *formé à partir des contraintes apparaissant dans* f. Cependant ce n'est pas forcément le cas, car l'opération *mon* ne tient pas compte de la sémantique des contraintes, et suivant l'ordre on n'obtient pas le même résultat, comme le montre la figure 6.1, même lorsque f est sous forme simplifiée.



(a) Pour l'ordre $(x \leqslant 0) < (x \leqslant 1)$, $f^m = vrai$

(b) Pour l'ordre $(x \leqslant 1) < (x \leqslant 0)$, $f^m = (x \leqslant 1)$

(c) Région du plan représentée par f

FIG. 6.1 – Factorisation du BDD numérique $f = (x \leqslant 0) \vee (x \leqslant 1 \wedge y \leqslant 1)$

6.1.3 Conversion de formules en états convexes

L'ensemble des états initiaux et la propriété sont donnés sous forme de formules. Il est clair que lors de la construction de la structure de contrôle initiale, il faudra convertir ces formules en états convexes.

Décomposition d'une formule en union exacte d'états convexes. — Soit f un BDD mixte, et supposons que nous voulions calculer une union exacte de valeurs abstraites telle que $\mathcal{S}(f) = \bigcup_i (B_i, P_i)$. La solution simple consiste à parcourir tous les chemins de f partant de la racine et menant à un sous-BDD booléen, et pour chacun de ces chemins à construire l'état convexe à partir du sous-BDD booléen et du polyèdre associé au chemin numérique.

Une meilleure solution consiste à exploiter le fait que l'ajout d'une contrainte à un polyèdre est incrémental, en prenant en compte dès que possible les contraintes communes à plusieurs polyèdres. Ceci se fait en factorisant de manière récursive f par la formule $\exists\vec{b}\ mon(f)$. L'algorithme correspondant est l'algorithme 6.1.

Algorithme 6.1 Conversion d'une formule en union exacte d'états convexes

```
Input: un BDD mixte f
Output: une union d'états convexes vérifiant ⋃_i(B_i, P_i) = S(f)
begin
  return decomp(⊤,f)
end
where decomp(P,f) ≡
  { P polyèdre en cours de construction, f BDD restant à parcourir }
  let m = ∃b⃗ mon(f) in
  P := P ⊓ S(m); f := f_m;
  let Δ(var,G,D) = f in
  if var est une contrainte then
    return decomp(P ⊓ S(var), f_var) ∪ decomp(P ⊓ S(¬var), f_¬var)
  else
    return (f,P)
  end if
```

Approximation d'une formule par un état convexe. — Une première solution, exacte, consiste à construire le plus petit état convexe contenant tous les états satisfaisant une formule f. Pour cela, il suffit d'utiliser l'algorithme précédent, mais en remplaçant les unions explicites du résultat par des bornes supérieures.

Si l'on veut éviter des calculs polyédriques, une seconde solution, approximée supérieurement, est d'extraire de f sa sous-formule convexe, qui sera alors une approximation du résultat exact.

6.1.4 Conversion approchée d'un état convexe en formule

Le problème inverse au précédent se posera au chapitre 9 décrivant les heuristiques de raffinement : on connaît un état convexe (B, P), et on veut construire une formule f l'approchant au mieux.

Le problème est que les contraintes de P ne sont pas forcément toutes des contraintes apparaissant dans le programme. La solution que nous utilisons consiste à approcher P par un monôme $m = \bigwedge_i c_i \ \wedge \ \bigwedge_j \neg c_j$ sur des contraintes du programme, tel que

- $\forall i\ P \models c_i$
- $\forall j\ P \models \neg c_j$

Bien sûr, m peut contenir des contraintes redondantes, qu'il convient éventuellement éliminer. Cette construction est linéaire en le nombre de contraintes apparaissant dans le programme.

La solution à notre problème est donc d'approcher (B, P) par la formule $f = m \wedge B$.

6.2 Manipulation de formules et de fonctions

Nous abordons au §6.2.1 les opérateurs d'évaluation partielle de ces fonctions par une formule booléenne, opérateurs qui permettront la spécialisation des fonctions de transition évoquée aux §4.4.5 et §4.4.6, et qui seront également utilisé pour notre algorithme de raffinement du chapitre 9. Nous expliquons ensuite au §6.2.2 comment ces opérateurs permettent d'améliorer l'élimination des chemins insatisfaisables ou des contraintes redondantes dans les diagrammes. Le §6.2.3 présente l'élimination des variables d'entrées d'une formule, opération qui sera utilisée par l'algorithme de raffinement du chapitre 9. Enfin, nous évoquons l'évaluation partielle de fonctions par un polyèdre au §6.2.4.

6.2.1 Cofacteurs généralisés : évaluation partielle de fonctions par une formule

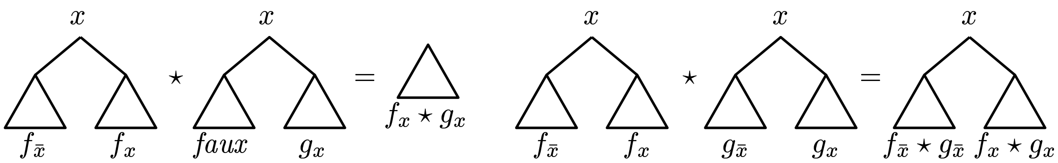
Nous avons mentionné au §4.4.5 que nous exploiterions la structure de contrôle pour simplifier les fonctions de transitions sur chacune de ses places et améliorer les calculs de transformation de prédicats. Supposons que l'on veuille spécialiser les fonctions de transition sur une place, et que sur chaque place nous disposions d'une *assertion de place*, qui est une formule spécifiant les valeurs admissibles des variables d'état et d'entrée sur cette place, c.f. §7.4. Cette spécialisation s'implémente dans le cadre des diagrammes par le calcul d'un *cofacteur généralisé*, défini au §5.1.2 dans le cadre des BDDs. Rappelons qu'un opérateur de cofacteur généralisé $\star$ calcule une formule $h = f \star g$ telle que $h \wedge g = f \wedge g$, i.e., f et h coincident quand g est vérifié. Généralement, on souhaite que h ait un BDD plus compact ou un support plus petit. Nous allons successivement présenter différentes opérations de cofacteur et examiner leurs propriétés.

Il est important de disposer d'un opérateur efficace, surtout dans le cas des BDDs et MTBDDs mixtes, car les décisions sur les contraintes numériques sont chères et il faut donc minimiser leur nombre dans les diagrammes.

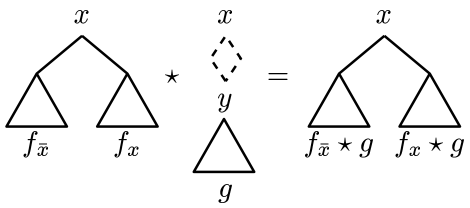
Les opérateurs "constraint" et "restrict"

Ces opérateurs, introduits par [CBM89b], s'implémentent par un parcours descendant et en parallèle des arguments. On pose $h = f \star g$. On opère cas par cas :

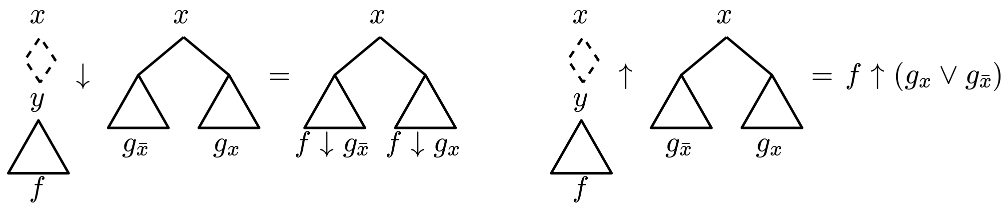
1. lorsque f et g ont une racine commune x, si l'un des fils de g est *faux*, on peut remplacer le fils correspondant de f par n'importe quoi ; choisir l'autre branche de f permet de supprimer le nœud x ; sinon on descend récursivement ;



2. lorsque la racine de f est au-dessus de celle de g, on simplifie les fils de f par g :



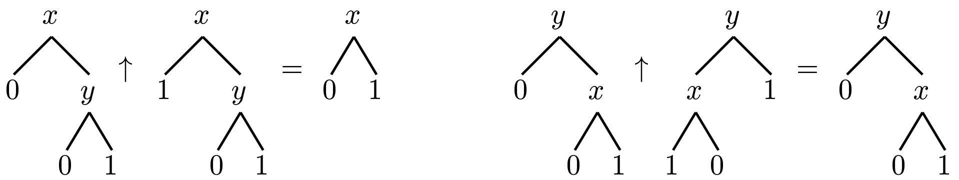
3. lorsque la racine de f est en-dessous de celle de g, deux choix sont possibles, qui distinguent les deux opérateurs :
   (a) Constraint ($\downarrow$) : on accepte de créer un nœud qui n'existait pas dans f ;
   (b) Restrict ($\uparrow$) : on ne crée pas de nœud.



Notons que lorsque g est un monôme et contient des variables non présentes dans f, $f \downarrow g \neq f_g$, alors que Restrict est toujours équivalent au cofacteur de f par les littéraux de g.

L'intérêt de Constrain est qu'il possède des propriétés particulières permettant le calcul efficace d'image d'ensemble booléen par une fonction de transition, qui sera exploité au chapitre 8, mais son défaut est que le résultat peut contenir plus de nœuds que le premier argument.

L'opérateur Restrict convient mieux à la simplification des fonctions de transitions par une formule (ici l'assertion). Cependant, le résultat des deux opérateurs dépend de l'ordre utilisé pour les variables, comme l'illustre l'exemple suivant.



(a) Ordre $x < y$ (b) Ordre $y < x$

FIG. 6.2 – Opérateur Restrict et ordre des variables

EXEMPLE. — Prenons $f = (x \wedge y)$ et $g = (x \Rightarrow y)$. Comme le montre la figure 6.2, pour l'ordre $x < y$, on obtient $f \uparrow g = x$, qui est le résultat souhaité, mais pour l'ordre $y < x$, $f \uparrow g = f$. □

La synthèse de bas en haut effectuée par Constrain et Restrict privilégie en effet la simplification des nœuds les plus profonds, et non du nœud étiqueté par y dans la figure 6.2(b).

Deux opérateurs plus puissants

[Ray91] propose un opérateur de cofacteur plus sophistiqué, du type "constraint" et que l'on notera $\downarrow^*$ qui procède par un parcours descendant et qui traite correctement l'exemple ci-dessus quel que soit l'ordre des variables.

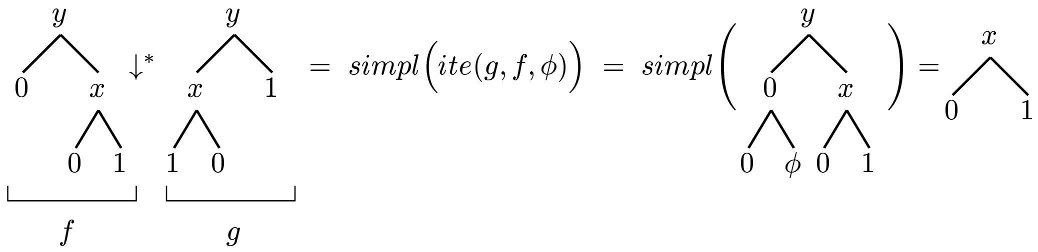


FIG. 6.3 – Opérateur de cofacteur amélioré

Au lieu d'éliminer un nœud localement comme les deux opérateurs précédent, lorsqu'une branche dans le premier diagramme correspond à une branche fausse dans le second BDD, on introduit une feuille distinguée ϕ, qui représente une fonction quelconque. Ceci se fait en construisant $ite(g, f, \phi)$, figure 6.3. Ensuite, lors du parcours descendant du résultat, pour chaque nœud on va essayer d'unifier ses deux fils en remplaçant les ϕ par des fonctions plus déterminées, mais qui peuvent encore contenir des feuilles ϕ. La définition de la fonction *simpl* est la suivante :

1. $simpl(e) = e$ pour $e \in E$
2. $simpl(\Delta(x, G, D)) =$ **let** $U = unifie(G, D)$ **in**
   if $U \neq \emptyset$ **then** $simpl(U)$ **else** $\Delta(x, simpl(G), simpl(D))$

avec la fonction symétrique $unifie$ définie par

1. $unifie(f, f) = unifie(f, \phi) = f$
2. $unifie(e, e') = \emptyset$ pour $e, e' \in E$ et $e \neq e'$
3. $unifie(\Delta(x, G, D), e) = \Delta(x, unifie(G, e), unifie(D, e))$ pour $e \in E$
4. $unifie(\Delta(x, G_1, D_1), \Delta(y, G_2, D_2)) = \Delta(x, unifie(G_1, f_2), unifie(D_1, f_2))$ pour $x < y$ et avec $f_2 = \Delta(y, G_2, D_2)$
5. $unifie(\Delta(x, G_1, D_1), \Delta(x, G_2, D_2)) = \Delta(x, unifie(G_1, G_2), unifie(D_1, D_2))$

$\emptyset$ représente l'échec et est un élément absorbant pour $unifie$ et Δ.

Dans le cas où l'on cherche le cofacteur d'un BDD (par opposition aux MTBDDs), il n'est pas nécessaire de manipuler explicitement des feuilles ϕ : on peut procéder uniquement au moyen d'opérations logiques et donc utiliser les TDGs. En effet on peut représenter les ϕ-BDDs par un encadrement de deux formules, en partant de l'encadrement $[(f \wedge g, f \vee \neg g]$.

Dans les deux cas, si l'on veut que la sémantique des contraintes soit prise en compte, par exemple pour obtenir $x \geqslant 0 \downarrow^* x \geqslant 2 = vrai$, il faut simplifier $ite(g, f, \phi)$.

Notons que comme $\downarrow^*$ est de type "constraint", le support de $f \downarrow^* g$ peut contenir des variables n'apparaissant pas dans f et introduite lors de la construction de la formule $ite(g, f, \phi)$. Pour éviter ce phénomène, il faut utiliser une variante "restrict" de $\downarrow^*$, naturellement notée $\uparrow^*$, et qui consiste à quantifier existentiellement g par les variables de son support n'apparaissant pas dans f, avant de construire $ite(g', f, \phi)$.

Ces deux opérateurs privilégient la réduction du support de la fonction, c'est-à-dire l'ensemble des variables qui apparaissent dans son BDD. Typiquement, si l'on arrive à éliminer la racine d'un BDD, la variable correspondante disparaît du support. Avec un algorithme de bas en haut, l'élimination totale d'une variable est beaucoup plus incertain. Or pour les fonctions de transition, la réduction du support est essentielle, puisqu'elle entraîne une diminution du nombre de décisions.

Ces opérateurs sont en revanche plus chers que ceux décrits précédemment dans la mesure où ils procèdent à un double parcours récursif.

6.2.2 Cofacteurs généralisés et simplification des BDDs interprétés

Les opérateurs de cofacteur généralisé permettent d'effectuer une simplification partielle des BDDs interprétés. En effet, on peut mémoriser dans une formule *insat* la liste des monômes sur des contraintes détectés comme insatisfaisables. Une telle formule représente un "*don't care set*" et le cofacteur généralisé d'une fonction par $\neg insat$ permet de supprimer des chemins insatisfaisables ou des nœuds redondants.

Naturellement, ceci ne garantit pas que tous les chemins du cofacteur calculé soient satisfaisables, puisque *insat* ne correspond pas d'ordinaire à l'union de tous les chemins insatisfaisables formés à partir des contraintes apparaissant dans le programme, mais cette opération peut être utilisée comme un prétraitement ou une simplification partielle. En particulier il est facile de l'utiliser pour traiter un cas simple et fréquent d'incompatibilité : les conjonctions de contraintes parallèles incompatibles. Pour cela, on peut partitionner les contraintes en une famille $((c_{i,j})_{0 \leqslant j \leqslant J_i})_i$, dans laquelle les partitions $(c_{i,j})_{0 \leqslant j \leqslant J_i}$ sont formées de contraintes parallèles. Si celles-ci sont triées de manière appropriée, il suffit d'initialiser *insat* par la formule :

$$\bigwedge_i \left(\bigwedge_{0 \leqslant j \leqslant J_i - 1} (c_{i,j} \Rightarrow c_{i,j+1}) \right)$$

Pour minimiser la taille du BDD *insat* et favoriser l'efficacité des opérations de cofacteur, il faut bien sûr que l'ordre des contraintes dans les BDDs reflète cette partition, les variables liées par des relations logiques dans un BDD devant toujours être proches dans l'ordre.

6.2.3 Quantification de formules

Nous abordons ici le problème de la quantification existentielle de variables booléennes et numériques de BDDs. Comme pour toute variable v et formule f, $\forall v\ f = \neg(\exists v\ f)$, on résout en même temps le problème de la quantification universelle. Nous aurons typiquement besoin d´éliminer les variables d'entrée d'une formule.

Éliminer une variable booléenne est aisé ; en effet par définition $\exists \beta f = f_{\bar{\beta}} \vee f_\beta$. En revanche éliminer une variable numérique est plus délicat. Pour éliminer une variable ξ d'un BDD f, il faut construire pour chaque chemin de f menant à un sous-BDD booléen le polyèdre associé, le projeter suivant la dimension ξ, et reconstruire un BDD numérique en extrayant les contraintes du polyèdre projeté. Cette opération peut produire de nouvelles contraintes, ce qui rend malaisé le partitionnement des contraintes en groupes de contraintes parallèles décrit ci-dessus.

Deux optimisations à l'algorithme de base peuvent être envisagées :

1. D'abord, on peut distinguer dans une conjonction de contraintes $m(\vec{x}, \vec{\xi})$ les contraintes faisant intervenir les variables $\vec{\xi}$ à éliminer des autres ; ceci revient à la décomposer ainsi :

   $$m(\vec{x}, \vec{\xi}) = m^s(\vec{x}) \wedge m^i(\vec{x}, \vec{\xi})$$

avec $m^i(\vec{x}, \vec{\xi})$ ne contenant que des contraintes faisant intervenir au moins une variable ξ_l. L'optimisation consiste à ne projeter que le polyèdre correspondant à $m^i(\vec{x}, \vec{\xi})$, après s'être assuré que le monôme $m^s(\vec{x})$ est satisfaisable.

2. du coup, on peut mettre en place un cache relativement efficace portant sur des BDDs pour l'opération $\lambda m^i.(\exists \vec{\xi}\ m^i)$.

Si l'on se refuse à effectuer des calculs polyédriques, en ne vérifiant pas la satisfaisabilité des chemins et en quantifiant les contraintes comme des variables booléennes lorsque la recherche dans le cache échoue, le BDD obtenu sera une surapproximation du résultat exact.

6.2.4 Évaluation partielle d'une fonction par un polyèdre

Nous verrons au chapitre suivant que les états concrets représentés par une place de la structure de contrôle sont connus à la fois sous forme de formule par l'*assertion de place* et sous forme de valeur abstraite par la définition de la place. Comme nous l'avons vu, les formules et les états convexes ont une expressivité incomparable. De plus, les opérations de cofacteur généralisé ignorent la sémantique des contraintes. On peut donc obtenir une évaluation partielle plus puissante en évaluant une formule f non seulement par l'assertion de la place considérée, mais aussi par sa définition (B, P), et plus précisément par le polyèdre P, B étant pris en compte par une opération booléenne de cofacteur généralisé.

L'évaluation partielle d'une fonction f par un polyèdre consiste à appliquer à f l'algorithme de suppression des chemins insatisfaisables, mais en partant du polyèdre P au lieu du polyèdre univers. Au cours du parcours descendant de f, tout nœud dont la contrainte ne sépare pas le polyèdre courant en deux polyèdres non vides est supprimé et remplacé par le fils adapté.

En fait, cette opération étant coûteuse, son utilisation n'est qu'optionnelle dans notre outil.

6.3 Conclusion

Les opérations que nous venons de présenter présentent des propriétés communes : leur version exacte, prenant en compte la sémantique des contraintes, nécessite des calculs polyédriques coûteux, mais on peut les approximer par des opérations purement booléennes et en utilisant les techniques de cache des BDDs évoquées au §5.1.1.

Nous utiliserons ces approximations lorsque cela sera possible. Par exemple, la simplification initiale des fonctions de transition et des formules spécifiant le programme sera effectuée par un algorithme exact, et les chemins insatisfaisables découverts seront mémorisé dans le "*don't care set*". Par la suite, nous simplifierons les fonctions et formules par une opération de cofacteur généralisé par le "*don't care set*". Bien sûr, nous ne supprimerons alors plus tous les chemins insatisfaisables.

Chapitre 7

Approximation des opérateurs de pré- et post-condition

Les opérateurs de postconditions et de préconditions ont été défini dans le cadre des systèmes dynamiques discrets. Ils calculent respectivement les successeurs et les prédécesseurs d'un ensemble d'états. Ce chapitre s'intéresse à l'implémentation de ces opérations sur notre treillis abstrait, et plus spécifiquement sur une structure de contrôle. Les opérateurs abstraits que nous allons décrire font des approximations supplémentaires par rapport à celles effectuées par l'abstraction standard des opérateurs concrets correspondants.
Ces opérateurs sont indispensables aux analyses avant/arrière de la structure de contrôle, mais jouent aussi une fonction importante, qui est de fournir les informations nécessaires aux heuristiques de raffinement de la structure de contrôle.

7.1 Introduction

Le problème est le suivant : étant donné une fonction de transition $\tau = ((\phi_i)_{1 \leqslant i \leqslant m}, (\psi_j)_{1 \leqslant j \leqslant p})$ et une valeur abstraite $(B, P) \in \mathcal{P}(\mathbb{B}^n) \times \mathcal{P}^{oly}(\mathcal{N}^p)$, calculer une surapproximation de $post(\tau)(B, P)$, sachant que l'on sait calculer simplement :

- la postcondition d'une formule booléenne par un vecteur de fonctions booléennes ;
- la postcondition d'un polyèdre par une transformation linéaire non gardée.

La difficulté vient du fait qu'un état convexe et la fonction de transition sont composites.

Pour illustrer la discussion, nous prendrons un programme possédant une variable d'état booléenne b et une variable d'état entière x, et dont la fonction de transition est :

$$\begin{cases} \phi_b &= b \wedge (x \leqslant 2) \\ \psi_x &= \textbf{if } b \wedge (x \leqslant 2) \textbf{ then } x + 10 \textbf{ else } x - 1 \end{cases}$$

7.1.1 Deux techniques naïves

Ain d'alléger les notations, nous supposons dans ce paragraphe qu'il n'y a pas d'entrée. Pour calculer la postcondition de la valeur abstraite (B, P), nous pouvons considérer la formule

$$B(\vec{b}) \wedge P(\vec{x}) \;\wedge\; \bigwedge_{1 \leqslant i \leqslant m} b' = \phi_i(\vec{b}, \vec{x}) \wedge \bigwedge_{1 \leqslant k \leqslant p} x'_k = \psi_k(\vec{b}, \vec{x})$$

et la projeter, d'une part sur $\vec{b'}$ — afin d'obtenir la partie booléenne du résultat —, et d'autre part sur $\vec{x'}$ en utilisant de manière appropriée des enveloppes convexes — pour obtenir la partie numérique. Le résultat obtenu est très imprécis, car les parties booléennes et numériques de la fonction de transition sont considérées de manière indépendante, et qu'on ne tire pas parti de la structure de contrôle.

Une solution plus précise est de décomposer la formule précédente en sous-formules gardées, de façon à ce que les gardes soient des formules convexes et que dans chaque sous-formule, les relations liant les variables numériques soient *linéaires* et les relations liant les variables booléennes ne fassent plus intervenir de contraintes numériques. La postcondition de chaque sous-formule gardée se calcule aisément, et on utilise la structure de contrôle pour obtenir le résultat sous la forme d'une union finie d'états convexes.

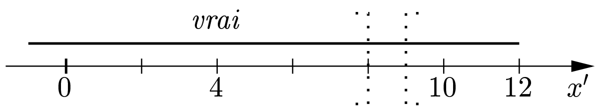
Exemple. — Supposons que l'on dispose d'une structure de contrôle à deux places définies par la partition $(vrai, x \leqslant 8) \cup (vrai, x \geqslant 9)$. Prenons $(B, P) = (vrai, 0 \leqslant x \leqslant 5)$.

Dans le premier cas nous projetons la formule

$$\begin{array}{lll} vrai \wedge 0 \leqslant x \leqslant 5 & \wedge & b' = b \wedge x \leqslant 2 \\ & \wedge & (b \wedge x \leqslant 2 \wedge x' = x + 10 \;\vee\; \neg(b \wedge x \leqslant 2) \wedge x' = x - 1) \end{array}$$

sur respectivement b' et x', ce qui donne

$$vrai \wedge (-1 \leqslant x \leqslant 12)$$



L'intersection avec les places de la structure de contrôle donne alors

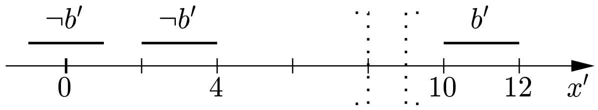
$$\boxed{(vrai, -1 \leqslant x \leqslant 8) \cup (vrai, 9 \leqslant x \leqslant 12)}$$

Dans le second cas la formule est découpée (d'après les valeurs de b et $x \leqslant 2$) en

$$\begin{array}{llllll} vrai \wedge 0 \leqslant x \leqslant 5 \;\wedge\; \{ & & x \leqslant 2 \wedge b & \wedge & b' \wedge x' = x + 10 & \\ & \vee & x \leqslant 2 \wedge \neg b & \wedge & \neg b' \wedge x' = x - 1 & \\ & \vee & x \geqslant 3 & \wedge & \neg b' \wedge x' = x - 1 & \} \end{array}$$

qui se projette en

$$\begin{array}{ll} & b' \wedge 10 \leqslant x' \leqslant 12 \\ \vee & \neg b' \wedge -1 \leqslant x' \leqslant 1 \\ \vee & \neg b' \wedge 2 \leqslant x' \leqslant 4 \end{array}$$



Si l'on effectue directement l'union convexe, on obtient certes le même résultat. Mais si l'on utilise effectivement la structure de contrôle pour représenter des unions explicites de valeurs abstraites, on va d'abord intersecter les sous-résultats avec la partition correspondante avant d'effectuer leur union convexe (c.f. §5.5.2). On obtient alors

$$\boxed{(\neg b', -1 \leqslant x \leqslant 4) \cup (b', 10 \leqslant x \leqslant 12)}$$

qui contient plus d'information.

□

La deuxième solution est en fait la plus précise qu'on peut envisager : elle correspond à la meilleure approximation supérieure de l'opérateur de postcondition concret dans le treillis abstrait partitionné.

Ces deux techniques souffrent cependant du même inconvénient : elles nécessitent la manipulation de formules portant à la fois sur les variables non primées et primées. Doubler ainsi le nombre de variables dans les BDDs est cher, et l'est encore plus dans les polyèdres, c.f. §2.4.4. Concernant la deuxième technique, la précision qu'elle offre va de pair avec une complexité élevée : le nombre de sous-formules à considérer croît exponentiellement avec le nombre de variables booléennes gardant des affectations linéaires et le nombre de contraintes présentes dans la fonction de transition.

7.1.2 Une technique issue de la compilation

Nous allons nous inspirer de la technique utilisée dans la compilation du langage LUSTRE [Ray91], qui propose une solution aux deux problèmes évoqués ci-dessus.

Pour éviter la multiplication du nombre de variables, la technique de compilation génère des versions séquentielles des fonctions de transitions comme suit : les variables sont triées selon leurs dépendances, un nombre réduit de variables auxiliaires sont introduites si nécessaire (par exemple pour l'échange de deux variables), et la postcondition est calculée progressivement par une séquence d'affectations.

EXEMPLE. — Dans l'exemple précédent, les variables b et x sont mutuellement dépendantes. En introduisant, par exemple, une variable primée b' associée à b pour briser le cycle de dépendance, une séquence possible serait :

$(vrai,\ 0 \leqslant x \leqslant 5)$
`if x<=2 then` $\longrightarrow (vrai,\ 0 \leqslant x \leqslant 2)$
    `b':= b` $\longrightarrow (b' = b,\ 0 \leqslant x \leqslant 2)$
`else` $\longrightarrow (\neg b,\ 3 \leqslant x \leqslant 5)$
    `b':= false` $\longrightarrow (c = \neg b \wedge \neg b',\ 3 \leqslant x \leqslant 5)$
`endif;` $\longrightarrow ((b' = b) \vee \neg b',\ 0 \leqslant x \leqslant 5)$
`if b then` $\longrightarrow (vrai,\ 0 \leqslant x \leqslant 5)$
    `if x<=2 then` $\longrightarrow (vrai,\ 0 \leqslant x \leqslant 2)$
        `x:= x+10` $\longrightarrow (vrai,\ 10 \leqslant x \leqslant 12)$
    `else` $\longrightarrow (vrai,\ 3 \leqslant x \leqslant 5)$
        `x:= x-1` $\longrightarrow (vrai,\ 2 \leqslant x \leqslant 4)$
    `endif` $\longrightarrow (vrai,\ 2 \leqslant x \leqslant 12)$
`else` $\longrightarrow (\neg b',\ 0 \leqslant x \leqslant 5)$
    `x:= x-1` $\longrightarrow (\neg b',\ -1 \leqslant x \leqslant 4)$
`endif;` $\longrightarrow (vrai,\ -1 \leqslant x \leqslant 12)$
`b:= b';`
$(vrai,\ -1 \leqslant x \leqslant 12)$

et le calcul progressif de la postcondition donne les résultats en petits caractères. Notons que si l'on n'effectue pas l'union convexe des résultats des deux branches du test sur b, on obtient un résultat identique à celui de la solution la plus précise du paragraphe précédent.

On peut faire mieux en remarquant que les conditions communes à plusieurs affectations peuvent être factorisées, ce qui permet en outre de ne pas introduire de variable primée :

$(vrai, 0 \leqslant x \leqslant 5)$
```
if x<=2 then
    if b then
        b := true; x := x+10     --> (b, 10<=x<=12)
    else
        x := x-1;     --> (¬b, -1<=x<=1)
    endif;
else
    b := false; x := x-1     --> (¬b, 2<=x<=4)
endif
```
$(vrai,\ -1 \leqslant x \leqslant 12)$

Si l'on n'opère pas d'union convexe lors de la fermeture des tests, l'union explicite des résultats des branches correspond encore à la deuxième solution décrite ci-dessus. □

Nous allons utiliser cette technique pour calculer les post- et pré- conditions. Suivant la séquence utilisée, on pourra obtenir des opérateurs plus ou moins précis. Comme nous l'avons mentionné au §4.4.6, il est en fait logique d'induire des approximations dans le calcul de postcondition, puisque la précision des analyses est de toute façon limitée par la finesse de la structure de contrôle.

7.1.3 Plan du chapitre

Nous définissons formellement les *séquences* au §7.2, ainsi que les opérateurs de postcondition et précondition associés à une séquence donnée.

Nous appellerons *séquentialisation* d'une fonction de transition l'opération qui consiste à générer une *séquence* (d'affectations) à partir de la fonction de transition. Comme nous venons de le voir, il n'y a pas de solution unique. La thèse [Ray91] nous a servi de référence, mais comme l'utilisation des séquences en vérification n'est pas la même qu'en compilation, nous avons fait des choix différents, qui résultent en un algorithme plus complexe. Cette opération est détaillée au §7.3.

Le §7.5 décrit le rôle de l'évaluation partielle et la prise en compte de l'*assertion* du programme (c.f. §3.3.3).

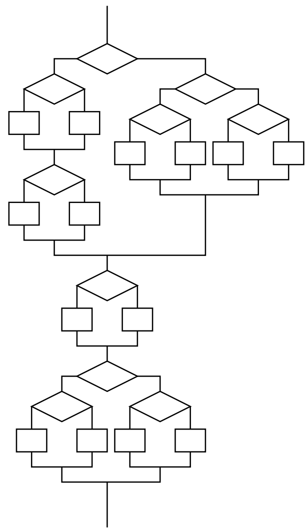
Nous expliquons enfin comment raffiner les séquences et généralisons le rôle des assertions dans le §7.6, ce qui permet à la fois d'améliorer la précision des calculs et l'obtention des informations nécessaires aux heuristiques de raffinement de la structure de contrôle, comme nous le verrons au chapitre 9.

7.2 Séquences

7.2.1 Syntaxe

Une *séquence*, qui est un graphe dirigé acyclique particulier, est définie par :

$$\begin{array}{rcl} \langle\text{sequence}\rangle & ::= & \langle\text{operation}\rangle; \ldots; \langle\text{operation}\rangle \\ \langle\text{operation}\rangle & ::= & \langle\text{action}\rangle \\ & | & \textbf{if } \langle\text{cond}\rangle \textbf{ then } \langle\text{sequence}\rangle \textbf{ else } \langle\text{sequence}\rangle \end{array}$$



Les *actions* sont des opérations pouvant être calculées de manière atomique — i.e. des affectations purement booléennes ou des affectations linéaires non gardées, et les *conditions* sont des formules partitionnant tout état convexe en deux états convexes — i.e. des contraintes linéaires ou des formules purement booléennes. Le schéma ci-contre est la représentation graphique d'une séquence sous forme d'organigramme classique, les losanges et rectangles représentant respectivement les tests et les actions. Comme pour les BDDs, les branches gauches correspondent à un test négatif.

7.2.2 Opérateurs associés

Les opérateurs de postconditions et préconditions associés à une séquence sont définis ainsi :

$$\begin{array}{rcl} post(op_1; op_2; \ldots; op_n)(X) & = & post(op_2; \ldots; op_n) \circ post(op_1)(X) \\ pre(op_1; op_2; \ldots; op_n)(X) & = & pre(op_1; \ldots; op_{n-1}) \circ pre(op_n)(X) \\ post(\textbf{if } c \textbf{ then } seq^+ \textbf{ else } seq^-) & = & post(seq^+)(X \sqcap c) \sqcup post(seq^-)(X \sqcap \neg c) \\ pre(\textbf{if } c \textbf{ then } seq^+ \textbf{ else } seq^-) & = & (pre(seq^+)(X) \sqcap c) \sqcup (pre(seq^-)(X) \sqcap \neg c) \\ post(act)(B,P) & = & \begin{cases} (bpost(act)(B), P) & \text{si l'action est booléenne} \\ (B, npost(act)(P)) & \text{si l'action est numérique} \end{cases} \\ pre(act)(B,P) & = & \begin{cases} (bpre(act)(B), P) & \text{si l'action est booléenne} \\ (B, npre(act)(P)) & \text{si l'action est numérique} \end{cases} \end{array}$$

Il nous reste à définir *bpost*, *bpre*, *npost* et *npre*. Dans chaque catégorie — booléenne ou numérique — d'actions, il en existe trois types : les actions standards $s := f$, les actions qui introduisent un tampon $s' := f$, et les actions de projection $s := s'$ qui projettent et éliminent la variable tampon.

Action booléenne. — Soit b une variable booléenne d'état, ϕ et B des formules purement booléennes. Dans le cas général

$$\begin{array}{rcl} bpost(b := \phi)(B) & = & B_b \wedge (b \Leftrightarrow \phi_b) \vee B_{\neg b} \wedge (b \Leftrightarrow \phi_{\neg b}) \\ & = & b \wedge \exists b(B \wedge \phi) \vee \neg b \wedge \exists b(B \wedge \neg\phi) \end{array}$$

Dans le cas de l'introduction d'une variable tampon b', on peut tirer profit du fait que b' n'apparaît ni dans B ni dans ϕ :

$$bpost(b' := \phi)(B) = B \wedge (b' \Leftrightarrow \phi)$$

Enfin, la projection se fait ainsi :

$$\begin{array}{rcl} bpost(b := b')(B) & = & \exists b'(\exists b B \wedge (b \Leftrightarrow b')) \\ & = & b \wedge (\exists b B)_{b'} \vee \neg b \wedge (\exists b B)_{\neg b'} \end{array}$$

La précondition est une simple substitution : $bpre(b \leftarrow \phi)(B) = B[\phi/b]$. Lorsque l'action est une introduction (vue en avant) $b' := \phi$, b' est automatiquement éliminé, et à l'inverse lorsque l'action est une projection $b := b'$, b' est introduite.

Action numérique. — Soient P un polyèdre et $x_k \leftarrow \sum_i a_i x_i + b$ une affectation linéaire. Cette affectation définit une transformation linéaire. La postcondition d'un polyèdre par l'affectation consiste à appliquer la transformation aux générateurs de P, et la précondition se fait en appliquant la transformation aux contraintes, ce qui opère une substitution, c.f. §2.4.4.

7.3 Séquentialisation d'une fonction de transition

7.3.1 Schéma général

Nous prendrons comme exemple la fonction de transition suivante :

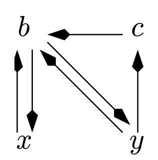
$$\begin{cases} b & := & x + y \leqslant -1 \\ c & := & b \wedge (z \geqslant 0) \\ x & := & \textbf{if } b \textbf{ then } x + 1 \textbf{ else } x - 1 \\ y & := & \textbf{if } b \wedge c \textbf{ then } y + 1 \textbf{ else } y - 1 \end{cases}$$

dans laquelle z est une entrée numérique.

La mise en séquence d'une fonction de transition se décompose en les trois étapes suivantes, résumées par l'algorithme 7.1.

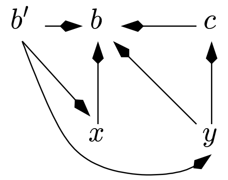
Construction du graphe de dépendance entre opérations. — On commence par construire un graphe dont les sommets sont étiquetés par les variables d'état $\vec{s}$; chaque sommet s_i représente l'opération d'affectation associée $s_i := f_i(\vec{s}, \vec{e})$, $\vec{e}$ représentant les variables d'entrée. Une arête $s_i \to s_j$ indique que pour calculer la valeur de s_i dans l'état successeur on a besoin de la valeur de s_j dans l'état courant ; autrement dit que la variable s_j apparaît effectivement dans la fonction f_i et qu'il faut calculer s_i avant s_j. La figure 7.1(a) représente le graphe de dépendance pour notre exemple. Les dépendances $s_i \to s_i$ ne sont pas représentées.

Introduction de tampons. — Le graphe construit à l'étape précédente peut comporter des cycles, rendant impossible tout ordonnancement correct des opérations associées aux sommets. Le but de cet étape est d'introduire un certain nombre de nouveaux sommets étiquetés par des variables *tampons*, afin d'éliminer ces cycles et d'obtenir un ordre partiel entre opérations. Sur notre exemple, le graphe précédent contenant des



(a) Graphe de dépendance

$$\begin{cases} b' := x + y \leqslant -1 \\ b := b' \\ c := b \wedge (z \geqslant 0) \\ x := \textbf{if } b \textbf{ then } x + 1 \textbf{ else } x - 1 \\ y := \textbf{if } b \wedge c \textbf{ then } y + 1 \textbf{ else } y - 1 \end{cases}$$



(b) Introduction d'un tampon b' et nouveau graphe de dépendance

FIG. 7.1 – Analyse de dépendance et introduction de tampon pour l'exemple

cycles passant par b, on a introduit une variable tampon b' en modifiant l'opération associée à b, ce qui a permis d'éliminer tout cycle, figure 7.1(b).

Génération proprement dite de la séquence. — La dernière étape consiste à générer une séquence en respectant les contraintes d'ordre, en choisissant les conditions sur lesquelles ouvrir des tests lorsque cela est nécessaire, puis les opérations à séquentialiser sous chaque branche d'un test avant sa fermeture, ceci en suivant le critère d'optimisation choisi. Plus précisément, lorsqu'on se trouve en présence d'un ensemble d'opérations à séquentialiser, les étapes sont les suivantes :

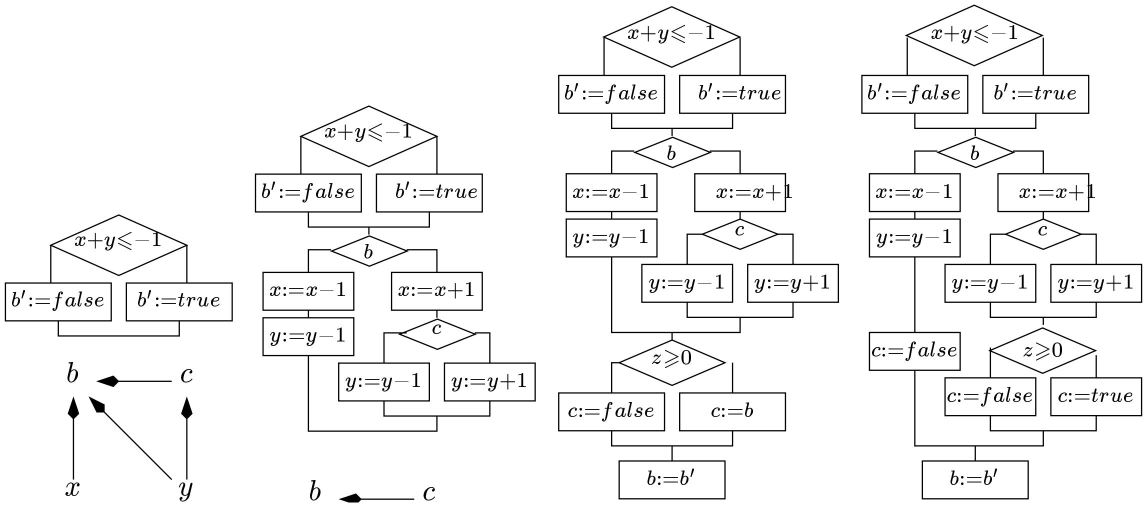
Génération des actions ordonnançables: on sélectionne les opérations qui sont des actions et qui n'ont plus de prédécesseurs dans le graphe de dépendance, et on les écrit dans la séquence, jusqu'à ce que ce cas soit épuisé. Si toutes les opérations ont été écrites, on a terminé, sinon il existe des opérations sans prédécesseur dans le graphe qui ne sont pas des actions, et il faut ouvrir un test pour débloquer la génération.

Sélection d'une condition et des opérations sous le test: il s'agit de sélectionner la condition sur laquelle on va ouvrir un test, et de décider ensuite des opérations que l'on va placer sous les deux branches du test. Si l'on ne procède à aucune factorisation, on ne sélectionne qu'une opération.

Appels récursifs: on réapplique l'algorithme récursivement aux opérations simplifiées sous les deux branches du test, et aux opérations non sélectionnées qui seront placées après la fermeture du test.

Algorithme 7.1 Sequentialise

```
Input: un ensemble Ops d'opérations
Output: une séquence Seq correspondante
begin
  G := GénèreGraphe(Ops)
  Gt := IntroduitTampons(G)
  Seq := Sequentialise(Gt)
end
where Sequentialise(G) ≡
  { Extraction des actions générables }
  actions := SelectActions(G)
  { Sélection d'une condition & Extraction des opérations sous test }
  cond := SelectCond(G)
  ops := SelectOps(cond,G); Gr := G \ ops
  { Évaluation des opérations sous test & Calcul des (nouvelles) dépendances }
  ops+ := EvalOps(cond,ops); ops- := EvalOps(¬cond ,ops)
  G+ := GénèreGraphe(ops+); G- := GénèreGraphe(ops-)
  { Appels récursifs }
  seq+ := Sequentialise(G+); seq- := Sequentialise(G-)
  seqr := Sequentialise(Gr)
  return (actions; if cond then seq+ else seq- endif; seqr)
```



(a) (b) (c) (d)

FIG. 7.2 – Séquentialisation de l'exemple

EXEMPLE. — Reprenons le graphe de la figure 7.1(b). Nous allons simuler l'algorithme de séquentialisation implémenté dans notre outil et décrit dans la suite. Initialement, il n'y a pas d'actions générables, et la seule opération *bloquante*, i.e., sans prédécesseur et n'étant pas une action, est celle associée à b'. Celle-ci ne comportant qu'une condition $x + y \leqslant -1$, on va donc la sélectionner pour le test à ouvrir. Dans les branches de ce test, seule b' sera sélectionnée, car $x + y \leqslant -1$ n'apparaît pas dans d'autre opération. On obtient donc la situation de la fig. 7.2(a), dans laquelle on a représenté la séquence générée et les opérations restant à séquentialiser. Les opérations bloquantes sont maintenant celles associées à x et y. La condition b apparaissant dans les deux opérations, alors que c n'apparaît que dans une, on sélectionne b pour ouvrir un test, et on placera dans ce test les opérations x et y évaluées sur b, auquelles on applique récursivement l'algorithme, pour obtenir la situation de la fig. 7.2(b). Maintenant c est la seule opération bloquante, dont la seule condition est $z \geqslant 0$. On ouvre donc un test sur cette contrainte en y plaçant c, et ensuite on écrit l'action b, ce qui donne la séquence de la fig. 7.2(c). Notons qu'on aurait pu placer c sous le test sur b, car c dépend de b ; on aurait alors obtenu la séquence de la fig. 7.2(d). □

7.3.2 Discussion des compromis

Examinons l'influence des choix possibles lors des différentes étapes de la séquentialisation. Les opérations seront écrites en caractères mathématiques, les séquences en caractères de machine à écrire. Pour une opération $s := f$, $\texttt{seq}(s := f)$ dénote une séquence traduisant l'opération.

Influence de la factorisation sur la précision. — Factoriser des opérations dépendant d'une condition commune c consiste à les intégrer au même test. Par exemple, si on a deux opérations $s_1 := (\textbf{if } c \textbf{ then } A \textbf{ else } B)$ et $s_2 := (\textbf{if } c \textbf{ then } C \textbf{ else } D)$, le choix de factoriser c

permet d'obtenir la séquence

if c then seq($s_1 := A; s_2 := C$) else seq($s_1 := B; s_2 := D$)

au lieu de

if c then seq($s_1 := A$) else seq($s_1 := B$);
if c then seq($s_2 := C$) else seq($s_2 := D$)

Éventuellement, les sous-séquences seq($s_1 := A; s_2 := C$) et seq($s_1 := B; s_2 := D$) peuvent être factorisées à leur tour. En compilation, le gain escompté est un code plus efficace en temps. Dans notre cas, nous en attendons en outre un gain en précision, comme l'a illustré notre exemple introductif au §7.1.2, puisque notre treillis n'est pas distributif (§5.5.2).

Influence de la factorisation sur la complexité des calculs — Cependant, si on a une opération supplémentaire $s_3 := f_3$ qui doit être ordonnancée entre s_1 et s_2 et ne dépend pas de c, la factorisation implique une *duplication de code* :

if c then seq($s_1 := A; s_3 := f_3; s_2 := C$) else seq($s_1 := B; s_3 := f_3; s_2 := D$)

En compilation, cela n'influence que la taille du code, dans notre cas l'influence s'étend à la complexité en temps, puisque les opérateurs parcourent potentiellement *toutes* les branches des tests et $s_3 := f_3$ peut être une opération complexe contenant beaucoup de conditions.

Influence de l'ordre des opérations sur la factorisation. — L'ordre étant partiel, il existe plusieurs ordres linéaires qui lui sont compatibles. Le choix d'un tel ordre linéaire a une influence sur la factorisation des conditions. Considérons trois opérations :

$s_1 :=$ **if** y **then** A **else** B
$s_2 :=$ **if** x **then** C **else** D
$s_3 :=$ **if** y **then** E **else** (**if** z **then** F **else** G)

On suppose de plus que s_1 et s_2 doivent être écrites avant s_3 mais sont indépendantes entre elles. Le premier ordre possible (s_1, s_2, s_3) ne permet pas la factorisation, si on s'interdit de dupliquer l'opération associée à s_2 :

if y then seq($s_1 := A$) else seq($s_1 := B$);
if x then seq($s_2 := C$) else seq($s_2 := D$);
if y then seq($s_3 := E$) else if z then seq($s_3 := F$) else seq($s_3 := G$)

Le deuxième ordre (s_2, s_1, s_3) est meilleur :

if x then seq($s_2 := C$) else seq($s_2 := D$);
if y then
  seq($s_1 := A; s_3 := E$)
else
  seq($s_1 := B$); if z then seq($s_3 := E$) else seq($s_3 := F$)

Un ordre peu contraint facilite donc la factorisation sans duplication d'opération.

Dans le compilateur LUSTRE, on fixe d'emblée un ordre linéaire compatible avec l'ordre partiel. Dans notre outil, comme la séquentialisation n'est pas coûteuse au regard des analyses avant et arrière, on considère toujours l'ordre partiel, et on le simplifie dès que possible, par exemple lorsque les opérations sont placées sous un test, ce qui simplifie leur fonction associée et peut faire disparaître des dépendances.

Nombre de variables tampon. — Nous avons vu que, dans le cas booléen comme dans le cas numérique, les variables supplémentaires augmentent la complexité en temps et/ou en espace des opérations élémentaires, il peut donc être intéressant de minimiser leur nombre. En contrepartie, on augmente les contraintes d'ordre entre opérations. A l'opposé, si on introduit autant de tampons qu'il y a de variables d'état, il n'y a plus de contraintes d'ordre, exceptées celles liant les variables à leur tampon associé. Il est à noter qu'on peut autoriser la réutilisation de tampons, au prix d'une augmentation des contraintes d'ordre (et de la complexité de l'algorithme !). Dans le cas numérique, on peut aussi utiliser le fait que certaines opérations sont inversibles pour éliminer des tampons.

Choix d'un espace de conditions. — A priori, une condition est soit une contrainte d'inégalité apparaissant dans une quelconque opération, soit une formule purement booléenne en garde d'une affectation numérique. On peut considérer un autre choix, qui est de ne prendre pour conditions booléennes que des littéraux au lieu de formules plus générales, et ce au prix d'un accroissement possible du nombre de tests.

Exemple. — Considérons par exemple l'affectation $x' :=$ **if** $b_1 \wedge b_2$ **then** $x+1$ **else** $x-1$. Si les formules quelconques sont autorisées, on obtient

```
if b1 and b2 then x := x+1 else x:= x-1;
```

alors que si seuls les littéraux sont acceptés deux tests sont nécessaires :

```
if b1 then
  if b2 then then x := x+1 else x := x-1
else
  x := x-1;
```

□

Clairement la première solution est meilleure. Cependant il ne paraît pas aisé de généraliser à des formules booléennes quelconques l'algorithme de sélection d'une condition, présenté au §7.3.5, et qui cherche à factoriser au mieux les tests. Les conditions, numériques ou booléennes, seront donc toujours des littéraux.

Nous détaillons dans les paragraphes suivants l'algorithme que nous utilisons pour séquentialiser une fonction de transition, en reprenant les étapes décrites dans le paragraphe précédent.

7.3.3 Construction du graphe de dépendance et introduction de tampons

Pour construire le graphe des dépendances, on crée un sommet par variable d'état, puis on construit les arêtes comme suit :

Pour chaque variable booléenne on calcule le support du BDD associé, qui se compose de variables booléennes et de pseudo-variables associées à des contraintes. On crée une arête pour chaque variable booléenne d'*état* y appartenant, ainsi que pour toute variable numérique d'*état* apparaissant dans une des contraintes.

Pour chaque variable numérique on procède comme pour les variables booléennes, c'est-à-dire qu'on inspecte le support de son MTBDD, mais il faut en outre tenir compte des dépendances induites par les affectations linéaires constituant les feuilles du MTBDD. Pour cela, on extrait du MTBDD l'ensemble de leurs feuilles, et on crée une arête pour toute variable numérique d'état y apparaissant.

Si le graphe ainsi construit est acyclique, on peut directement passer à la génération de la séquence. Sinon, il faut introduire des variables tampons. Nous avons fait le choix de la simplicité dans l'introduction des tampons : on en minimise autant que possible le nombre, mais on associe chacun d'eux à une unique variable d'état (on ne réutilise pas les tampons).

Trouver un ensemble minimal (au sens de la cardinalité) de sommets d'un graphe tels que le graphe ôté de ces sommets soit acyclique est un problème NP-complet [GJ79], aussi utilisons-nous un algorithme heuristique. Plus précisément, nous utilisons l'algorithme de Bourdoncle [Bou93] de décomposition d'un graphe en sous-composantes fortement connexes. Les têtes des sous-composantes forment un ensemble ayant la propriété recherchée. Le même problème se posant également pour la détermination des points d'élargissement dans un graphe de contrôle, l'algorithme sera abordé plus en détail au §8.3.

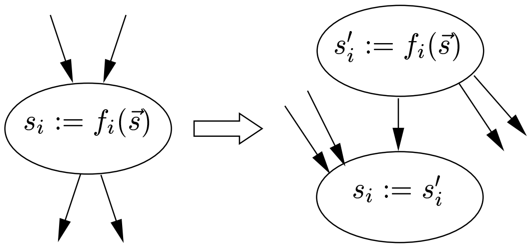


FIG. 7.3 – Introduction d'un tampon

Un sommet associé à son opération $(s_i,\ s_i := f_i(\vec{s}))$ ainsi sélectionné, est remplacé par $(s_i, s_i := s'_i)$, et on ajoute au graphe le couple sommet-opération $(s'_i, s'_i := f_i(\vec{s}))$, figure 7.3. Les dépendances sont mises à jour en remplaçant les arêtes $s_i \to s_j, j \in succ(i)$ par $s'_i \to s_j, j \in succ(i)$, et en ajoutant $s'_i \to s_i$.

Théorème 3 *Le graphe résultant est acyclique.*

Preuve: Notons K l'ensemble des sommets du graphe G, avant introduction des tampons. On a $K = K_\nabla \cup (K \setminus K_\nabla)$ avec K_∇ l'ensemble des têtes des sous-composantes fortement connexes. Après introduction des tampons, on obtient un graphe G' dont l'ensemble des sommets est $K' = K_\nabla \cup K'_\nabla \cup (K \setminus K_\nabla)$.

Montrons que G' ne peut posséder de cycle. Supposons que G' contienne un cycle. Le sous-graphe de G' induit par $(K \setminus K_\nabla)$ est identique au sous-graphe de G induit par $(K \setminus K_\nabla)$, lequel est acyclique. Donc le cycle contient au moins un sommet appartenant à $K_\nabla \cup K'_\nabla$. Or les sommets de K_∇ n'ont pas de successeurs dans G', et les sommets de K'_∇ n'ont pas de prédécesseurs, c.f. fig 7.3. □

7.3.4 Sélection des actions

L'algorithme est simple : on enlève du graphe les sommets sans prédécesseur associés à des actions, et on les écrit dans la séquence en construction. La suppression de ces sommets fait que de nouveaux sommets se retrouvent sans prédécesseur, aussi itère-t-on cette opération jusqu'à ce que le graphe soit vide ou qu'aucun des sommets sans prédécesseurs ne soit une action.

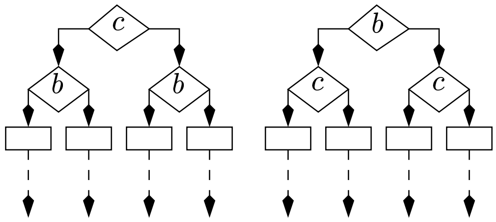
7.3.5 Sélection d'une condition

Une fois les actions immédiatement générables retirées du graphe, la séquentialisation est bloquée par des opérations sans prédécesseur dans le graphe mais qui nécessitent l'ouverture de tests. Le principe de l'algorithme est de sélectionner une condition apparaissant dans le maximum d'opérations sans prédécesseur, i.e. les opérations *bloquantes*. Pour cela, on calcule les supports de ces opérations, et on trie les conditions apparaissant dans les supports suivant leur fréquence d'apparition. Concrètement, ceci est réalisé en calculant les unions multi-ensemblistes des supports, et en choisissant les éléments ayant la multiplicité maximale.

Nous avons testés les possibilités suivantes, lors de la construction du multi-ensemble :

1. on prend l'union multi-ensembliste des conditions booléennes et numériques des MTBDDs concernés, et on y ajoute les conditions numériques des BDDs ;
2. on y ajoute également les conditions booléennes des BDDs apparaissant aussi dans des MTBDDs ; en effet quand un BDD ou un MTBDD dépendent de la même condition booléenne, il peut être intéressant de favoriser le placement du BDD sous le même test que le MTBDD, même si cela n'est pas nécessaire. Par exemple dans la figure 7.2(d), page 76, le fait que l'opération associée à la variable c soit placée sous le test sur b permet de lier la valeur de c avec celles de x et y, alors que la solution de la figure 7.2(c) ne le permet pas.

Une autre heuristique est nécessaire lorsque plusieurs conditions ont la multiplicité maximale : on peut favoriser soit les contraintes, soit les booléens. Ouvrir les tests d'abord sur les contraintes peut être judicieux, car cela diminue dans certains cas (fig 7.4) le nombre d'intersection avec des contraintes lors d'un calcul, sachant qu'elles sont généralement plus chères que les intersections booléennes. Nous favorisons donc les contraintes dans l'algorithme 7.2.



(a) numériques (b) booléennes

FIG. 7.4 – Priorité aux conditions ...

7.3.6 Sélection des opérations

Nous avons fait le choix d'autoriser la duplication d'actions, afin de favoriser la factorisation, sans trop augmenter la taille de la séquence.

L'algorithme 7.3 commence par sélectionner et ôter du graphe les opérations sans prédécesseurs dépendant de la condition sélectionnée. Comme pour la sélection des actions, on réitère ce processus, en autorisant aussi la sélection des actions, jusqu'à ce qu'aucune opération sans prédécesseur ne dépende de la condition. L'algorithme réel fait même une seconde passe pour ôter les dernières actions inclues dans l'espoir de factoriser des opérations.

7.3.7 Conclusion

L'algorithme de séquentialisation présenté est adapté de [Ray91] pour les besoins de la vérification. Il convient d'avoir à l'esprit dans les paragraphes suivants que la factorisation des tests est très importante pour la précision (et le coût) des opérateurs associés aux séquences, et que celle-ci est favorisée bien évidemment par un nombre réduit de conditions dans la fonction de transition, ainsi que par des dépendances faibles entre opérations.

Algorithme 7.2 SelectCond

Input: un graphe G d'opérations
Output: une condition Cond
begin
  minops := $\{op \in \mathrm{G} \mid pred(op) = \emptyset\}$
  multiset := MultiSetCond(minops)
  conds := maxs(multiset)
  Cond := choose(conds)
end
where MultiSetCond(ops) $\equiv$
  multiset := $\emptyset$
  let ops = num_ops $\cup$ bool_ops **in**
  { opérations numériques }
  for op **in** num_ops **do**
    supp:= *supp*(op)
    multiset:= multiset $\cup$ $\{(x, 1) \mid x \in \mathrm{supp}\}$
  done
  { Opération booléennes : 2 possibilités }
  let set = $\boxed{1}$ $\emptyset$ $\boxed{2}$ $\{x \mid \exists k \;\; (x, k) \in \mathrm{multiset}\}$ **in**
  for op **in** bool_ops **do**
    supp := *supp*(op) $\cap$ (CONS $\cup$ set) { CONS : ens. des contraintes }
    multiset := multiset $\cup$ $\{(x, 1) \mid x \in \mathrm{supp}\}$
  done
  return multiset

Algorithme 7.3 SelectOps

Input: une condition cond, un graphe G
Output: un ens. d'opérations ops
begin
  ops := $\emptyset$
  repeat
    mins := $\{op \in \mathrm{G} \mid pred(op) = \emptyset\}$
    sels := $\{op \in \mathrm{mins} \mid \mathrm{is\_action}(op) \vee \mathrm{cond} \in supp(op)\}$
    G := G $\setminus$ sels; ops := ops $\cup$ sels
  until sels = $\emptyset$
  return ops
end

7.4 Assertions de place et d'arc

Une séquence construite selon l'algorithme du paragraphe précédent ne tient pas compte de l'*assertion* du programme, qui restreint les combinaisons possibles des entrées et des états et permet de modéliser l'environnement du programme.

La notion d'assertion va avoir une signification plus large dans notre contexte de vérification, combinant analyses avant et arrière. Nous allons distinguer les assertions d'*arc* et les assertions de *place* :

Une assertion d'arc est une formule $f(\vec{s}, \vec{e})$ associée à un arc $q \rightsquigarrow q'$ et qui représente une condition nécessaire sur l'état et l'entrée pour que la postcondition de la définition de q intersecte la définition de q' ;

Une assertion de place est une formule $f(\vec{s}, \vec{e})$ associée à une place q, qui est égale à la disjonction des assertions associées aux arcs sortants de q ; elle représente une condition nécessaire pour que la postcondition de la définition de q intersecte l'ensemble des états considérés dans la structure de contrôle, i.e., l'ensemble des états dangereux.

Pour une place de la structure de contrôle initiale, son assertion de place est la conjonction de l'assertion du programme et de la définition de la place, alors connue sous forme de formule ; pour un arc de la structure de contrôle initiale, son assertion est égale à l'assertion de sa place origine.

Par la suite, lors des analyses et du raffinement, ces assertions seront renforcées par la méthode décrite au §7.6. En fait le mécanisme des assertions est utilisé non seulement pour modéliser l'environnement du programme, mais aussi pour stocker les informations obtenues par les analyses sur la structure de contrôle, et ce mécanisme est crucial dans les heuristiques de raffinement du chapitre 9.

7.5 Séquences, évaluation partielle et assertions

7.5.1 Rôle de l'évaluation partielle

L'un des intérêts d'une structure de contrôle est de pouvoir utiliser des fonctions composantes, plus simples que la fonction dont on calcule le point fixe. Les fonctions composantes sont plus simples car on utilise des techniques d'évaluation partielle pour les spécialiser sur le domaine sur lequel on va les appliquer. Dans le cas de programmes impératifs par exemple, cette évaluation partielle est implicite. Dans notre cas, cette méthode est puissante. En effet l'évaluation partielle de la fonction de transition globale par une formule a pour effet de faire disparaître des conditions dans la fonction de transition, ce qui ne peut qu'améliorer la séquence correspondante.

Dans notre cadre, l'évaluation partielle sur une place se fait en spécialisant les fonctions de transition sur l'assertion de la place et en appliquant aux fonctions simplifiées l'algorithme de séquentialisation.

Après une analyse, qui réduit les définitions *et les assertions de place* à leur partie dangereuse, on peut aussi profiter du gain en précision et en complexité que procure l'évaluation partielle en reséquentialisant la fonction de transition simplifiée par les nouvelles définitions.

Le raffinement de la structure de contrôle s'accompagne donc aussi du raffinement des opérateurs de postcondition associés aux séquences.

Algorithme 7.4 SequentialisePlace

```
Input: un ensemble d'opérations Ops, une place Pl de la structure de contrôle
Output: une séquence Seq spécialisée sur la définition de la place
begin
  Ops:= restrict(Ops,ast(Pl))
  Seq := SequentialiseAssert(Ops,ast(Pl))
end
```

7.5.2 Prise en compte des assertions dans la séquentialisation

Si nous spécialisons les fonctions de transition sur l'assertion de place, il faut s'assurer que les états dont on calcule la postcondition vérifie cette assertion. Si l'assertion est une formule convexe, et représente une valeur abstraite A, il suffit d'intersecter la valeur abstraite X dont on veut calculer la postcondition avec A. Nous étendons dans ce but la syntaxe des séquences par une nouvelle opération assert dont l'interprétation est :

$$post(assert(A))(X) = X \cap A \quad , \quad pre(assert(A))(X) = X \cap A$$

Lorsque l'assertion est une formule non convexe, les choses sont un peu plus compliquées : on peut ne tenir compte que partiellement de l'assertion, en se contenant d'intersecter les valeurs abstraites par l'état convexe obtenu par la factorisation de la formule (c.f. paragraphe 6.1.2), ou on peut factoriser récursivement la formule comme le fait l'algorithme 6.1 qui convertit une formule en union exacte d'états convexes.

EXEMPLE. — Reprenons toujours le même exemple, et supposons que l'assertion soit $(b \Rightarrow c) \wedge (c \Rightarrow y \geqslant 0)$. Si on ne tient compte que de la partie convexe de la formule, on va obtenir la séquence

```
assert(c or not b); ...
```

où la sous-formule $(c \Rightarrow y \geqslant 0)$ est ignorée. En exploitant la décomposition hiérarchique de l'assertion, on obtient

```
assert(c or not b);
if y>=0 then
  ...
else
  assert(not c);
  ...
endif
```

□

L'algorithme 7.5 implémente la *séquentialisation sous assertion*. Nous faisons le choix de ne pas faire d'approximation à ce niveau vu le rôle important de ce mécanisme. Par conséquent on factorise récursivement l'assertion et les tests ouverts pour la satisfaire ne sont pas refermés avant que toutes les opérations n'ait été effectuées.

Algorithme 7.5 SequentialiseAssert

```
Input: un ens. d'opérations ops, une assertion assertion
Output: une séquence seq prenant en compte l'assertion
begin
  return SequentialiseRec(ops, vrai, assertion)
end
where SequentialiseRec(ops, sat, assertion) ≡
  { sat : partie de l'assertion satisfaite }
  let (bool,mon,reste) = Decompose(Assertion) in
  sat := sat ∧ bool ∧ mon
  if reste = vrai then
    ops := Eval(ops,sat)
    seq := Sequentialise(ops)
    return assert(bool,mon); seq
  else
    let Δ(c,G,D) = reste in
    seq- := SequentialiseRec(ops, sat ∧¬c, G)
    seq+ := SequentialiseRec(ops, sat ∧ c, D)
    return (assert(bool,mon); if c then seq+ else seq-)
  end if
```

7.6 Micros-états et renforcement d'assertion

Les opérateurs de post- et pré- conditions ont aussi pour rôle de nous fournir des informations destinées aux heuristiques de raffinement d'états. Ces heuristiques ont besoin d'informations du type : quelles sont les conditions devant être satisfaites pour que tel arc soit emprunté dans la structure de contrôle ?

7.6.1 Principe du renforcement d'assertion

Exemple introductif. — Examinons la figure 7.5, dans laquelle on a représenté deux points de contrôle q, q' avec leur définition (cercles gras), relié par une transition $q \rightsquigarrow q'$. La séquence correspond à la fonction de transition

$$\left\{\begin{array}{lcl} b' & := & b; \\ x' & := & \textbf{if } b \textbf{ then } x+1 \textbf{ else } x-1 \end{array}\right.$$

Les rectangles arrondis clairs représentent les étapes du calcul de la postcondition de la définition du point q, et les rectangles arrondis plus foncés les étapes du calcul de la précondition de la définition du point q'.

Il est clair que la transition $q \rightsquigarrow q'$ ne peut être prise (en avant ou en arrière) que par un état satisfaisant $\neg b$. Mais les unions convexes, que ce soit dans un calcul avant ou arrière, nous empêchent de le détecter. Si on intersecte le résultat du calcul avec la définition adéquate *avant* de réaliser l'union convexe, on obtiendrait les résultats souhaités : $(\neg b, -1 \leqslant x < 0)$ en

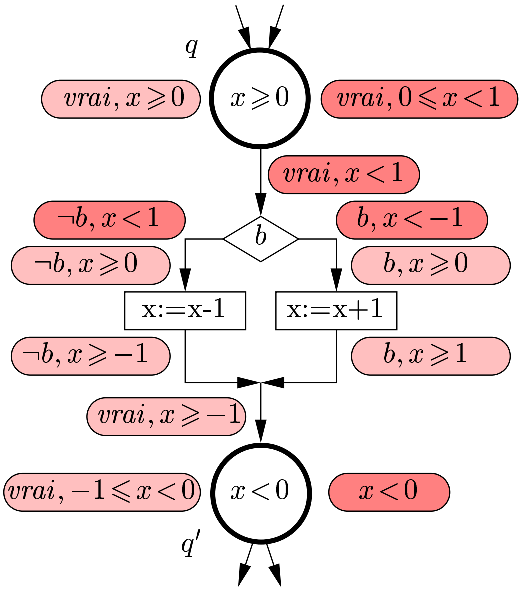


FIG. 7.5 – Séquence entre deux points de contrôle

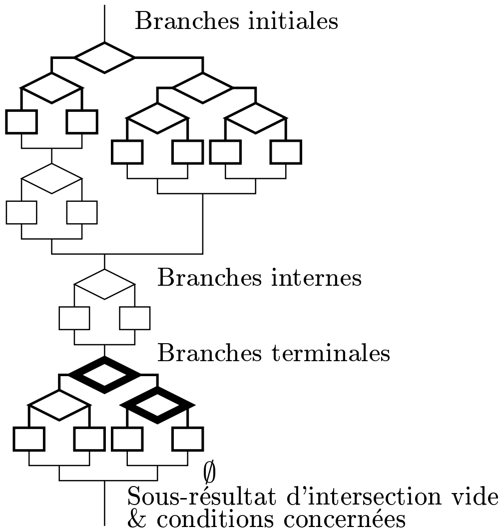


FIG. 7.6 – Classification des tests et branches d'une séquence

avant, et $(\neg b, 0 \leqslant x < 1)$ en arrière, et on détecte que $\neg b$ est condition nécessaire pour qu'un état appartenant à $def(q)$ puisse emprunter l'arc $q \rightsquigarrow q'$. □

Une première approche, implémentée dans l'outil, a été d'intersecter les résultats des branches des tests *terminaux* (c.f. fig 7.6) avec la définition de l'état destination avant de calculer leur union convexe. Lors d'un calcul avant, lorsque un sous-résultat a une intersection vide avec la définition du successeur, la conjonction des conditions de test sous lesquelles se trouve la branche concernée constitue une condition *suffisante* pour que *l'arête ne soit pas prise*, et surtout cette condition porte sur la valeur des variables *dans l'état source* : en effet les tests sont toujours ouvert avant modification des variables concernées, puisque les dépendances sont respectées. On peut donc étiqueter les arcs $q \rightsquigarrow q'$ par des assertions renforcées :

$$ast(q \rightsquigarrow q') = ast(q \rightsquigarrow q') \wedge \bigwedge_{t \in T} \neg m_t$$

où T est l'ensemble des branches terminales dont le résultat n'intersecte pas $def(q')$ et m_t est la conjonction de conditions de test sous lesquelles se trouve la branche t.

Dans le cas du calcul arrière, on intersecte les sous-résultats des branches *initiales* avec la définition origine avant de réaliser l'union convexe, et en cas d'intersection vide, on renforce l'assertion d'arc comme pour le calcul avant. On peut aussi exploiter le calcul arrière pour renforcer $ast(q \rightsquigarrow q')$ par la partie booléenne de $pre(def(q'))$. En revanche on ne peut aisément exploiter la partie numérique, car elle est donnée sous forme de polyèdre et non de formule, et les contraintes définissant ce polyèdre ne correspondent pas forcément à des contraintes apparaissant dans le source du programme, seules contraintes autorisées dans les formules.

Afin d'exploiter au mieux les assertions d'arc, il apparaît naturel de construire une séquence par *arc* plutôt que par *sommet* du graphe de contrôle, et de spécialiser bien sûr ces séquences sur les assertions d'arcs. La détection de branches vides permet le renforcement des assertions et une simplification des séquences. Ce choix peut augmenter le coût des analyses, notamment lorsque les assertions des arcs sortants d'un sommet sont peu différenciées, mais est nécessaire pour obtenir des informations suffisamment précises pour les heuristiques de raffinement.

7.6.2 Micro-états et analyse interne de séquences

Le mécanisme que nous venons de décrire permet donc de spécialiser une séquence non seulement sur la définition d'un sommet origine, mais aussi sur la définition d'un sommet destination. Cependant, seules les branches des tests initiaux ou terminaux sont analysées. Une approche plus générale consiste à effectuer un véritable calcul d'accessibilité combiné avant et arrière au sein de la séquence, en considérant ses *micro-états.*

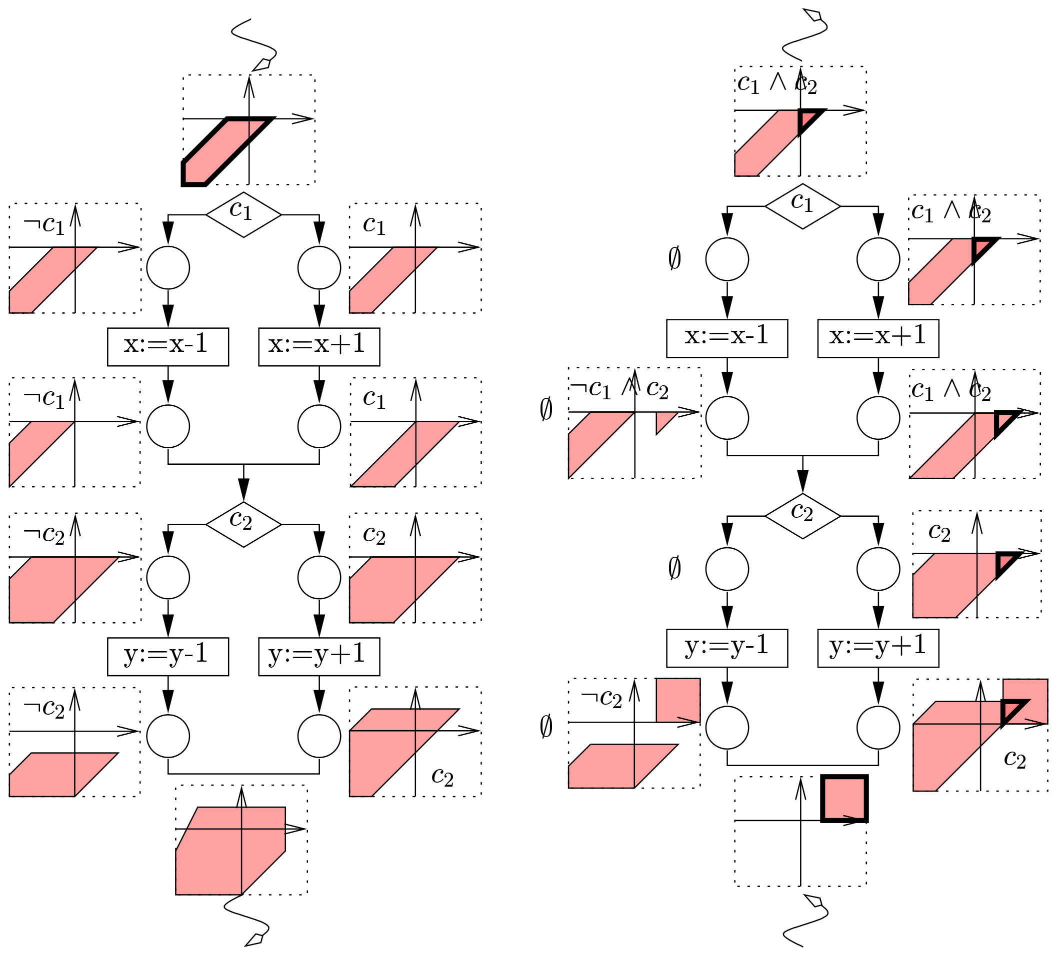
États synchrones et micro-états. — Le grand intérêt des programmes synchrones en vérification est qu'on ne considère que les états *synchrones* ou globaux et qu'on ne se préoccupe pas des calculs élémentaires permettant de passer d'un état synchrone à un autre. C'est au contraire un rôle dévolu au compilateur, qui a pour mission de générer un ordonnancement correct de ces calculs. Par exemple le compilateur LUSTRE utilise pour cela les séquences. À titre de comparaison, dans les systèmes asynchrones, ces différentes étapes sont détaillées, ainsi que tous les entrelacements (ou ordonnancements) possibles de ces étapes, ce qui multiplie considérablement le nombre d'états. Dans la communauté synchrone, on appelle *micro-états* [PS88] ces états intermédiaires se trouvant sur un chemin de calculs reliant deux états synchrones globaux. Les propriétés ne portent que sur les états synchrones, mais on va appliquer les notions définies sur les états synchrones (définition, parties accessibles, coaccessibles, dangereuses) aux micro-états de la séquence pour contrôler les approximations induites par les unions convexes.

EXEMPLE. — Soit la séquence suivante :

```
if x+y<=-1 then b := true else b := false fi;
if c1 then x:=x+1 else x:=x-1 fi;
if c2 then y:=y+1 else y:=y-1 fi;
c1 := c2; c2:= z>=0;
```

avec z une entrée numérique. Nous supposons que les définitions source et origine sont respectivement $-1 \leqslant y - x \leqslant 1 \wedge y \leqslant 0$ et $x \geqslant 1 \wedge y \geqslant 0$, figure 7.7. La technique du paragraphe précédent consistant à ne pas refermer les tests initiaux et terminaux ne permet pas ici de simplifier cette séquence. Nous allons voir comme la techniques des micro-états le permet, en examinant le bout de séquence concernant x et y. □

Utilisation de micro-états dans les séquences. — Nous allons considérer dans les séquences les micro-états se trouvant juste après l'ouverture d'un test et juste avant sa fermeture, comme l'illustre la figure 7.7. Lors de la création d'une séquence, la *définition* de ces micro-états sera l'état convexe univers. Ensuite, par calcul de la postcondition de la définition source de la séquence, on va propager en avant cette définition dans les micro-états et obtenir leur partie *accessible*, fig 7.7(a). De manière similaire, par calcul de la précondition de la définition destination de la séquence, on va propager en arrière cette définition, et l'intersecter avec la partie accessible en avant des micro-états, fig 7.7(b). On obtient donc pour chaque micro-état sa partie *dangereuse*, c'est-à-dire atteignable à partir du point de contrôle origine et pouvant mener au point de contrôle destination.



(a) Propagation en avant (b) Propagation en arrière et intersection

FIG. 7.7 – Micro-états dans une séquence

Si la partie dangereuse des micro-états est vide, on renforce l'assertion de la transition, et on reséquentialise. Dans l'exemple de la fig 7.7, cette propagation des définitions nous permet de renforcer l'assertion avec la formule $c_1 \wedge c_2$ et de supprimer deux ouvertures de tests et les enveloppes convexes induites. Par ailleurs, lors d'un calcul de post- ou pré- condition, on intersecte les résultats intermédiaires avec la définition des micro-états. Ce mécanisme permet non seulement d'opérer les intersections avec les définitions avant les unions convexes dans les branches terminales et initiales, comme expliqué au paragraphe précédent, mais permet également de détecter des incompatibilité à l'intérieur des séquences.

7.6.3 Formalisation

Nous reprenons la syntaxe précédente des séquences, mais toute séquence ou sous-séquence *seq* sera munie d'un attribut, son micro-état initial *seq.initial*. Nous montrons en effet que considérer les autres micro-états n'améliore pas la précision, et leur élimination est un gain en temps comme en espace.

Algorithme de séquentialisation avec raffinement. — La création d'une séquence entre deux sommets q et q' commence par la séquentialisation de la fonction de transition sous l'assertion $ast(q \rightsquigarrow q')$, suivie d'une propagation en avant de $def(q)$ dans les micro-états, puis d'une propagation arrière de $def(q')$ avec intersection. Si on trouve des branches vides, on renforce l'assertion et on réitère le processus. On a fait le choix de ne faire qu'un cycle combiné de propagation avant / arrière.

Algorithme 7.6 SequentialiseMicroEtat

```
Input: un ens. d'opérations ops, une assertion ast, deux états convexes initial
et final
Output: un séquence seq spécialisée sur l'arc initial ⇝ final
begin
  repeat
    seq := SequentialiseAssert(ops,ast)
    propage_avant(seq)(initial)
    propage_arriere(seq)(final)
    insat := Insat(seq)
    ast := ast and not insat
  until insat = faux
  return seq
end
```

Collecte des branches vides. — La collecte des branches vides se fait en parcourant tous les tests de la séquence. On remarque qu'il suffit d'examiner les micro-états en tête de séquence, puisque la vacuité d'un micro-état se propage en arrière lors de la propagation arrière. La propagation des définitions est décrite par l'algorithme 7.8.

Propagation des définitions et opérateur de post- et pré- condition. — Lors des analyses sur le graphe de contrôle, comme nous l'avons dit, nous allons intersecter les résultats intermédiaires obtenus à l'intérieur des séquences avec les micro-états.

Le nombre de micro-états étant cependant élevé, nous allons montrer qu'il suffit d'opérer des intersections avec les seuls micro-états initiaux se trouvant juste après une ouverture de test, ainsi que l'état initial de la séquence complète.

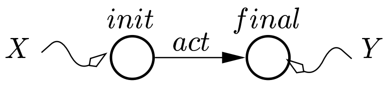
En effet toute séquence, fig 7.8.(a), peut-être vue comme un automate interprété du type de la fig 7.8.(b). Dans un tel automate, il y a 3 types d'arcs : ceux correspondant aux ouvertures de test, ceux aux actions, et ceux qui correspondent aux fermetures de test. Notons que les 2 constructeurs de séquence : la mise en séquence et l'ouverture de test, sont représentés. Nous allons montrer que seuls les intersections suivant l'ouverture d'un test sont utiles.

Micro-états successeurs d'une action: prenons une action *act*. Cette action forme une séquence encadrée par les micro-états *init* et $final$. La propagation des définitions donne:

Algorithme 7.7 Insat

```
Input: une séquence seq étiquetée avec des micro-états
Output: l'ens. des branches vides de la séquence donnée par la formule insat
begin
  if seq.initial = ⊥ then
    return vrai
  else
    insat := faux
    for op in seq do
      if op = if c then seq+ else seq- then
        insat := insat ∨ c ∧ Insat(seq+) ∨ ¬c ∧ Insat(seq-)
    done
  end if
end
```

$$\begin{array}{rcl} init & = & X \sqcap pre(act)(final) \\ final & = & post(act)(X) \sqcap Y \end{array}$$



On a donc $post(act)(init) \subseteq post(act)(pre(act)(final)) \subseteq final$, par monotonicité de *post* puis en utilisant $post(act) \circ pre(act) \leqslant id$ [1]. Donc pour tout X, $post(act)(X \sqcap init) \subseteq final$, et il est inutile d'opérer l'intersection avec $final$ lorsqu'on calcule une postcondition. Il est clair que ceci se propage le long d'une séquence d'actions. Pour le calcul d'une précondition, montrons que l'intersection avec $final$ est également inutile, i.e. que $pre(act)(X \sqcap final) \sqcap init = pre(act)(X) \sqcap init$.

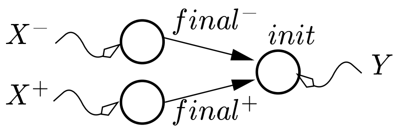
$$\begin{array}{rcl} pre(act)(X \sqcap final) \sqcap init & = & pre(act)(X) \sqcap pre(act)(final) \sqcap init \\ & = & pre(act)(X) \sqcap pre(act)(final) \sqcap X \sqcap pre(act)(final) \\ & = & pre(act)(X) \sqcap init \end{array}$$

Pour une séquence d'actions, la propriété se propage en arrière. Supposons-la vraie pour une séquence de n actions.

$$\begin{array}{l} pre(act_1)(pre(act_2; \ldots ; act_{+1})(X \sqcap final) \sqcap int) \sqcap init \\ = pre(act_1)(pre(act_2; \ldots ; act_{+1})(X) \sqcap int) \sqcap init \\ = pre(act_1)(pre(act_2; \ldots ; act_{+1})(X)) \sqcap init \end{array}$$

Micro-états successeurs d'une fermeture de test: On a

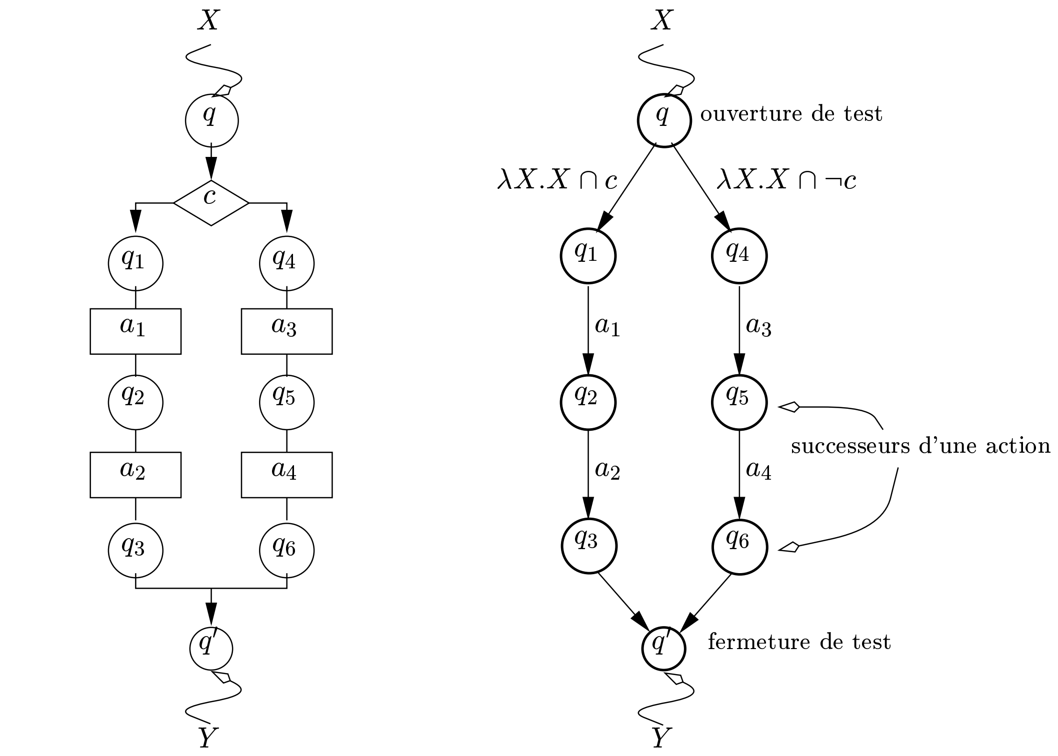
$$\begin{array}{rcl} final^+ & = & X^+ \sqcap init \\ final^- & = & X^- \sqcap init \\ init & = & (X^+ \sqcup X^-) \sqcap Y \end{array}$$



Pour la postcondition, on a $final^+ \sqcup final^- \subseteq init$, donc l'intersection avec le micro-état $final$ est inutile, et de même pour la précondition.

Micro-états prédécesseurs d'une ouverture de test: On a

[1] En effet la postcondition et la précondition d'un état convexe par une action est exact, et $post(act)$ est déterministe

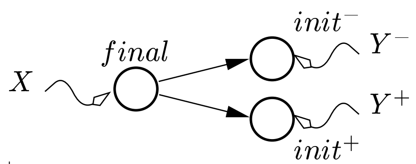


(a) Représentation en organigramme

(b) Représentation sous forme d'automate interprété

FIG. 7.8 – Séquences et micro-états

$$\begin{array}{rcl} final & = & X \sqcap (init^+ \sqcup init^-) \\ init^+ & = & X \sqcap c \sqcap Y^+ \\ init^- & = & X \sqcap \neg c \sqcap Y^- \end{array}$$



De même que pour le cas précédent, comme $init^+ \sqcup init^- \subseteq final$, l'intersection avec $final$ est inutile.

Les algorithmes de propagation (7.8), et de post- et pré-condition, (7.9), ne considèrent donc que les micro-états initiaux.

Notons toutefois qu'il pourrait être utile de mémoriser les micro-états finaux des séquences pour le calcul de la précondition. En effet, on est obligé de parcourir en arrière toutes les branches des tests, même si après intersection avec les micro-états initiaux, on se rend compte que la plupart de ces branches ne sont pas “actives”. Ceci est susceptible de se produire notamment lorsque l'on débute une analyse arrière et que les valeurs abstraites manipulées sont plus petites que les définitions.

7.6.4 Micro-états et détection de contraintes incompatibles

Le raffinement d'assertion par détection de branches non dangereuses (donc vides) est aussi un moyen de repérer des contraintes incompatibles. Cependant le mécanisme des micro-états permet de repérer l'incompatibilité de contraintes sur lesquelles des tests ont été ouverts

Algorithme 7.8 : Propagation des définitions

propage_avant(seq)(X) $\equiv$
  begin
    seq.initial := X
    return prop_rec(seq)(seq.initial)
  end
  where
    prop_rec(**if** c **then** seq^+ **else** seq^-)(X) $\equiv$
      propage_avant(seq^+)(X $\sqcap$ c) $\sqcup$ propage_avant(seq^-)(X $\sqcap$ $\neg c$)
    prop_rec($op_1;\ldots;op_n$)(X) $\equiv$ prop_rec($op_2;\ldots;op_n$) $\circ$ prop_rec(op_1)(X)
    prop_rec(act)(X) $\equiv$ $post$(act)(X)

propage_arriere(seq)(X) $\equiv$
  begin
    seq.initial := prop_rec(seq)(X) $\sqcap$ seq.initial
    return seq.initial
  end
  where
    prop_rec(**if** c **then** seq^+ **else** seq^-)(X) $\equiv$
      propage_arriere(seq^+)(X) $\sqcup$ propage_arriere(seq^-)(X)
    prop_rec($op_1;\ldots;op_n$)(X) $\equiv$ prop_rec($op_1;\ldots;op_{n-1}$) $\circ$ prop_rec(op_n)(X)
    prop_rec(act)(X) $\equiv$ pre(act)(X)

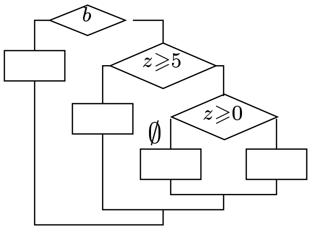
Algorithme 7.9 : Opérateurs de postcondition et de précondition

post(seq)(X) $\equiv$ post_rec(seq)(X $\sqcap$ seq.init)
where
  post_rec(**if** c **then** seq^+ **else** seq^-)(X) $\equiv$ post(seq^+)(X) $\sqcup$ post(seq^-)(X)
  post_rec($op_1;\ldots;op_n$)(X) $\equiv$ post_rec($op_2;\ldots;op_n$) $\circ$ post_rec(op_1)(X)
  post_rec(act)(X) $\equiv$ $post$(act)(X)

pre(seq)(X) $\equiv$ pre_rec(seq)(X) $\sqcap$ seq.init
where
  pre_rec(**if** c **then** seq^+ **else** seq^-)(X) $\equiv$ pre(seq^+)(X) $\sqcup$ pre(seq^-)(X)
  pre_rec($op_1;\ldots;op_n$)(X) $\equiv$ pre_rec($op_1;\ldots;op_{n-1}$) $\circ$ pre_rec(op_n)(X)
  pre_rec(act)(X) $\equiv$ pre(act)(X)

seulement dans la branche dans laquelle se trouve la cascade de tests, et non de manière globale.

Exemple. — Prenons la séquence de la figure ci-contre. La conjonction $z \geqslant 5 \wedge \neg(z \geqslant 0)$ est insatisfaisable, mais l'assertion d'arc sera renforcée par la formule $(\mathbf{b} \wedge z \geqslant 5) \Rightarrow z \geqslant 0$, alors que le renforcement par la formule $z \geqslant 5 \Rightarrow z \geqslant 0$ serait plus fort. □



Le mécanisme des micro-états n'est donc pas capable de distinguer ce qui relève de l'incompatiblité de contraintes de ce qui relève de la condition nécessaire pour que la postcondition d'une définition intersecte une autre définition.

7.7 Conclusion

Les opérateurs que nous avons définis permettent donc de calculer des post- et préconditions d'état avec un bon compromis précision / complexité, et surtout de séparer les préconditions des différentes transitions sortantes d'un sommet du graphe de contrôle, par spécialisation des transitions non seulement sur la définition origine, mais aussi sur la définition destination. Le compromis précision/complexité est maintenu au cours du raffinement de la structure de contrôle, car celui-ci induit des séquences plus simples et plus précises par spécialisation.

La seule technique ayant ces propriétés dont nous ayons connaissance dans la littérature est la thèse de H. Wong-Toï [WT94], dans le contexte plus simple de la vérification d'automates temporisés ([AD90] et plus spécifiquement le modèle défini dans [NSY92]). L'analyse utilise un treillis abstrait similaire au nôtre, dans lequel une valeur abstraite est la conjonction (A, Z) d'un ensemble d'états de contrôle et d'une zone (c.f. §2.4.3). Les postconditions correspondant aux *transitions discrètes* ne sont pas approximées, mais celles correspondant au *passage du temps* le sont. L'effet du passage du temps sur une valeur abstraite (A, Z) est ainsi défini, en notant Urg l'ensemble des états de contrôle dont on peut sortir par une action urgente :

$$post^{\nearrow}(A, Z) = (A \cap Urg, Z) \cup \bigcup_{q \in A \setminus Urg} (q, Z^{\nearrow} \cap Inv(q))$$

L'approximation utilisée est la suivante :

$$\overline{post}^{\nearrow}(A, Z) = \begin{cases} (A, Z) & \text{si } A \subseteq Urg \\ (A, Z^{\nearrow} \cap Inv(q)) & \text{si } A \not\subseteq Urg \ \wedge \ \forall q' \in A,\ Inv(q) = Inv(q') \\ (A, Z) & \text{sinon} \end{cases}$$

Ce type d'approximation est cependant un peu du type "tout ou rien" : il consiste à ignorer l'effet des invariants limitant le passage du temps à moins que ceux-ci soient tous identiques, ou que pour chaque état dans A il y ait une action urgente. Comme dans notre outil, le raffinement de la structure de contrôle fait que le nombre d'états de contrôle représentés par une valeur abstraite (A, Z) diminue, ce qui améliore la précision de l'opérateur $\overline{post}^{\nearrow}$

Chapitre 8

Analyse sur une structure de contrôle

Nous détaillons dans ce chapitre les algorithmes d'analyses de la structure de contrôle. Ceux-ci sont relativement classiques, mais il convient d'en tirer parti au mieux dans notre cadre particulier de raffinement de structure de contrôle. Nous proposons aussi plusieurs variantes de l'élargissement standard susceptibles d'améliore la précision des résultats.

LA manière dont nous menons les différents types d'analyses va être décrite de manière chronologique. Le §8.1 présente les analyses préliminaires que nous commençons par effectuer, ainsi que la création de la structure de contrôle initiale. Le §8.2 décrit l'analyse d'une structure de contrôle qui utilise le treillis abstrait des états convexes. Le §8.3 s'intéresse plus spécifiquement au choix d'une stratégie d'itération chaotique ainsi qu'à la détermination d'un ensemble de points d'élargissement. Enfin le §8.4 se concentre sur l'élargissement en décrivant différents perfectionnements dans l'utilisation d'un opérateur d'élargissement sur un treillis abstrait, qui n'est pas nécessairement celui des polyèdres convexes.

8.1 Structure de contrôle initiale

8.1.1 Espace d'état initial

Soient S et E respectivement l'espace des états et des entrées du programme. En l'absence d'assertion, la structure de contrôle initiale devra couvrir S. En revanche, si l'assertion est différente de *vrai*, on peut se restreindre aux états tels qu'il existe des entrées satisfaisant l'assertion. Nous prenons donc comme espace d'état initial

$$S' = \{\vec{s} \in S \mid \exists \vec{e} \in E \ assert(\vec{s}, \vec{e})\}$$

L'assertion est donnée par une formule mixte. Le calcul de $\exists \vec{e} \in E \ assert(\vec{s}, \vec{e})$ n'est cependant pas si trivial qu'il en a l'air car les quantifications portent non seulement sur des booléens, mais aussi des entiers ou rationnels. Il faut appliquer le paragraphe 6.2.3. Nous ne souhaitons pas faire d'approximation à ce niveau, aussi utilisons-nous un calcul exact. On obtient ainsi S' sous la forme d'une formule mixte $\Lambda(\vec{s})$.

8.1.2 Réduction par analyse d'accessibilité booléenne

On peut mener une analyse d'accessibilité booléenne, avant puis arrière, prenant en compte l'insatisfaisabilité de contraintes, pour réduire l'espace des états considérés. Ceci permet de démarrer les analyses utilisant le treillis abstrait des états convexes sur un espace réduit, et simplifie parfois considérablement la taille des séquences de la structure de contrôle initiale.

Ces analyses booléennes sont très classiques, le seul point original est la prise en compte des contraintes incompatibles. De façon à limiter leur coût, nous n'effectuerons pas à chaque étape une simplification exacte, mais nous utiliserons le *don't care set* prenant en compte les contraintes parallèles et dans lequel on a mémorisé les chemins insatisfaisables détectés lors de la simplification des fonctions de transitions, et des divers formules du programme. Comme l'analyse booléenne va combiner des contraintes provenant de fonctions différentes, certaines combinaisons incompatibles ne seront pas prise en compte par ce *don't care set*.

Soient $(\phi_i)_{i=1\ldots m}$ les fonctions de transition associées aux variables booléennes d'état (b_i), $care$ la négation du *don't care set*, $init^b(\vec{b})$, $inv^b(\vec{b})$ et $bad^b(\vec{b})$ les parties booléennes de l'ensemble des états initiaux, de la propriété et de sa négation, obtenues par élimination existentielle des contraintes dans les formules $init(\vec{b},\vec{x})$, $inv(\vec{b},\vec{x})$, $\neg inv(\vec{b},\vec{x})$,

La prise en compte des conjonctions de contraintes insatisfaisables mémorisées dans le *don't care set* s'effectue en considérant les contraintes comme des entrées booléennes devant satisfaire la formule $care$. Nous avons donc en fait une assertion globale $A = ast \wedge care$ sur les vraies entrées booléennes et les contraintes.

Par ailleurs, nous ne nous intéressons ni au futur des états erronés, ni au passé des états initiaux.

Analyse avant. — L'analyse avant consiste à itérer l'application de l'opérateur

$$\lambda X\ .\ init^b \vee post(\vec{\phi})(X \wedge inv^b \wedge A)$$

jusqu'à convergence, avec

$$post(\vec{\phi})(f)(\vec{b}) = \exists \vec{b'}, \vec{e}\ .\ f(\vec{b'},\vec{e}) \wedge \vec{b} = \vec{\phi}(\vec{b'},\vec{e})$$

L'opération coûteuse est le calcul de la postcondition, pour lequel [CBM89a] a proposé des optimisations importantes, évitant l'introduction de variables primées lorsque la relation de transition est une fonction, ce qui est notre cas.

Ces optimisations sont fondées sur la propriété suivante de l'opérateur "constraint", noté $\downarrow$ et présenté au §6.2.1 :

Pour tout vecteur $\vec{\phi}$ et toute formule f,
$post(\vec{\phi})(f) = post(\vec{\phi} \downarrow f)(vrai)$, avec $\vec{\phi} \downarrow X = \overrightarrow{\phi_i \downarrow X}$.

On a alors, en notant $\phi_i^f = \phi_i \downarrow f$,

$$\begin{array}{rcl}
post(\vec{\phi})(f)(\vec{b}) & = & \exists \vec{b'}, \vec{e} \,.\, \bigwedge_{1 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e})) \\
& = & \exists \vec{b'}, \vec{e} \,.\, \bigvee \left\{ \begin{array}{l} b_1 \wedge \phi_1^f(\vec{b'}, \vec{e}) \wedge \bigwedge_{2 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e})) \\ \neg b_1 \wedge \neg \phi_1^f(\vec{b'}, \vec{e}) \wedge \bigwedge_{2 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e})) \end{array} \right. \\
& = & \bigvee \left\{ \begin{array}{l} b_1 \wedge \exists \vec{b'}, \vec{e} \,.\, \phi_1^f(\vec{b'}, \vec{e}) \wedge \bigwedge_{2 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e})) \\ \neg b_1 \wedge \exists \vec{b'}, \vec{e} \,.\, \neg \phi_1^f(\vec{b'}, \vec{e}) \wedge \bigwedge_{2 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e})) \end{array} \right. \\
& = & \bigvee \left\{ \begin{array}{l} b_1 \wedge \exists \vec{b'}, \vec{e} \,.\, \bigwedge_{2 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e}) \downarrow \phi_1^f(\vec{b'}, \vec{e})) \\ \neg b_1 \wedge \exists \vec{b'}, \vec{e} \,.\, \bigwedge_{2 \leqslant i \leqslant m} (b_i = \phi_i^f(\vec{b'}, \vec{e}) \downarrow \neg \phi_1^f(\vec{b'}, \vec{e})) \end{array} \right. \\
& = & \ldots
\end{array}$$

Analyse arrière. — L'analyse arrière consiste à itérer l'application de l'opérateur

$$\lambda X \,.\, bad^b \vee pre_{inv^b \wedge A}(\vec{\phi})(X \wedge \neg init^b)$$

jusqu'à convergence, avec

$$pre_g(\vec{\phi})(f)(\vec{b}) = \exists \vec{b'}, \vec{e} \,.\, g(\vec{b}, \vec{e}) \wedge \vec{b'} = \vec{\phi}(\vec{b}, \vec{e}) \wedge f(\vec{b'})$$

Le calcul de $pre_g(\vec{\phi})(f)$ consiste en fait à substituer de manière simultanée les $\phi_i(\vec{b}, \vec{e})$ aux b_i dans $f(\vec{b})$, puis à intersecter le résultat avec g et à éliminer $\vec{e}$. La substitution simultanée se fait de la manière suivante :

$$\begin{array}{rcl}
f[\vec{b} \leftarrow \vec{\phi}] & = & f[b_i \leftarrow \phi_i, 1 \leqslant i \leqslant m] \\
& = & ite(\phi_1,\, f_{b_1}[b_i \leftarrow \phi_i, 2 \leqslant i \leqslant m],\, f_{\neg b_1}[b_i \leftarrow \phi_i, 2 \leqslant i \leqslant m]) \\
& = & \ldots
\end{array}$$

Pour que les opérations de cofacteur sur f soient efficaces, il convient de traiter les variables b_i dans l'ordre croissant dans les BDDs.

Bien sûr, on n'applique l'analyse arrière qu'au sous-ensemble de l'espace d'état calculé par l'analyse avant, bien que par souci de simplicité nous ne l'ayons pas reflété dans les notations.

L'intersection du résultat de ces deux analyses booléennes avec la formule mixte $\Lambda(\vec{s})$ du paragraphe précédent nous donne une nouvelle formule mixte définissant l'espace des états potentiellement dangereux, que l'on note toujours $\Lambda(\vec{s})$.

8.1.3 Création de la structure de contrôle

Avant toute analyse, nous n'avons aucune idée de la structure de contrôle adéquate pour la preuve de la propriété, car celle-ci dépend de la dynamique du programme. Nous construisons donc initialement une structure de contrôle très grossière, qui distingue seulement les états initiaux, finaux et corrects.

Nous ne souhaitons pas faire d'approximations à ce niveau, pour des raisons mentionnées ci-dessous, aussi nous exigeons une couverture exacte de ces ensembles par des états convexes, de telle manière que l'ensemble des places Q vérifient les propriétés suivantes :

$$\begin{array}{lcl} \bigcup_{q\in Q_0} \mathit{def}(q) & = & \mathit{init}(\vec{s}) \wedge \Lambda(\vec{s}) \\ \bigcup_{q\in Q_F} \mathit{def}(q) & = & \neg\mathit{inv}(\vec{s}) \wedge \Lambda(\vec{s}) \\ \bigcup_{q\in Q\setminus Q_F} \mathit{def}(q) & = & \mathit{inv}(\vec{s}) \wedge \Lambda(\vec{s}) \end{array}$$

Les membres droits des égalités sont connues sous la forme de formules mixtes. Leur décomposition exacte en union d'états convexes se calculent par l'algorithme 6.1.

Avant toute analyse, la relation de transition $\rightsquigarrow$ est complète modulo deux exceptions :

- les places finales sont des puits, car nous ne nous intéressons pas au futur des états erronés ;
- les places initiales sont des sources ; ceci est fondé car comme $\mathit{init} \Rightarrow \mathit{inv}$ (sans quoi la propriété est trivialement fausse), les états initiaux sont représentés à la fois par les places initiales et correctes ; pour toute transition concrète, il existe donc toujours une transition abstraite lui correspondant.

La séparation exacte des différents types d'états évite l'inclusion d'états erronés dans les points de contrôle corrects et garantit donc une précision minimale. Ce choix peut résulter en un grand nombre de places pour la structure de contrôle initiale. Le nombre de places initiales et finales n'est pas pénalisant, puisque ces places sont respectivement sources et puits ; en revanche le nombre de places correctes l'est beaucoup plus, car le nombre d'arcs dans le graphe varie quadratiquement avec ce paramètre. Heureusement, on peut toujours "retarder" la propriété d'invariance dans une variable d'état booléenne `ok`, de telle manière qu'il n'y ait qu'une seule place correcte définie par ok et une seule place finale définie par $\neg ok$, comme l'illustre l'exemple qui suit.

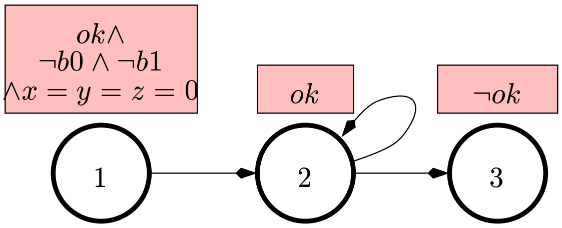
8.1.4 Assertions et séquences associées aux arcs

Pour chacun des arcs $q \rightsquigarrow q'$, on va initialiser son assertion associée (c.f. §7.4) par l'assertion spécialisées sur le sommet q : $\mathit{ast}(q \rightsquigarrow q') = \mathit{assert} \wedge \mathit{def}(q)$. On applique ensuite l'algorithme de séquentialisation avec raffinement du 7.6.3. S'il s'avère après raffinement que $\mathit{ast}(q \rightsquigarrow q') = \mathit{faux}$, la transition est supprimée.

Exemple. — Reprenons le programme suivant, déjà utilisé au chapitre 4, page 33 :

```
state b0,b1,ok: bool; x,y: int;
transition
  b0' = not b1;
  b1' = b0;
  x' = if b0=b1 then x+1 else x;
  y' = if b0 xor b1 then y+1 else y;
  ok' = ok and (x>=y);
initial ok and not b0 and not b1 and x=0 and y=0;
invariant ok;
```

La structure de contrôle initiale pour ce programme est donné figure 8.1. La propriété $x \geqslant y$ a été retardée dans la variable ok. Les rectangles grisés donnent les définitions des places. Les micro-états des séquences sont donnés par les étiquettes **micro**. On constate ici le grand défaut de l'algorithme actuel de séquentialisation, qui n'ouvre des tests que sur des variables booléennes et non des formules. □



`Assertion` $1 \rightsquigarrow 2$ = $ok \wedge \neg b0 \wedge \neg b1 \wedge x = y = 0$

`Assertion` $2 \rightsquigarrow 2$ = $ok \wedge x \geqslant y$

`Assertion` $2 \rightsquigarrow 3$ = $ok \wedge \neg(x \geqslant y)$

`Séquence` $1 \rightsquigarrow 2$

micro $= (ok \wedge \neg b0 \wedge \neg b1, \{x = y = 0\})$

`x := x+1; b0 := true;`

`Séquence` $2 \rightsquigarrow 2$

micro $= (ok, \{x \geqslant y\})$

`Tb0 := not b1;`

`if` `b0` **`then`**

  `if` `b1` **`then`**

    micro $= (ok \wedge b0 \wedge \neg Tb0 \wedge b1, \{x \geqslant y\})$
    `x := x+1;`

  `else`

    micro $= (ok \wedge b0 \wedge Tb0 \wedge \neg b1, \{x \geqslant y\})$
    `y := y+1;`

  `end if` ;

`else`

  `if` `b1` **`then`**

    micro $= (ok \wedge \neg b0 \wedge \neg Tb0 \wedge b1, \{x \geqslant y\})$
    `y := y+1;`

  `else`

    micro $= (ok \wedge \neg b0 \wedge Tb0 \wedge \neg b1, \{x \geqslant y\})$
    `x := x+1;`

  `end if` ;

`end if` ;

`b1 := b0; b0 := Tb0;`

`Séquence` $2 \rightsquigarrow 3$

micro $= (ok, \{y \geqslant x + 1\})$

`Tb0 := not b1; ok := false;`

`if` `b0` **`then`**

  `if` `b1` **`then`**

    micro $= (\neg ok \wedge b0 \wedge \neg Tb0 \wedge b1, \{y \geqslant x + 1\})$
    `x := x+1;`

  `else`

    micro $= (\neg ok \wedge b0 \wedge Tb0 \wedge \neg b1, \{y \geqslant x + 1\})$
    `y := y+1;`

  `end if` ;

`else`

  `if` `b1` **`then`**

    micro $= (\neg ok \wedge \neg b0 \wedge \neg Tb0 \wedge b1, \{y \geqslant x + 1\})$
    `y := y+1;`

  `else`

    micro $= (\neg ok \wedge \neg b0 \wedge Tb0 \wedge \neg b1, \{y \geqslant x + 1\})$
    `x := x+1;`

  `end if` ;

`end if` ;

`b1 := b0; b0 := Tb0;`

FIG. 8.1 – Structure de contrôle initiale, avec séquences et assertions d'arc

8.2 Analyse d'une structure de contrôle

L'analyse d'une structure de contrôle est une application directe des paragraphes §4.4.1, §4.4.2, §4.4.3 et §7.6, et repose sur la combinaison classique des analyses avant et arrière, statiques et dynamiques. On suppose qu'avant un cycle d'analyse, toutes les places sont statiquement dangereuses, i.e. se trouvent sur un chemin reliant un place initiale à une place finale.

Analyse avant. — Dans un premier temps, on effectue une analyse dynamique en avant qui calcule pour chaque point de contrôle une surapproximation de la partie accessible de sa définition. Ceci revient à déterminer une surapproximation de la plus petite solution du système :

$$\mathit{reach}(q) = \bigsqcup_{q' \rightsquigarrow q} \mathit{post}(\mathit{seq}_{q' \rightsquigarrow q})(\mathit{reach}(q')) \cap \mathit{def}(q)$$

où $\mathit{post}(\mathit{seq}_{q' \rightsquigarrow q})$ est l'opérateur de postcondition correspondant à la séquence associée à l'arc $q' \rightsquigarrow q$.

Après convergence, on restreint les définitions à leur partie accessible : $\mathit{def}(q) = \mathit{reach}(q)$, puisque les autres états ne se trouvent pas sur un chemin partant d'un état initial ; bien sûr les places dont la nouvelle définition est $\perp$ sont supprimées du graphe.

Pour chaque arc $q \rightsquigarrow q'$ du graphe, on cherche également à renforcer l'assertion $\mathit{ast}(q \rightsquigarrow q')$ en inspectant les micro-états de sa séquence par l'algorithme 7.7. Si la nouvelle assertion est fausse, alors on supprime l'arc. Si elle est seulement renforcée, on reséquentialise la fonction de transition en utilisant l'algorithme 7.6.

Une analyse statique en arrière permet alors de détecter et d'éliminer les places accessibles mais ne pouvant plus mener à une place finale, après la suppression des arcs d'assertion vide.

Analyse arrière. — On effectue ensuite une analyse en arrière qui calcule une surapproximation de la partie coaccessible des définitions (réduites). On détermine donc une surapproximation de la plus petite solution du système :

$$\mathit{coreach}(q) = \bigsqcup_{q \rightsquigarrow q'} \mathit{pre}(\mathit{seq}_{q \rightsquigarrow q'})(\mathit{coreach}(q')) \cap \mathit{def}(q)$$

Encore une fois, on réduit les définitions : $\mathit{def}(q) = \mathit{coreach}(q)$ et on renforce les assertions d'arc, en supprimant l'arc ou en raffinant sa séquence le cas échéant.

Puis on élimine par une analyse statique en avant les places devenues ainsi inaccessibles.

On effectue ces analyses alternées jusqu'à convergence.

Il est important de préciser l'intérêt de ce schéma : d'une part le fait de simplifier la relation de transition abstraite simplifie les itérations chaotiques et diminue le nombre de points d'élargissement ; d'autre part les opérateurs de postcondition et de précondition sont des surapproximations des opérateurs concrets, dont la précision dépend de leur domaine et de leur codomaine. À chaque fois que l'on réduit les définitions, ceci permet de spécialiser ces opérateurs (§7.6) et d'améliorer les résultats.

Une fois la convergence obtenue, et si bien sûr il n'y a pas eu succès de la preuve entretemps, seul un raffinement de la structure de contrôle peut permettre de restreindre encore l'espace des états dangereux.

EXEMPLE. — Le tableau suivant montre les étapes d'un calcul dynamique en avant de la structure de contrôle initiale de la figure 8.1, en utilisant la stratégie d'itération $1(23)^*$.

Étapes	Places		
	1	2	3
1	$ok \wedge \neg b0 \wedge \neg b1$ $\{x=y=0\}$	$ok \wedge b0 \wedge \neg b1$ $\{x=1, y=0\}$	$\perp$
2	–	$ok \wedge b0$ $\{x=1, 0 \leqslant y \leqslant 1\}$	$\perp$
3	–	$ok \wedge (b0 \vee b1)$ $\{x \geqslant 1, y \geqslant 0\}$	$\neg ok \wedge (\neg b0 \vee b1)$ $\{1 \leqslant x \leqslant y, x+y \geqslant 4\}$
4	–	ok $\{x \geqslant 1, y \geqslant 0\}$	$\neg ok$ $\{1 \leqslant x \leqslant y, x+y \geqslant 4\}$
5	–	–	–

La figure 8.2 donne la nouvelle structure de contrôle. Les rectangles grisés donnent les nouvelles définitions, les deux formules non encadrées sont des assertions d'arc (évaluées sur la définition de la place 2). Les assertions et séquences restent identiques. Sur cet exemple, l'analyse en arrière ne nous donne aucune information supplémentaire. □

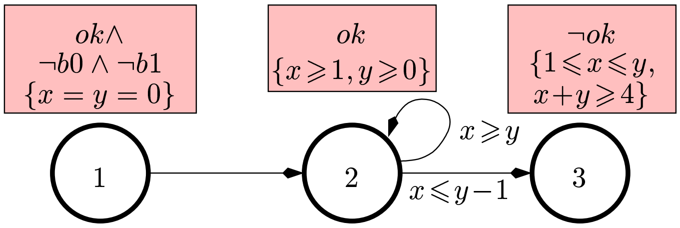


FIG. 8.2 – Résultat de l'analyse avant sur la structure de contrôle initiale

8.3 Stratégie d'itération chaotique

La notion d'itération chaotique a été présentée au §2.3.2. L'objet de ce paragraphe est le choix d'une bonne stratégie pour les analyses dynamiques du paragraphe précédent. Notre référence ici est l'article de F. Bourdoncle [Bou93]. Comme nous allons le voir, ce choix est également lié au choix d'un ensemble de points d'élargissement, ce qui est relativement intuitif : lorsqu'un élargissement a lieu sur une place, il est souhaitable qu'il soit propagé le plus rapidement possible aux places successeurs.

8.3.1 Ordre topologique faible d'un graphe

En l'absence de cycles dans le graphe, il est clair que les bonnes stratégies sont les ordres linéaires compatible avec l'ordre topologique défini par le graphe. Bourdoncle généralise cette notion dans le cas de graphes cycliques.

Définition 7 (Ordre hiérarchique) *Un ordre hiérarchique sur un ensemble est une permutation bien parenthésée de cet ensemble.*

Un exemple d'ordre hiérarchique sur $\{1, 2, 3, 4, 5\}$ est $1(2(34)5)$. Un ordre hiérarchique sur un ensemble E définit un ordre total $\preceq$ sur E. Les éléments entre deux parenthèses se correspondant forment une *composante* et le premier élément d'une composante est sa *tête*. Pour tout élément $e \in E$, on notera $\omega(e)$ les têtes des composantes imbriquées contenant e.

Définition 8 (Ordre topologique faible) *Un* ordre topologique faible *d'un graphe (dirigé) est un ordre hiérarchique sur ses sommets tel que pour tout arc $u \to v$,*

$$(u \prec v \wedge v \notin \omega(u)) \vee (v \preceq u \wedge v \in \omega(u))$$

Les arcs $u \to v$ tels que $v \preceq u$ sont appelés *arcs de retour*, et leur destination est nécessairement une tête.

Exemple. — L'ordre $1(2(34)5)$ est un ordre topologique faible pour le graphe de la figure 8.3 ; les têtes sont représentées en traits gras. Le graphe contient deux arcs de retour $4 \to 3$ et $5 \to 2$, et par exemple $\omega(4) = \{2, 3\}$. □

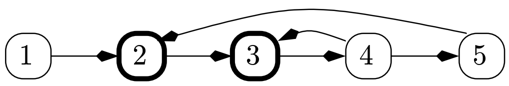


Fig. 8.3 – Exemple de graphe dirigé

À l'exception des arcs de retour, tous les autres arcs du graphe sont compatibles avec l'ordre total $\preceq$ induit par l'ordre topologique faible. Une conséquence de cela est que *tout cycle du graphe contient au moins une tête.*

Les têtes forment donc un ensemble admissible de points d'élargissement, et l'ordre $\preceq$ définit une stratégie d'itération chaotique ayant les bonnes propriétés : à l'exception des arcs de retour, l'ordre de l'itération est compatible avec les arcs du graphe, et les points d'élargissement seront évalué avant leurs successeurs directs dans le graphe. Bourdoncle propose un algorithme astucieux pour calculer un ordre topologique faible d'un graphe, fondé sur une généralisation de l'algorithme de Tarjan de calcul les composantes fortement connexes d'un graphe : il consiste intuitivement à calculer les composantes fortement connexes du graphe, à retirer pour chacune des composantes obtenues son point d'entrée (qui est une tête), et à itérer l'algorithme aux composantes ainsi réduites. L'algorithme obtenu est de complexité $O(\delta.\epsilon)$ dans le pire des cas, où δ est la profondeur d'imbrication maximale des composantes et ϵ le nombre d'arcs. Il est donc très efficace.

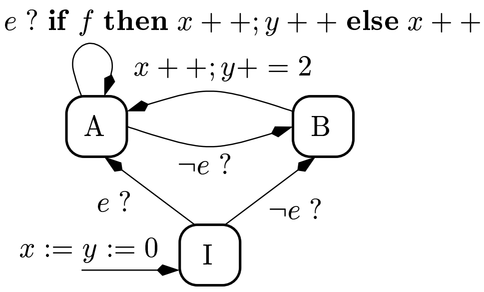
Notons que l'ensemble des points d'élargissement ainsi déterminé n'est pas nécessairement de cardinal minimal, ce qui serait souhaitable puisque l'élargissement perd de l'information, mais que ce problème est de toute manière NP-complet (problème du "FEEDBACK VERTEX SET", [GJ79]) et n'a pas de solution unique.

8.3.2 Stratégie d'itération

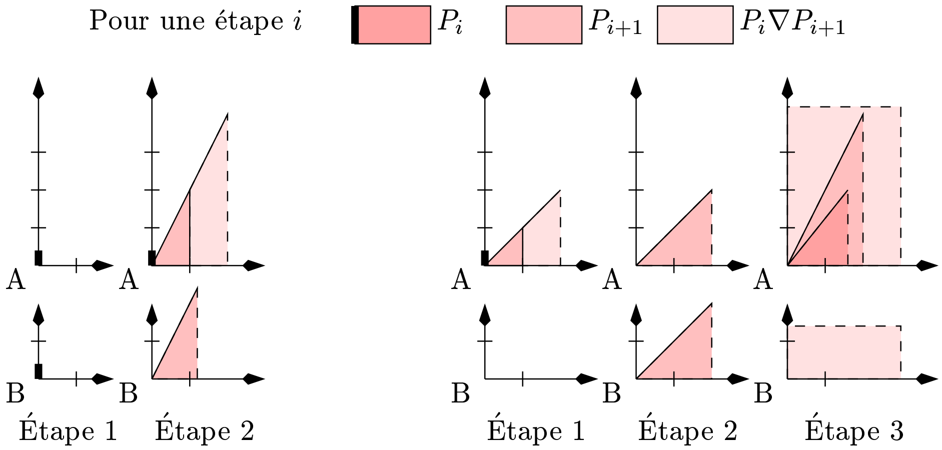
Nous utilisons l'algorithme précédent pour déterminer l'ensemble des points d'élargissement et un ordre topologique faible. Concernant le choix d'une stratégie d'itération étant donné un ordre topologique faible, [Bou93] étudie deux possibilités, illustrées sur l'ordre $1(2(34)5)$:

Stratégie itérative : on peut répéter l'ordre total $\preceq$ jusqu'à stabilisation : $1(2345)^*$;

Stratégie récursive : on peut aussi utiliser la hiérarchie de l'ordre topologique, en essayant de stabiliser en premier les composantes les plus profondes et en dernier la composante principale, et à répéter ce processus : $1(2(34)^*5)^*$.



(a) Automate interprété



(b) Stratégie itérative

(c) Stratégie récursive

FIG. 8.4 – Comparaison des stratégies d'itération pour une analyse en avant

[Bou93] effectue une comparaison de ces stratégies dans un cadre théorique et affirme qu'en général la seconde stratégie est meilleure, en nombre d'itérations et en nombre d'applications de l'élargissement. Étant donné les propriétés du treillis des polyèdres et en particulier de son élargissement, la première méthode est préférable dans notre cas. En effet d'une part les hypothèses sur le treillis abstrait faites dans la comparaison ne sont guère valables pour les polyèdres, et d'autre part l'élargissement sur les polyèdres est d'autant plus précis qu'il prend en compte le plus rapidement possible des informations globales (ce qui n'est pas le cas de l'élargissement sur les intervalles par exemple).

EXEMPLE. — Considérons l'automate interprété de la figure 8.4(a), dans lequel e et f sont des entrées booléennes, et x et y des variables d'état numériques. La boucle sur A consiste à incrémenter x et éventuellement y, tandis que le cycle entre A et B a pour effet d'incrémenter x une fois et y deux fois. Nous allons effectuer une analyse en avant. Avec une stratégie itérative $I(AB)^*$, on obtient les étapes de la figure 8.4(b), alors qu'avec la stratégie récursive $I(A^*B)^*$ celles de la figure 8.4(c), dont le résultat final est moins précis. En effet il faut prendre en compte au plus tôt la croissance dans la direction induite par la transition $B \to A$. □

8.3.3 Application

Analyse en avant. — Pour l'analyse en avant, nous calculons un ordre topologique faible du graphe de contrôle, avec pour plus petits éléments les places initiales, et nous utilisons une stratégie itérative avec élargissement sur les têtes de composantes.

Analyse en arrière. — De manière duale, nous calculons un ordre topologique faible du graphe de contrôle *transposé* (i.e. dont les arcs ont été inversés), avec pour plus petits éléments les places finales, et nous utilisons encore une fois une stratégie itérative avec élargissement sur les têtes de composantes.

Application de l'élargissement. — L'élargissement sur les têtes peut être appliqué avec une fréquence variable. On peut élargir à chaque itération, ou seulement toutes les n itérations. Pratiquement, il est souhaitable d'élargir à chaque itération, sans quoi la propagation des contraintes multiplie dangereusement le nombre de contraintes définissant les polyèdres et la taille de leurs coefficients. Notre outil permet cependant de définir ainsi la fréquence d'application de l'élargissement.

Nous allons aussi voir dans le paragraphe suivant que pour diminuer l'usage de l'élargissement, il est possible de déterminer de manière dynamique les points d'élargissement, en ne prenant en compte que les arcs "actifs" du graphe à l'étape courante.

8.4 Élargissement

L'opérateur d'élargissement que nous utilisons est un raffinement de l'opérateur standard décrit au chapitre 2. En effet l'approximation introduite par cet opérateur standard est parfois trop forte, et peut être réduite dans un certain nombre de cas.

Nous décrivons différentes techniques et raffinements, allant du plus classique au plus original, permettant de diminuer la perte d'information, en général au prix d'une légère augmentation du nombre d'itérations nécessaire à la convergence. Les résultats expérimentaux correspondant seront données au §11.4.2.

8.4.1 Séquence décroissante

Une première possibilité, très simple à mettre en œuvre et proposée par [CC77], est d'effectuer après convergence du calcul normal une ou plusieurs itérations décroissantes, sans élargissement. Cela consiste à effectuer une nouvelle itération du calcul de point fixe, qui est décroissante puisque le point fixe atteint est une surapproximation du plus petit point fixe dans le treillis abstrait. Dans la pratique, en ce qui nous concerne, c'est surtout la première itération décroissante qui est susceptible d'améliorer de façon significative les résultats.

8.4.2 Paramétrage par des contraintes "importantes"

Une seconde possibilité consiste à exploiter les gardes de la fonction de transition pour en quelque sorte *borner* l'élargissement.

Lorsqu'une contrainte subit par exemple une translation vers l'extérieur du polyèdre, entre deux étapes successives, l'opérateur d'élargissement standard l'enlève. L'intuition est qu'elle peut être translatée ainsi une infinité de fois. Ceci introduit souvent une dimension infinie dans le polyèdre, et est peu subtil lorsque l'on a affaire à des compteurs, dont l'incrémentation est gardée par une condition et qui sont donc en général bornés.

On peut alors, et cette notion est très générale, borner l'élargissement : supposons que l'on élargisse P par Q et que les deux valeurs abstraites satisfont une garde g ; il alors souhaitable que le résultat de l'élargissement conserve cette propriété importante, par exemple en prenant $(P\nabla Q) \sqcap g$. Cette idée apparaît entre autre dans [CC92b], dans le contexte de l'analyse d'intervalles, et dans [HPR97], qui l'utilise dans l'analyse de relations linéaires.

Définition 9 (Élargissement paramétré) *Soit A un treillis de prédicats, ∇ un opérateur d'élargissement, et $\mathcal{C} \subset A$ un ensemble fini de prédicats. L'*élargissement paramétré *par $\mathcal{C}$, noté $\nabla_{\mathcal{C}}$, consiste à intersecter $P\nabla Q$ avec tous les prédicats de $\mathcal{C}$ satisfaits à la fois par P et Q :*

$$P\nabla_{\mathcal{C}}Q = (P\nabla Q) \sqcap \Big(\sqcap\{c \in \mathcal{C} \mid P \sqsubseteq c \land Q \sqsubseteq c\}\Big)$$

Théorème 4 *L'opérateur $\nabla_{\mathcal{C}}$ ainsi défini a les propriétés d'un élargissement.*

Preuve: Il est clair que $\forall y_1, y_2 \in A,\ y_1 \sqcup_A y_2 \sqsubseteq_A y_1\nabla_{\mathcal{C}}y_2$: en effet pour tout $c \in A$ tel que $y_1 \sqsubseteq c$ et $y_2 \sqsubseteq c$, $(y_1 \sqcup_A y_2) \sqsubset c$, et donc en notant $C = \sqcap\{c \in \mathcal{C} \mid y_1 \sqsubseteq c \land y_2 \sqsubseteq c\}$

$$y_1 \sqcup_A y_2 \sqsubseteq (y_1 \sqcup_A y_2) \sqcap C \sqsubseteq (y_1\nabla y_2) \sqcap C = y_1\nabla_{\mathcal{C}}y_2$$

Soit maintenant une fonction $G : A \to A$ monotone et une suite croissante $(y_n)_{n\geqslant 0}$ définie par

$$y_0 = \bot \quad , \quad y_{n+1} = y_n\nabla_{\mathcal{C}}G(y_n)$$

La suite décroissante $C_i = \sqcap\{c \in \mathcal{C} \mid y_i \sqsubseteq c \land G(y_i) \sqsubseteq c\}$ converge à partir d'un certain rang I vers $\bar{C}$, puisque $\mathcal{C}$ est un ensemble fini. Donc $\forall i \geqslant I,\ y_{n+1} = y_n\nabla_{\mathcal{C}}G(y_n) = (y_n\nabla y_{n+1}) \sqcap \bar{C}$; ∇ étant un élargissement, la suite $(y_n)_{n\geqslant 0}$ converge donc en temps fini. □

En général, on choisit pour $\mathcal{C}$ les contraintes gardant les opérations numériques, comme l'illustre l'exemple suivant.

Exemple. — Considérons l'automate interprété de la figure 8.5, dont le comportement consiste à incrémenter ou décrémenter le compteur x suivant que l'on se trouve dans l'état I ou D. Les changements d'"états sont commandés par des seuils.

Les deux états de contrôle sont des points d'élargissement. Si on effectue une analyse en avant en utilisant l'élargissement standard, le premier élargissement va supprimer la borne supérieure de x au point I, et supprimer la borne inférieure de x au point D. Par propagation, on va finalement obtenir $reach_I = reach_D = \top$: on n'aura obtenu aucune information. *Une itération décroissante n'améliore pas ces résultats.*

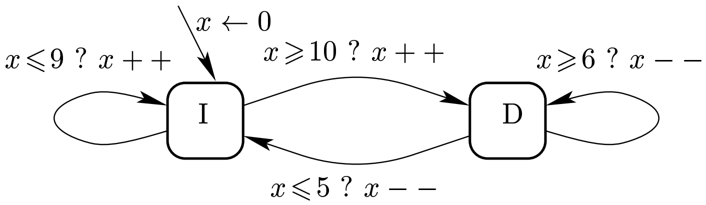


Fig. 8.5 – Incrémentation et décrémentation alternée d'un compteur

En utilisant l'élargissement paramétré, avec $\mathcal{C} = \{x \leqslant 10\}$ au point I et $\mathcal{C} = \{x \geqslant 5\}$ au point D, le premier élargissement donne $reach_I = \{0 \leqslant x \leqslant 10\}$ sur I et $reach_D = 5 \leqslant x \leqslant 11$ sur D, qui est un point fixe. □

On peut remarquer que ce petit exemple est une version simplifiée du régulateur de train du §3.3.1, ce qui signifie que l'analyse du régulateur décrite au chapitre 3, a utilisé l'élargissement paramétré. En général l'utilisation de l'élargissement paramétré rend plus ou moins inutile le calcul d'une séquence décroissante. Notons aussi que la structure de contrôle définit déjà pour chaque place un tel ensemble $\mathcal{C}$ de contraintes, puisque sur une place q avec $def(q) = (B, P)$ on intersecte le résultat de l'élargissement avec P.

Maintenant, comment déterminer automatiquement un "bon" ensemble $\mathcal{C}$ de contraintes, pour une place de la structure de contrôle qui est un point d'élargissement ? Pour traiter le cas d'un compteur x_j dont l'incrémentation $x_j := x_j + c$ est gardée par une contrainte $\sum_i a_i x_i + b \leqslant 0$ avec $a_j > 0$, il suffit d'inclure dans $\mathcal{C}$ la contrainte $\sum_i a_i x_i + b \geqslant a_j c$. On traite le cas des compteurs décrémentés sur le même principe.

8.4.3 Prise en compte de l'activation d'une nouvelle transition

Un autre phénomène est le suivant : l'élargissement standard suppose une croissance régulière de la suite de polyèdres. Or [HPR97] fait la remarque qu'il est un cas évident où cette croissance n'est pas régulière : lorsqu'une transition entrante d'une place, jusqu'alors inactive, devient active car sa source a grossi,

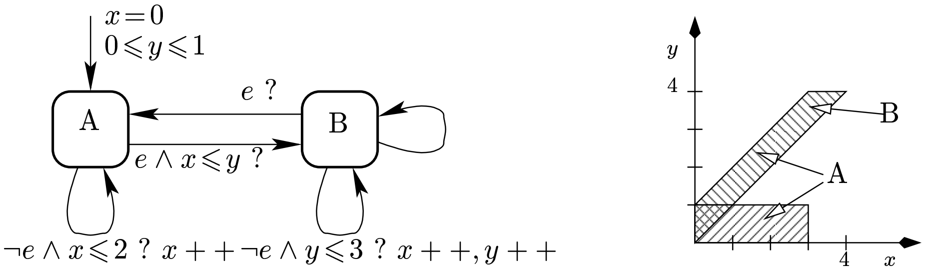


FIG. 8.6 – Automate interprété et graphe des états accessibles pour ses états de contrôle

EXEMPLE. — Considérons l'automate de la figure 8.6. x et y sont deux variables numériques d'état, et e une entrée booléenne. Initialement, dans l'état de contrôle A, suivant l'entrée e, x est incrémenté jusqu'à 3, ou le contrôle passe à B si $x \leqslant y$. Dans B, y peut être incrémenté jusqu'à 4, ou le contrôle peut repasser dans A. Une analyse en avant, avec pour stratégie d'itération $\boxed{\text{A}} \rightarrow \boxed{\text{B}}$ et élargissement correctement paramétré ($\{x \leqslant 3$ pour A et $y \leqslant 4$ pour B), donne les résultats de la figure 8.7. Le problème se produit lors de l'étape 3, lorsqu'on élargit `A` et que la transition $B \rightarrow A$ injecte des valeurs dans une nouvelle direction, figure 8.7 $\square$

Pour pallier ce problème, [HPR97] propose d'élargir le premier polyèdre non vide par le polyèdre courant lorsqu'une transition entrante dans une place est nouvellement activée, ce qui donne l'étape 3' de la figure 8.7 et des résultats plus précis. Nous appelerons ce type d'élargissement *élargissement partiel.*

Cependant, au risque d'augmenter le nombre d'itérations, on peut dans ce cas ne pas élargir du tout ; en effet, si la transition entrante de A injecte un carré $\{0 \leqslant x \leqslant 1, 0 \leqslant y \leqslant 1\}$ au lieu d'une ligne l'élargissement utilisant la première valeur non nulle va redonner le résultat de l'étape 3. En fait, dans le cas particulier des polyèdres convexes, cette heuristique fonctionne bien quand la première valeur non nulle est géométriquement de dimension strictement

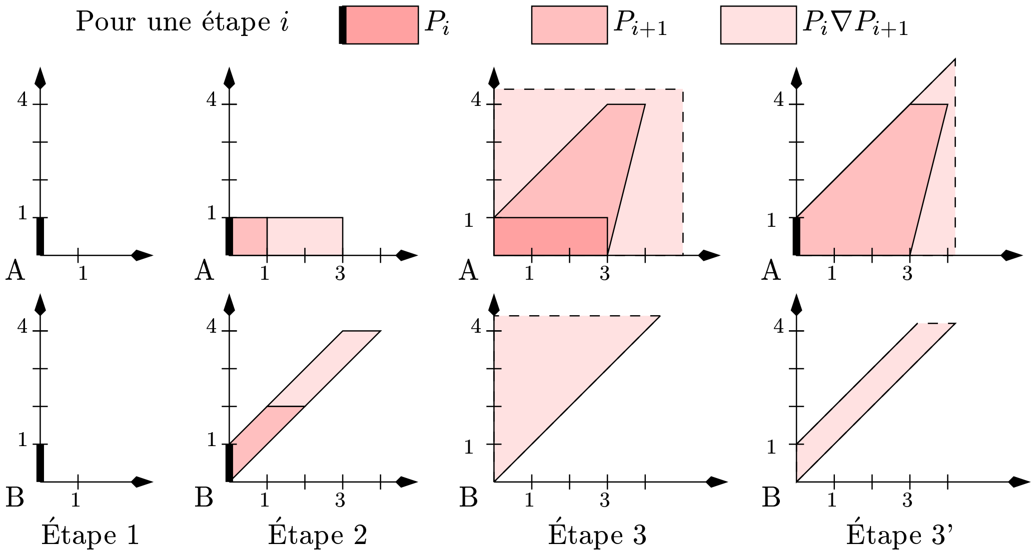


FIG. 8.7 – Analyse de l'exemple

inférieure à la dimension de la valeur courante. Mais la première valeur non nulle d'un point de contrôle n'est pas toujours "petite", car elle peut être le résultat d'un élargissement ayant eu lieu en amont de ce point. C'est pourquoi il est possible de choisir entre la technique de [HPR97] et celle consistant à ne pas élargir du tout.

Par ailleurs, les postconditions entrantes d'une place sont calculées par une séquence ; l'activation d'une branche interne d'une séquence peut également provoquer une irrégularité dans la suite des résultats.

L'outil propose donc les options suivantes :

- l'élargissement partiel peut être
  - soit inactivé ;
  - soit utilisé lorsqu'une transition entrante est activée pour la première fois ;
  - ou enfin utilisé lorsqu'une branche d'une quelconque séquence entrante est nouvellement activée.
- il peut consister soit à élargir la première valeur non nulle par la valeur courante, ou à ne pas élargir du tout (union convexe).

Dans tous les cas, la convergence reste garantie, car l'élargissement spécial ne peut être appliqué en chaque place qu'un nombre fini de fois.

8.4.4 Ensemble dynamique de points d'élargissement

Une dernière possibilité, qui est beaucoup moins classique à notre connaissance, est de déterminer de manière dynamique à chaque itération l'ensemble des points d'élargissement. L'idée est que lors des premières itérations, peu d'arcs du graphe sont *actifs*, c'est-à-dire relient des places dont la partie accessible est non vide ; par conséquent, le nombre de points d'élargissement réellement nécessaires est bien inférieur à celui qui est nécessaire au vu du graphe avec tous ses arcs, actif ou inactifs.

La détermination dynamique des points d'élargissement est en quelque sorte une manière de n'élargir peu ou pas du tout lors des premières itérations.

La mise en œuvre de cette idée se fait en utilisant à chaque itération l'algorithme de Bourdoncle *sur le graphe partiel* formé par l'ensemble des sommets et arcs actifs à l'itération courante, pour déterminer les points d'élargissement. Le coût des calculs effectués lors d'une itération étant largement supérieur au coût de l'algorithme de Bourdoncle, le surcoût de cette technique résulte essentiellement d'une augmentation possible du nombre d'itérations nécessaire pour converger.

La stratégie d'itération, en revanche, doit être globale, et donc on garde celle définie par l'ordre topologique faible sur le graphe entier.

Chapitre 9

Heuristiques de raffinement

Ce chapitre décrit les heuristiques de raffinement, en exposant d'abord ce qu'on peut attendre de la séparation d'une place, ensuite le type de raffinement que nous mettons en œuvre, et enfin les heuristiques que nous avons implémentées.

9.1 Critères de raffinement

Lorsque le cycle d'analyses décrite au chapitre précédent a convergé, et que la propriété n'a pas pu être prouvée, il y a deux possibilités : soit elle est effectivement fausse, soit les approximations effectuées sont trop grossières. Nous supposons bien sûr que nous sommes dans le second cas.

Comme nous l'avons mentionné au chapitre 4, §4.1.1, la principale source d'approximations vient de l'union convexe sur le treillis abstrait, celles introduites par l'opérateur d'élargissement étant en comparaison plus limitées ; le raffinement de la structure de contrôle permet de diminuer ces approximations.

Une heuristique de raffinement relève du compromis : d'une part elle doit être suffisamment efficace pour trouver des critères de division de places, même dans les cas difficiles, d'autre part elle ne doit pas compliquer inutilement la structure de contrôle. Bien qu'un raffinement améliore nécessairement la précision des analyses, certains sont inutiles pour prouver une propriété donnée.

Une première idée pour le raffinement consiste à détecter quelles sont les "mauvaises" unions convexes, et les éviter en raffinant la structure de contrôle. Les §9.1.1, §9.1.2 et §9.1.3 explorent différentes possibilités dans cette direction, qui se distinguent entre elles par le critère définissant les "mauvaise" unions. Cependant, celles-ci ne sont pas facilement exploitables dans notre contexte. Entre autre, nous souhaitons avoir un raffinement qui termine. C'est pourquoi nous proposons une autre approche dans le §9.1.4, qui sépare les états d'une place suivant leur futur, en utilisant des conditions ou contraintes apparaissant dans le source du programme. Cette approche constitue notre heuristique principale, mais nous décrivons également au §9.1.5 une heuristique qui cherche à "convexifier" les assertions.

Mentionnons que le travail qui se rapproche le plus du nôtre, la thèse de H. Wong-Toï [WT94], utilise un raffinement fondé sur la détection des mauvaises unions, et que notre outil repose donc sur un principe différent.

9.1.1 Critères géométriques

On peut chercher à déterminer des critères géométriques, nous indiquant si une union convexe perd plus ou moins d'information. Plaçons-nous dans un premier temps dans le treillis des polyèdres, dans lequel l'approximation est uniquement due à l'enveloppe convexe, et non à la perte des relations liant booléens et contraintes numériques

Utilisation d'une mesure

Une idée assez immédiate est d'utiliser une *mesure*, au sens mathématique du terme, pour comparer la taille de l'enveloppe convexe à la somme des tailles des opérandes. Si les deux mesures sont très différentes, cela signifie que l'enveloppe convexe a ajouté beaucoup d'éléments à l'union ensembliste, et qu'elle engendre donc une perte d'information importante. Il s'avère cependant qu'une mesure adaptée est difficile à trouver.

Tout d'abord, les polyèdres que nous manipulons ne sont pas bornés. Le volume ne convient donc pas. Une idée pour surmonter cette difficulté est d'utiliser des angles solides : les polyèdres de $\mathbb{Q}^n$ que nous manipulons peuvent être vus comme des cônes dans l'espace vectoriel $\mathbb{Q}^{n+1}$;[1] on pourrait donc utiliser comme mesure sur les polyèdres l'*angle solide* de son cône correspondant. Dans un espace vectoriel euclidien de dimension n, l'*angle solide* d'un cône est la surface de la portion de la sphère centrée sur l'origine et de rayon 1 qui se trouve à l'intérieur du cône. Cette mesure est finie, puisque la surface d'une telle sphère est $(2\pi)^{n-1}$. Notons cependant que le fait que deux polyèdres bornés soient de même volume n'implique pas que l'angle solide des cônes correspondants soit identique.

Ce premier problème surmonté, s'en pose ensuite un second : la dimension des polyèdres — ou des cônes correspondants, n'est pas toujours égale à la dimension de l'espace dans lesquels ils sont définis. Comment comparer, dans un espace de dimension 3 par exemple, une droite infinie, un quart de plan, et un petit hypercube ? Formellement, les deux premiers ensembles sont de volume nul. Il faudrait donc utiliser une mesure qui n'en serait pas une au sens mathématique.

Enfin, il faut savoir que les calculs de volumes ou de surface dans un espace de dimension n sont extrêmement coûteux. Concernant le calcul de volume, [BEF98] cite des complexité du type $O(n^k n!)$ ou $O(n^k k'^n)$, avec k et k' des constantes, pour des polyèdres qui sont topologiquement équivalents à des hypercubes. Nous n'avons pas trouvé dans la littérature d'algorithme calculant des angles solides de cônes en dimension n, mais on peut s'attendre à des complexités au moins équivalentes, puisqu'un angle solide en dimension n est un volume en dimension $n-1$, dans un espace plus compliqué !

Nous avons donc éliminé cette piste dans le cas des polyèdres. Notons cependant que si l'on utilise un treillis plus simple, comme celui des intervalles et peut-être celui des zones, il est possible de trouver une mesure convenable.

Égalités et directions infinies

Une autre idée, d'ambition plus modeste, est de repérer un certain nombre de cas particuliers pour lesquels il est évident que l'enveloppe convexe engendre une grosse approximation.

[1] cette transformation classique est décrite dans [Wil93]

(a) Perte d'égalités

(b) Gain de droites

FIG. 9.1 – Égalités et droites

Lorsqu'un polyèdre est sous forme minimale, on connaît le nombre d'égalités indépendantes qu'il satisfait ainsi que le nombre de droites indépendantes qu'il contient. Ceci est également vrai si l'on manipule des intervalles ou des zones.

Lorsqu'on calcule l'enveloppe convexe de deux polyèdres, chacun satisfaisant une égalité, et que le résultat n'en satisfait aucune, il y a clairement perte d'information, figure 9.1(a). Il en est de même lorsque chaque polyèdre contient une droite, et que le résultat en contient deux, figure 9.1(b). Ceci arrive en fait lorsque les deux égalités ou les deux droites sont indépendantes. La généralisation de ce critère est le suivant :

Définition 10 *Soit $(P_i)_{1\leqslant i\leqslant n}$ des polyèdres dont le nombre d'égalités et de droites est respectivement e_i et d_i. L'enveloppe convexe $\sqcup_i P_i$, satisfaisant e égalités et contenant d droites, est dite* mauvaise *si l'une des deux conditions suivantes est vérifiée :*

1. $e < \min(e_1, \ldots, e_n)$
2. $d > \max(d_1, \ldots, d_n)$

Cette définition se transpose aux intervalles ou aux zones.

Ce critère, qui repère les grosses pertes d'information, est particulièrement économique à mettre en œuvre. Toutefois, pour une raison qui sera expliquée au §9.1.4, il ne sera utilisé que pour les enveloppes convexes effectuées à l'intérieur des séquences lors d'un calcul arrière.

Prise en compte de l'aspect booléen ?

Nous nous plaçons à nouveau dans le treillis $\mathbb{B}^n \times \mathcal{P}(\mathcal{N}^p)$. La différence ensembliste de l'union convexe et de l'union ensembliste de (B_1, P_1) et (B_2, P_2) est

$$\Big(B_1 \setminus B_2, P_2 \setminus P_1\Big) \;\cup\; \Big((B_2 \setminus B_1, P_1 \setminus P_2\Big) \;\cup\; \Big((B_1 \vee B_2, (P_1 \sqcup P_2) \setminus (P_1 \cup P_2)\Big)$$

Nous avons examiné dans les paragraphes précédents quelles étaient les possibilités pour évaluer la taille du troisième terme, du moins de sa partie numérique.

Le premier terme (et de manière symétrique le second) est d'autant plus important que $B_1 \setminus B_2$ et $P_2 \setminus P_1$ le sont. Un critère pour juger ce terme trop important pourrait être le suivant :

1. $P_1 \sqcap P_2 = \emptyset$ et la dimension de P_2 est importante (i.e., $P2$ satisfait peu d'égalités).
2. $B_1 \setminus B_2$ a une grosse "mesure"

$\mathbb{B}^n$ étant un ensemble fini, la mesure canonique d'un sous-ensemble est de compter ses éléments, ce qui revient pour une formule à compter le nombre de monômes complets qui l'impliquent. Cette opération est aisément implémentable dans une bibliothèque de BDDs

(La bibliothèque CUDD [Som] que nous utilisons la propose), et ne pose pas de problème algorithmique.

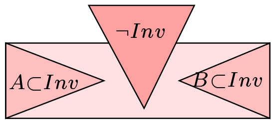
Cependant, cette mesure sur les formules booléennes n'est pas forcément pertinente, car beaucoup d'états booléens sont équivalents. C'est pourquoi nous n'avons finalement pas utilisé de critères portant sur la partie booléenne des états convexes.

Conclusion

L'idée d'utiliser une mesure pour déterminer la perte d'information causée par une union convexe, bien que séduisante, n'est pas facile à mettre en œuvre. Dans notre cas, il est clair que le seul critère efficace, qui est fondé sur la perte d'égalités ou de droites, ne peut suffire à lui seul à raffiner une structure de contrôle, comme par exemple celle analysée au chapitre précédent.

Par ailleurs, un grand défaut de ce principe est qu'il ne tient pas compte de la propriété à prouver — ou alors de manière très indirecte, du fait que l'espace d'état considéré est l'espace des états dangereux.

9.1.2 Critères fondés directement sur la propriété



Un second type de critère permettant de détecter les mauvaises bornes supérieures consiste à comparer les relations des opérandes et du résultat avec la propriété. Typiquement, si deux valeurs abstraites ne violent pas la propriété, mais que leur union convexe la viole, il est clair qu'il faut l'éviter, figure ci-contre.

Dans notre contexte, cette situation est impossible, puisque la structure de contrôle sépare déjà de manière exacte les états satisfaisant la propriété de ceux la violant, et les unions convexes sur les places sont effectuées après intersection avec leur définition.

Bien sûr, les états qui mènent *à coup sûr* à une violation de la propriété doivent aussi être évités, mais nous ne sommes pas capable de détecter ces états, puisque les analyses arrières sont surapproximées.

Par l'utilisation de *sous-approximations*, [WT94] est en revanche capable d'utiliser ce critère. Après avoir effectué une analyse arrière sous-approximée, lors d'une analyse avant sur-approximée, il peut utiliser le résultat de l'analyse précédente pour repérer les mauvaises unions convexes ; l'union convexe de P et Q doit alors être évitée lorsque :

1. L'une des deux valeurs abstraites contient un état menant à une violation de la propriété, tandis que l'autre n'intersecte pas la sous-approximation des états menant à une violation.
2. Les deux valeurs abstraites n'intersecte pas la sous-approximation des états menant à une violation, mais leur union convexe oui.

Le premier critère sert à différencier les valeurs abstraites suivant leur comportement vis-à-vis de la propriété, tandis que le second détecte les unions convexes abusives. Notons que ces critères restent assez heuristiques dans la mesure où l'on ne connaît qu'une sous-approximation des états menant à une violation, et qu'il considère une union binaire : lors d'une union n-aire, la décomposition en union explicite dépendra de l'ordre de prise en compte des n valeurs abstraites. [WT94] ne fait aucune mention de ce problème.

Une autre caractéristique de ce critère est qu'il est utilisé *pendant* l'analyse, et non a-posteriori. Ceci est une qualité importante : en effet lorsque les approximations effectuées par une analyse classique sont grossières, il se peut qu'il ne soit plus possible de diagnostiquer le problème après convergence.

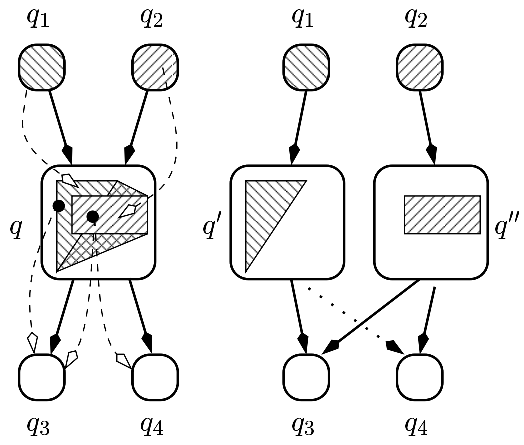
Ce critère constitue la principale technique de raffinement de [WT94].

Nous ne l'avons pas expérimenté dans notre contexte, le problème principal étant la mise en œuvre d'une analyse sous-approximée. En effet, si l'on considère uniquement les états numériques, dans le cas des automates temporisés les ensembles d'état dont on cherche une approximation sont *denses*, alors que les nôtres sont *discrets*. La sous-approximation d'ensembles discrets est beaucoup plus difficile et en tout cas ne peut pas se faire avec des polyèdres.

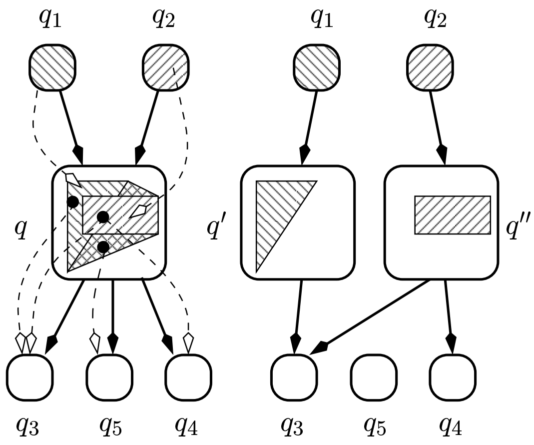
Par ailleurs, même dans le cas d'ensembles denses, il est finalement plus facile de sur-approximer que de sous-approximer avec précision. L'opérateur d'union sous-approximé de Wong-Toï est relativement grossier, ce qu'il reconnaît lui-même : hormis quelques cas particuliers, il consiste à choisir une des deux valeurs abstraites dont on veut calculer l'union. De ce fait, l'analyse sous-approximée construit essentiellement une trace contre-exemple, qui est "un peu symbolique".

9.1.3 Critères fondés sur la suppression de transitions

Un troisième type de critère pour détecter les unions convexes à éviter consiste à changer de point de vue : au lieu de chercher à réduire l'espace des états dangereux, on peut essayer de déconnecter les places initiales des places finales, et donc considérer que l'objet du raffinement est de supprimer des arcs sur le graphe abstrait.



(a) Division de q d'après les prédécesseurs et suppression d'une transition



(b) Division de q d'après les prédécesseurs et suppression d'une place

FIG. 9.2 – Suppression de transition et de place

Examinons la figure 9.2(a), qui représente une portion d'une structure de contrôle. Nous appellerons dans ce paragraphe *contribution* de q_i à q l'intersection de la postcondition de la définition de q_i avec la définition de q. On suppose ici que la contribution de q_1 ne peut mener qu'à q_3, tandis que celle de q_2 mène à q_3 et q_4. Si on divise la place q suivant les prédécesseurs

q_1 et q_2 comme sur la figure, on retarde l'union convexe des contributions, et on détecte que le chemin $q_1 \rightsquigarrow q \rightsquigarrow q_4$ était en fait impossible dans le graphe non raffiné. Et par ailleurs le gain de précision du à la suppression de l'enveloppe convexe se propage à q_3 et q_4.

Une telle division peut avoir des conséquences supplémentaires. Prenons le cas de la figure 9.2(b), identique à la précédente hormis le fait que q a un troisième successeur q_5, et que ceci est uniquement du aux états ajoutés par l'union convexe des contributions de q_1 et de q_2. La division de q a alors pour conséquence de rendre q_5 inaccessible.

Notons que le critère de division qui vient d'être décrit est facile à mettre en œuvre dans le cas d'une union binaire, mais devient plus délicat si l'union est n-aire et que l'on ne veut pas forcément diviser une place en n nouvelles places, mais en un nombre plus réduit. Par ailleurs, le nombre de test et de calculs de postcondition à effectuer est est en $O(p.s)$, où p est le nombre de prédécesseurs et s le nombre de successeurs de la place examinée.

Nous avons tenté au début de ce travail d'utiliser ce critère, avant d'expérimenter celui présenté au paragraphe suivant. Un des problèmes rencontrés, entre autre, réside dans le fait que la structure de contrôle initiale est trop grossière pour que cette technique réussisse à effectuer les premiers raffinements.

9.1.4 Critères fondés sur les conditions discriminantes

Les critère exposés précédemment peuvent mener à des raffinements infinis. En particulier, si le programme contient un compteur x, nous voulons absolument éviter que les heuristiques de raffinement se mettent à diviser une place suivant que $x = 0$, $x = 1$, etc. . . .

Par ailleurs les raffinement obtenus peuvent être en quelque sorte redondants : en effet ils permettent de recouvrir l'espace d'états par les définitions de places, mais non de le partitionner : un même état peut être pris en compte dans plusieurs places ; ceci empêche la terminaison du raffinement et engendre des calculs redondants.

C'est pour éviter ces phénomènes que nous avons fait le choix suivant :

Choix 1 *nous ne diviserons les places que sur des conditions apparaissant dans le source du programme* ; notons que comme assertion et propriété font partie du programme, les conditions qui y apparaissent constituent également des conditions de division admissibles ;

Une conséquence de ce choix est qu'il existe une structure de contrôle la plus précise possible. Notons que celle-ci est très fine, puisqu'elle est obtenue par évaluation partielle de toutes les variables booléennes, et séparation subséquente de l'espace numérique par toutes les contraintes du programme. Par conséquent cette structure de contrôle est très grosse et induit une précision maximale de nos analyses qui n'est absolument pas rédhibitoire pour la preuve d'une propriété "raisonnable".

Ceci posé, il s'agit maintenant de déterminer quelles sont les "bonnes" conditions de division. Intuitivement ces conditions, que nous appellerons *conditions discriminantes*, sont celles qui séparent des comportement différents, en particulier vis-à-vis de la propriété. Considérons un arc du graphe abstrait $q \rightsquigarrow q_e$, avec q_e une place erronée, figure 9.3. Les états concrets satisfaisant l'assertion $ast(q \rightsquigarrow q_e)$ peuvent mener en une seule étape à la violation de la propriété, tandis que les autres nécessiteront plusieurs étapes. Il apparaît judicieux de séparer ces deux types d'états dans la structure de contrôle, de manière à ce qu'elle reflète le fait que les états ne satisfaisant pas $ast(q \rightsquigarrow q_e)$, représentés par la place q', ne peuvent mener en une seule étape à q_e. En quelque sorte, *le raffinement de place s'effectue via un raffinement de*

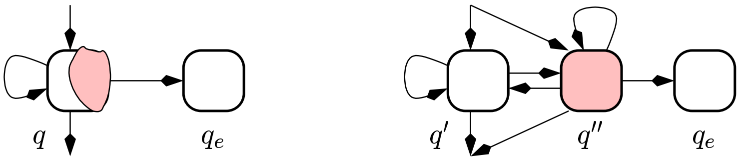


FIG. 9.3 – Principe du raffinement par stabilisation

transition. Il est important de noter que les états satisfaisant l'assertion, représentés par q'', ne mènent pas à coup sûr à une violation de la propriété : d'une part l'assertion est une condition nécessaire mais non suffisante, d'autre part ils peuvent apparaître comme tels uniquement à cause des approximations dues aux bornes supérieures sur les places et dans les séquences. Bien sûr, lors d'une telle division, on espère aussi supprimer d'autres transitions q' et q''.

EXEMPLE. — Reprenons le programme d'exemple du chapitre 4, page 33, pour lequel nous avons construit une structure de contrôle initiale et que nous avons analysé au chapitre précédent (§8.1 et §8.2), obtenant les résultats de la figure 9.4(a). Rappelons que dans ce programme, lorsque $b0$ et $b1$ sont égaux x est incrémenté, sinon y l'est, et que ces incrémentations s'alternent. Les assertions d'arcs sur la figure 9.4(a) nous indiquent que lorsque la condition du programme $x \geqslant y$ est satisfaite, l'arc $2 \rightsquigarrow 2$ est emprunté, tandis que si $x \leqslant y-1$ est vérifié, la relation de transition mène à la place erronée 3. Divisons donc la place 2 en deux places $2'$ et $2''$ suivant la satisfaction de $x \geqslant y$:

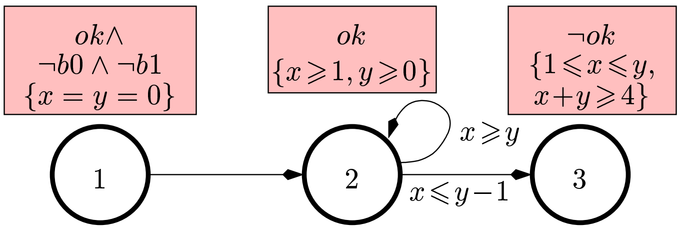
$$def(2') = def(2) \sqcap \{x \geqslant y\} \quad , \quad def(2'') = def(2) \sqcap \{x \leqslant y-1\}$$

La structure obtenue est montrée figure 9.4(b) : non seulement l'arc $2' \rightsquigarrow 3$ a été supprimé, mais une analyse statique a permis de supprimer aussi les arcs $1 \rightsquigarrow 2'', 2'' \rightsquigarrow 2', 2'' \rightsquigarrow 2''$. Une analyse dynamique en avant permet ensuite d'obtenir les informations supplémentaires données dans les ovales. La figure 9.4(c) montre la structure de contrôle obtenue après renforcement des définitions. Une analyse dynamique en arrière donne les résultats dans les ovales et permet la preuve de la propriété. Le succès de la preuve est dû au fait que lors du calcul arrière, la précondition de la contribution de la place $2''$ à la place $2'$ n'intersecte pas la définition de $2'$: par conséquent la boucle sur $2'$ devient inactive, et en l'absence de propagation par cette boucle la précondition de la place $2'$ n'intersecte plus la définition de la place initiale 1. □

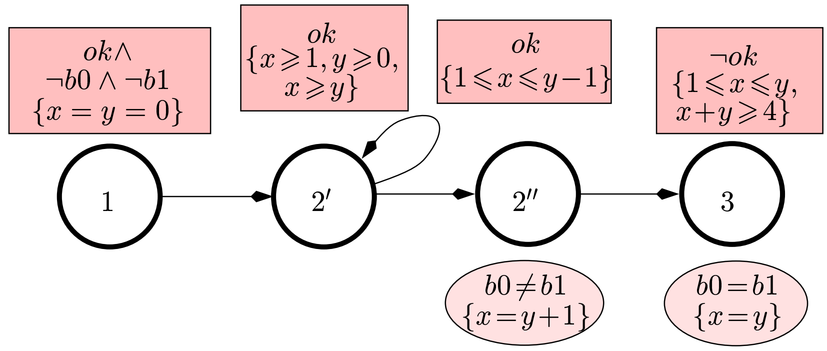
Cet exemple, bien que trivial, est éclairant. Intuitivement, nous voulions séparer les états booléens suivant que $b0 = b1$ ou $b \neq b1$, cf. §4.1.2. C'est en fait ce que nous avons obtenu, mais en divisant sur la condition $x \geqslant y$. Il se révèle ainsi que les conditions séparant des comportement différents permet, après les analyse subséquentes, d'obtenir la partition intuitivement souhaitée.

Une caractéristique, et peut-être un inconvénient de cette technique, est qu'on ne raffine pas en cours d'analyse, mais seulement après convergence ou dès que la vérification est montrée impossible.

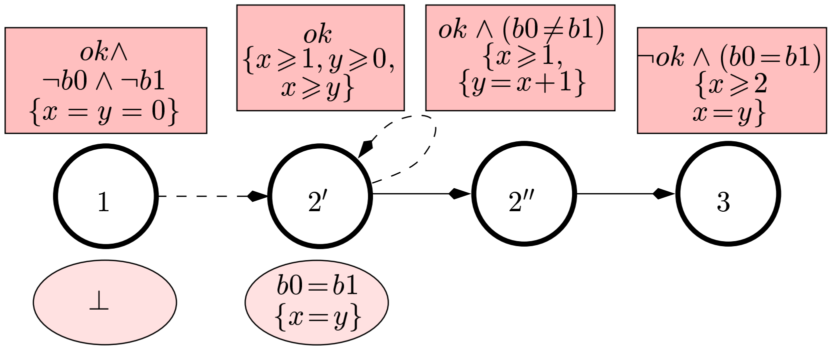
Mentionnons que [SUM96], que nous déjà cité au §4.5 et qui propose une méthode de tableau, dans laquelle le tableau courant représente l'ensemble des contre-modèles de la propriété à montrer et est progressivement raffiné, utilise une heuristique de raffinement fondée sur le même principe : elle consiste à séparer un sommet du tableau suivant la postcondition d'un arc entrant ou la précondition d'un arc sortant. Le fait que les formules manipulées



(a) Résultat des analyses sur la structure de contrôle initiale



(b) Division de la place 2, analyse statique, puis dynamique en avant (ovales)



(c) Analyse dynamique en arrière et succès

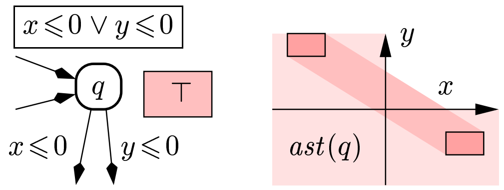
FIG. 9.4 – Division et analyses avant et arrière sur le petit exemple

peuvent être complémentées plus facilement que nos valeurs abstraites, et que par conséquent les sommets peuvent être divisés sur des conditions arbitraires, rende la mise en œuvre de leur heuristique plus simple que la nôtre.

9.1.5 Assertions de place non convexes

La définition des places est certes un état convexe, mais son assertion de place (quantifiée sur les entrées) est une formule non nécessairement convexe. Lorsque ce cas se présente, cela signifie que l'on dispose d'une information relativement précise mais qui ne peut être représentée par une valeur abstraite, et qui par conséquent ne peut être complètement propagée.

Exemple. — Considérons la figure ci-contre : la place q a deux arcs sortants, étiquetés par leur assertion. L'assertion de place, qui est l'union des assertions des arcs sortants, est une formule non convexe. Le graphique illustre le phénomène : imaginons que les deux petits rectangles soit les préconditions des deux successeurs. Leur borne supérieure engendre une approximation importante, et en particulier inclut des valuations violant l'assertion, qui seront ensuite propagées lors d'un calcul en arrière. Dans ce cas, une division de la place q suivant une des contraintes permet d'éviter ce phénomène. □



Il est important d'avoir à l'esprit que la non-convexité d'une formule peut aussi bien être due à des relations entre booléens et contraintes qu'à des disjonctions de contraintes. Une heuristique cherchant à éviter les assertions non convexes prend donc aussi bien en compte l'aspect booléen que l'aspect numérique.

Nous utiliserons ce critère pour diviser des places en second recours, en cas d'échec de notre heuristique principale.

9.1.6 Plan du chapitre

Notre heuristique principale est celle fondée sur les conditions discriminantes, qui séparent les comportements différents vis-à-vis de la propriété. L'exemple illustratif est bien sûr particulièrement simple. Les paragraphes suivants vont développer la mise en œuvre de cette idée dans des cas plus compliqués. Le §9.2 décrit le choix des places à diviser et des conditions de division sur une structure de contrôle générale. Le §9.3 décrit ensuite les heuristiques pour extraire une condition de division adéquate à partir de conditions plus générales. Le §9.4 explique le processus de la division d'une place sur une condition binaire. Précisons que les principes qui s'appliquent lorsqu'on raffine une structure de contrôle sur les *postconditions* des transitions (vue "en arrière") sont identiques. Ces deux heuristiques sont nos heuristiques principales

La division d'une place implique certes un raffinement des séquences associés aux arcs entrants et sortants. Cependant des approximations importantes peuvent se produire dans les séquences, ce qui nécessite des heuristiques spécialisées, abordée au §9.5. Enfin, en cas d'échec des heuristiques principales, nous utilisons au §9.6 l'heuristique secondaire cherchant à rendre les assertions de place convexes.

Enfin, le §9.7 explique comment nous combinons les trois heuristiques, à savoir division sur les conditions discriminantes (en avant et en arrière), raffinement de séquence et "convexification" des assertions de place, et le §9.8 conclut le chapitre.

9.2 Choix des places et des arcs à raffiner

Dans ce paragraphe nous abordons le problème suivant : étant donné une structure de contrôle analysée sans succès, comment choisir les places et les arcs à raffiner ?

Nous allons d'abord présenter notre solution dans le cas où on considère une vue "en avant" du programme et où on compare les *préconditions* des transitions pour séparer les comportement différents. L'intérêt de considérer les préconditions est que celles-ci sont connues de manière assez fine par les *assertions d'arc* qui sont des *formules*, alors que les postconditions ne sont connues que sous forme d'*état convexe*. La mise en œuvre des idées qui suivent "en arrière" est expliquée au §9.2.4.

9.2.1 Principe

La généralisation du §9.1.4 consisterait à examiner tous les prédécesseurs directs des places erronées, et pour chacune de ces places, à partir de l'assertion de l'arc menant à une place erronée évaluée partiellement sur l'assertion de la place, d'extraire une condition discriminante. Mais ceci est insuffisant pour les raisons suivantes :

1. si on veut couper les chemins menant des places initiales aux places finales, tous les arcs se trouvant sur ces chemins sont dangereux et sont candidats pour être raffinés, et pas seulement ceux qui mènent directement aux places erronées ;
2. les approximations les plus pénalisantes n'ont pas forcément lieu juste en amont des places erronées, mais aussi dans les boucles internes du graphe. Celles-ci, par propagation, se répercutent ensuite dans tout le graphe.

En première approximation, on pourrait d'abord essayer de couper les arcs reliant les composantes connexes du graphe, puis ensuite ceux appartenant aux cycles du graphe. Mais on peut mieux structurer la décomposition du graphe en utilisant encore une fois l'algorithme de décomposition d'un graphe dirigé en sous-composantes fortement connexes de Bourdoncle, déjà utilisé pour la séquentialisation et la détermination des points d'élargissement.

Considérons la figure 9.5. Une décomposition en sous-composantes fortement connexes est donnée par l'ordre topologique faible $q_i \Big(q_1 \big(q_3 (q_4) \big) \big(q_5 q_6 \big) \big(q_2 \big) \Big) q_e$ (c.f. §8.3.1).

On peut d'abord essayer de couper les arcs menant des places de la composante de profondeur 1 à q_e. Ceci revient à diviser les places q_3 et q_6 resp. sur les assertions $ast(q_3 \rightsquigarrow q_e)$ et $ast(q_6 \rightsquigarrow q_e)$. On peut aussi se placer dans la composante de profondeur 1, et raffiner les arcs reliant ses éléments ou ses sous-composantes de profondeur 2 ; par exemple on pourrait diviser q_2 suivant $ast(q_2 \rightsquigarrow q_1)$, ou q_4 suivant $ast(q_4 \rightsquigarrow q_5) \vee ast(q_4 \rightsquigarrow q_6)$. Si on échoue, parce que les conditions obtenues ne permettent pas de diviser les places, on peut raffiner les boucles plus internes en se plaçant dans les composantes de profondeur 2 ; par exemple on peut essayer de diviser q_2 suivant $ast(q_2 \rightsquigarrow q_2)$ ...

Bien sûr, comme la décomposition en sous-composantes n'est pas unique, il y a une certaine dose de hasard.

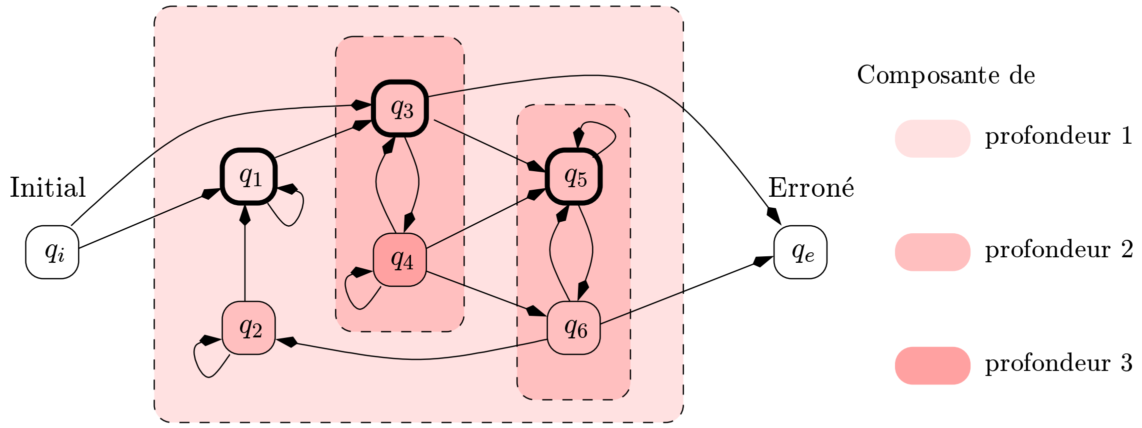


FIG. 9.5 – Décomposition du graphe en sous-composantes connexes

Maintenant, faut-il raffiner en priorité les boucles les plus internes ou les arcs reliant les composantes connexes ? Faut-il effectuer un raffinement à la fois, ou plusieurs ? Le fait est qu'un raffinement suivi d'un cycle d'analyses peut supprimer la nécessité apparente d'autres raffinements.

9.2.2 Choix généraux

Ce paragraphe répond aux questions précédentes. D'abord *nous ne diviserons pas les places initiales ou erronées.* En effet ces places sont sources ou puits et donc ne propagent pas d'information au cours d'une analyse entre des places prédécesseurs et des places successeurs. Du même coup les enveloppes convexes qui y prennent place lors d'une analyse, avant ou arrière, ne sont pas pénalisantes pour l'analyse duale, grâce au mécanisme des micro-états des séquences qui jouent le rôle de filtres.

Ensuite nous effectuerons plusieurs divisions entre deux cycles d'analyse. Le fait de n'effectuer qu'une division à la fois minimise certes la taille de la structure de contrôle mais multiplie le nombre d'analyses, et notre objectif final est de prouver la propriété en un temps raisonnable. La philosophie consistant à générer la structure de contrôle la plus petite possible permettant la preuve de la propriété n'est qu'un moyen pour y parvenir.

Ceci a aussi le mérite d'éliminer le problème de déterminer quelle place diviser entre les analyses, et d'élaborer une stratégie équitable pour ne pas raffiner toujours la même partie de la structure de contrôle. En fait, *nous allons essayer, entre chaque cycle d'analyses, de diviser chaque place sur une condition.* Toutes les places ne seront pas nécessairement divisées, car le principe de division esquissé ci-dessus n'est pas toujours capable d'extraire une condition de division pour chaque place.

Enfin, nous allons privilégier le raffinement des arcs reliant les composantes connexes au raffinement des boucles les plus internes du graphe. Il convient d'observer que cela ne signifie pas que les boucles internes ne seront raffinées qu'en dernière extrémité. Typiquement, si les assertions de deux arcs sont la négation l'une de l'autre, diviser suivant l'une ou l'autre produit le même résultat.

Algorithme 9.1 : Choix des places et des conditions de division

```
Input: Une structure de contrôle G
Output: Un ensemble de couples (place, condition de division adéquat)
begin
  scfc1 := Scfc(G);
  scfc := TransformeScfc(scfc1);
  traités := ∅; couples := ∅;
  for i=0 to Profondeur(scfc)−1 do begin
    for C_ext in Composantes(i) do begin
      { Pour chaque composante extérieure de profondeur i }
      for C_int ∈ C_ext do begin
        { Pour chaque élément de la composante extérieure }
        for q ∈ C_int\traités do begin
          { Pour chaque place de la composante interne pas encore divisé e }
          f := ⋁_{q'∈(C_ext\C_int)} ast(q ⇝ q')
          { f = condition pour quitter la composante interne tout en restant dans
          la composante externe }
          if f est une condition qui sépare def(q) then
            traites := traites∪q; couples := couples∪{(q, f)};
        end
      end
    end
  end
  return couples
end
```

9.2.3 Algorithme en avant

L'algorithme 9.1 correspondant au principe et aux choix décrits ci-dessus. Un dernier problème technique à résoudre est le suivant : les têtes des composantes non réduites à un élément ne forment jamais à elle seule une sous-composante, même lorsqu'un arc boucle sur elle. Par conséquent, sur l'exemple précédent, l'algorithme pourra diviser q_4 sur sa boucle, mais pas q_5. C'est pourquoi on transforme la décomposition précédente du graphe $q_i\Big(q_1\big(q_3(q_4)\big)\big(q_5q_6\big)\big(q_2\big)\Big)q_e$ en $q_i\Big(\big(q_1\big)\big(q_3(q_4)\big)\big((q_5)q_6\big)\big(q_2\big)\Big)q_e$.

9.2.4 Alogrithme en arrière

Dans les paragraphes précédents, nous avons cherché à diviser les places en trouvant des conditions discriminantes "en avant", ces conditions étant des disjonctions d'assertions d'arc.

L'algorithme précédent peut aussi s'implémenter en "arrière". Nous n'avons cependant pas l'équivalent des assertions d'arc en arrière, nous connaissons que les postconditions des arcs, qui sont des états convexes. Or, rappelons-le, nous avons fait le choix de ne diviser les places que sur des conditions apparaissant dans le programme. Une première possibilité est de ne considérer que la partie booléenne des postconditions, ce qui permet d'obtenir des conditions discriminantes booléennes. On peut cependant faire mieux, en approchant le polyèdre d'un état convexe par un monôme sur des contraintes du programme, comme on l'a décrit au §6.1.4. On dispose ainsi d'une approximation des postconditions des arcs par des formules convexes.

On peut donc réutiliser les principes des deux paragraphes précédents en arrière, et nous avons implémenté dans notre outil le raffinement sur les conditions discriminantes en arrière. En général, les conditions discriminantes en arrière sont moins précises que celles en avant, puisqu'elles correspondent à des disjonctions de formules convexes, alors que les conditions discriminantes en avant sont des disjonctions de formules générales.

9.3 Division d'une place par séparation d'un groupe d'arcs

Nous expliquons en détail dans ce paragraphe l'opération de division de place. Le problème est le suivant : étant donné un arc, ou un ensemble d'arcs issus de la même place, extraire une condition de division *adéquate*. La partie difficile est l'extraction d'une condition de division adéquate à partir d'une condition discriminante plus générale.

Nous définissons d'abord au §9.3.1 quelles sont les conditions de division adéquates, et détaillons les phénomènes à prendre en compte sur un exemple. Le §9.3.2 explique ensuite l'extraction d'une condition de division adéquate à partir d'une condition plus générale,

9.3.1 Définition des conditions adéquates

Outre la décision de ne diviser les places que sur des conditions du programme, un second choix essentiel est le suivant :

Choix 2 *les places ne pourront être divisées que sur une condition binaire d'état.* Une condition *binaire* est une formule telle qu'elle et sa négation correspondent chacune à un unique état convexe. Par ailleurs, la condition ne doit porter que sur les *états* du programme : nous n'allons pas distinguer les places suivant les entrées admissibles dans chacune d'elle. L'ensemble des conditions de division est donc constitué

- des formules sur les variables d'états booléennes ;
- et des contraintes d'inégalités sur les variables d'état numériques qui apparaissent dans le programme[2].

La restriction à des conditions binaires mérite de plus amples justifications. Supposons que nous divisions une place q avec $def(q) = \top$ suivant la formule $x \geqslant 0 \wedge y \geqslant 0$. La négation de cette formule n'est pas convexe.

1. Une première solution est de décomposer la formule et sa négation en union de formules convexes, et pour cet exemple de diviser q en *trois* nouvelles places. Dans les exemples réels cela multiplierait trop le nombre de places, et le processus de division deviendrait coûteux (il faut potentiellement dupliquer toutes les transitions sortantes et entrantes).
2. Une seconde solution est d'approximer la formule non convexe par une valeur abstraite la contenant : nous obtenons alors la situation de la figure 9.6, dans laquelle les assertions de place sont données par les formules encadrées. La définition de q'' englobe tous les

[2] nous allons voir au paragraphe suivant qu'il faut ajouter à cet ensemble les contraintes d'état provenant de l'élimination des entrées numériques

états ; certes, les états ne satisfaisant pas $ast(q'')$ mais inclus dans $def(q'')$ ne pourront pas "sortir" de q'' par un calcul de postcondition, grâce au mécanisme des assertions, mais les unions convexes sur la place q'' vont tout de même faire perdre beaucoup d'information, comme le montre le graphe de la figure. C'est pourquoi nous préférons nous restreindre à des conditions binaires. Bien sûr, le mécanisme des assertions d'arc et de leur renforcement peut naturellement produire ce phénomène d'assertions non convexes, mais mieux vaut ne pas l'accroître et même le réduire lorsque l'on divise une place. Nous verrons plus loin au §11.4.1 que les résultats expérimentaux valident indirectement ce choix.

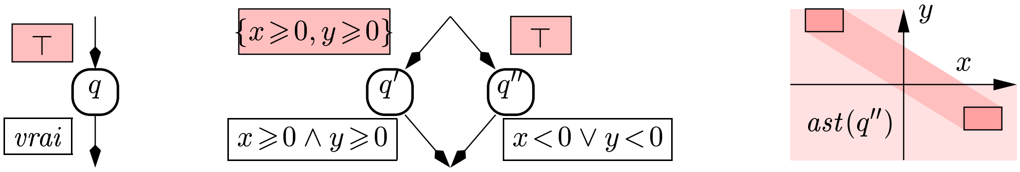


FIG. 9.6 – Division de q sur la condition non binaire $x \geqslant 0 \wedge y \geqslant 0$

Ce point étant acquis, nous allons illustrer différents phénomènes à prendre en compte lorsque l'on cherche à diviser une place. Considérons la situation de la figure 9.7, dans laquelle on cherche à diviser la place q. La définition de q figure dans le rectangle grisé, les assertions d'arcs sont données par les formules encadrées, et l'assertion de la place q, qui est l'union des assertions des arcs sortants, est donnée par la formule encadrée et en traits gras. Par souci de simplicité, on a supposé que le programme n'a pas d'entrées.

Une observation générale est que l'assertion de place est ici plus grosse que la définition. Ceci provient du fait que l"'expressivité" des valeurs abstraites et des formules est incomparables, du fait que ces dernières ne peuvent porter que sur des contraintes apparaissant dans le programme, c.f. §6.1. D'autres observations peuvent être faites :

1. L'assertion de l'arc $q \rightsquigarrow q_1$ ne peut servir à diviser q, puisqu'elle est égale à l'assertion de la place.
2. Supposons que l'on veuille diviser q en considérant l'arc $q \rightsquigarrow q$. L'assertion correspondante n'est pas une condition binaire ; mais puisqu'on sait que de toute manière l'assertion de la place est satisfaite, *on peut évaluer partiellement* $ast(q \rightsquigarrow q)$ *par* $ast(q)$, ce qui donne la condition c, qui convient.
3. Considérons l'assertion $ast(q \rightsquigarrow q_2)$: une troisième observation est que la contrainte $y \geqslant 0$ est redondante dans cette formule, qui contient donc des chemins insatisfaisables. En effet une assertion d'arc, générée et renforcée lors des calculs de post- et pré- condition, contient deux informations de nature différente :
   (a) elle représente une condition nécessaire pour que l'arc puisse être emprunté ;
   (b) elle accumule aussi les connaissances sur la satisfaisabilité des conjonctions de contraintes : lorsqu'un micro-état de la séquence est inaccessible parce qu'il se trouve sous une conjonction de conditions insatisfaisables, le renforcement de l'assertion mémorise ce fait.

   Bien sûr, si l'on veut diviser q en considérant l'arc $q \rightsquigarrow q_2$, après évaluation partielle qui donne la formule $y \geqslant 0 \wedge y \geqslant 5$, il faut choisir la contrainte $y \geqslant 5$ et non $y \geqslant 0$. *Il faut donc travailler sur des formules simplifiées.*

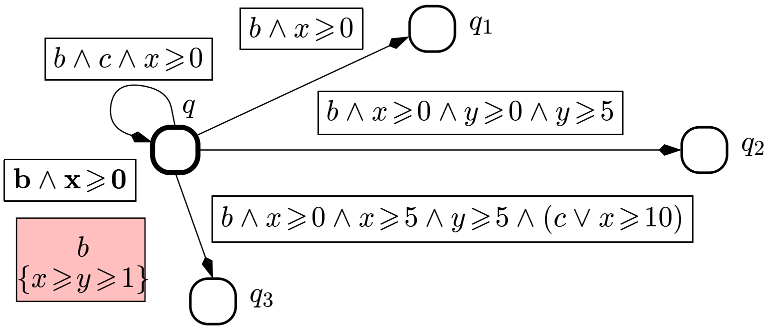


FIG. 9.7 – Exemple de situation pour la division d'une place

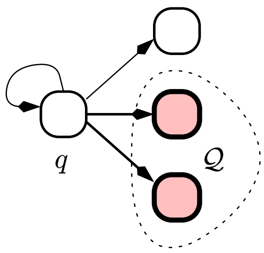
4. La dernière observation illustre le problème d'obtenir une condition binaire. Supposons que l'on veuille diviser q en considérant l'arc $q \rightsquigarrow q_3$. Après évaluation partielle de l'assertion correspondante et simplification du résultat, on obtient la formule $x \geqslant 5 \wedge y \geqslant 5 \wedge (c \vee x \geqslant 10)$, qui n'est pas binaire. Plusieurs solutions sont envisageables :
   - choisir une des deux contraintes en facteur ($x \geqslant 5$ ou $y \geqslant 5$) ;
   - considérer que l'on privilégie la division suivant les variables booléennes : on choisirait alors c, qui appartient au support de la formule ;
   - ou même choisir n'importe quelle condition du support, comme $x \geqslant 10$, qui par ailleurs implique $x \geqslant 5$.

   Si l'on choisit une sous-formule qui n'est pas en facteur de la formule, on n'est pas certain que la division de la place va permettre d'éliminer des arcs du graphe de contrôle.

9.3.2 Extraction d'une condition de division

Nous détaillons dans ce paragraphe l'extraction d'une condition binaire. Considérons une place q, et supposons que nous voulons distinguer parmi les états ceux qui peuvent emprunter directement un ensemble d'arcs $\mathcal{A}$ des autres. Soit $\mathcal{Q}$ l'ensemble des destinations de ces arcs. On va initialement construire la formule

$$f = \bigvee_{q' \in \mathcal{Q}} ast(q \rightsquigarrow q')$$



Notre démarche est ensuite la suivante : nous allons d'abord simplifier f au maximum, en suivant les observations du paragraphe précédent (simplification, évaluation partielle, ...), afin d'obtenir une formule plus simple, dont idéalement *toutes les sous-formules binaires constitueront des conditions de division coupant en 2 la définition de la place.* Ensuite nous allons utiliser des heuristiques pour extraire une condition de division binaire à partir de la formule obtenue.

Construction d'une formule simplifiée

La première chose à faire est d'éliminer les entrées des formules manipulées, c'est-à-dire f et les assertions. Nous l'avons vu au §6.2.3, quantifier sur des booléens est aisé, mais les entrées numériques compliquent les choses. Nous allons utiliser l'algorithme décrit dans ce

Algorithme 9.2 : extraction d'une formule dont les sous-formules binaires sont supposées être des conditions de division adéquates

Input: Une place q et une formule f quelconque

Output: Une formule f' simplifiée, d'état, dont les sous-formules binaires sont des conditions de division adéquates (coupant en 2 la définition de q)

begin
  f' := ElimineEntrées(f)
  if $ast^{state}(q) \Rightarrow f$ **then return** false;
  else begin
    f' := $f' \uparrow^* (ast^{state}(q) \wedge care)$;
    return f'
  end
end

paragraphe-là qui en minimise le coût. Notons que cet algorithme implique une élimination des chemins insatisfaisables. Dans la suite, les assertions manipulées sont des assertions *d'état*, i.e., des assertions dont les entrées ont été éliminées.

Les traitements à effectuer sur f sont donc dans l'ordre les suivants :

1. Il faut quantifier existentiellement les entrées de f, nommons f' la formule obtenue.
2. Il faut comparer f' à l'assertion d'état de la place : si elle lui est égale, f' ne peut servir de critère de division. En toute rigueur il faut comparer f' à la fois à la définition et à l'assertion de place quantifiée, ce qui implique des calculs polyédriques.
3. Comme nous l'avons vu, il convient d'évaluer partiellement la formule obtenue par l'assertion d'état et la définition de la place.

Entre chaque cycle d'analyse, de nombreuses tentatives de raffinement devront être effectuées. C'est pourquoi les opérations précédentes doivent être peu coûteuses ; nous avons donc choisi de n'utiliser que des opérations booléennes. Ainsi l'élimination de chemins insatisfaisables sera effectuée par évaluation partielle par le *don't care set*, les comparaisons de formules seront purement booléennes, la définition de la place ne sera utilisée ni pour des comparaisons avec des formules, ni pour l'évaluation partielle de formules. En raison des approximations qui en découlent, il se peut que certaines sous-formules du résultat obtenu par l'algorithme 9.2 ne découpent pas en fait la définition en deux.

Comme il est très important d'utiliser un opérateur de cofacteur généralisé efficace, nous utilisons celui défini par [Ray91], noté $\uparrow^*$ et décrit au §6.2.1.

Extraction d'une sous-formule binaire

L'obtention d'une condition binaire à partir d'une formule représente un problème délicat, pour lesquelles plusieurs solutions sont possibles, la meilleure ne pouvant être distinguée que par l'expérience.

Si la formule obtenue par l'algorithme précédent est une formule binaire, *en apparence* il n'y a rien à faire : on peut directement diviser la place sur ce critère. Nous allons cependant voir que même dans ce cas, lorsque la formule est booléenne, il peut être préférable d'en extraire une sous-formule plus simple.

Soit f une formule simplifiée. Rappelons qu'on peut d'abord décomposer f en la conjonction d'une formule convexe et d'un "reste" (§6.1.2).

$$f = (f^m \wedge f^b) \wedge f^{nc}$$

avec f^m un monôme sur des contraintes, f^b une formule ne portant que sur des variables booléennes, et f^{nc} un reste. Lorsque f est binaire, soit elle est purement booléenne et $f^m = f^{nc} = vrai$, soit f^m est un littéral et $f^b = f^{nc} = vrai$.

Nous allons prendre comme exemple illustratif la formule décomposée suivante :

$$f = \Big\{ \Big(x \geqslant 0 \wedge y \geqslant 0 \Big) \wedge \Big(b \wedge c \wedge (d \vee e) \Big) \Big\} \wedge (d \Rightarrow x \geqslant 5 \vee y \geqslant 5)$$

Une remarque importante est que si l'on choisit une sous-formule en facteur, *on est sûr d'éliminer un arc*, puisque la sous-formule est impliquée par f, qui elle-même une condition nécessaire pour que l'un des arcs de l'ensemble $\mathcal{A}$ puisse être emprunté. *On favorise donc toujours les divisions sur une sous-formule en facteur.*

On peut par ailleurs privilégier les divisions suivant les formules booléennes ou les contraintes. Enfin, si l'on peut ou l'on choisit d'utiliser f^b, on peut envisager

1. de prendre f^b ; dans l'exemple, $b \wedge c \wedge (d \vee e)$;
2. ou de prendre le monôme en facteur de f^b, si celui-ci est différent de *vrai* ; dans l'exemple, $b \wedge c$;
3. ou si ce monôme contient plusieurs littéraux, d'en choisir un au hasard ; dans l'exemple, b ou c ; cette dernière possibilité consiste à n'autoriser que des divisions sur des littéraux.

Il est aussi judicieux de profiter de la division d'une place pour simplifier les séquences sortantes : le fait que la condition de division ou sa négation soit satisfaite permet éventuellement de supprimer des tests. Dans cette optique, diviser sur des monômes, ou mieux, sur des littéraux, favorise la simplification des séquences. L'inconvénient des monômes par rapport aux littéraux est que leur négation est une disjonction, sur laquelle l'évaluation partielle est moins efficace.

L'algorithme implémenté avec ses options est donné par l'algorithme 9.3. On privilégie toujours les sous-formules de la partie en facteur. Par défaut, on privilégie aussi les divisions sur les formules booléennes.

9.4 Division d'une place sur une condition binaire

Nous expliquons ici le processus de division d'une place q sur une condition binaire *cond*, séparant la définition en deux valeurs abstraites disjointes. Ce processus ne pose pas de problème technique et est donné par l'algorithme 9.4. On crée deux nouvelles places q' et q''. Les séquences sortantes de q sont reséquentialisées sur les nouvelles assertions (spécialisation des séquences sur leur nouvelle définition origine). Les séquences entrantes sont dupliquées et raffinées par propagation arrière de la définition des nouvelles places (spécialisation des séquences sur leur nouvelle définition de destination). La duplication de la boucle éventuelle sur q est traitée à part. Finalement, on enlève de la structure de contrôle la place q et tous les arcs concernés.

Bien sûr, dès qu'une assertion d'arc devient fausse, l'arc est implicitement supprimé.

Algorithme 9.3 Extraction d'une condition de division adéquate

Input: une formule f simplifiée

Output: une sous-formule binaire

Options (booléennes)

  atom : on ne divise que sur des littéraux

  num : on privilégie les divisions sur les contraintes dans les monômes

  bool : on privilégie les divisions sur les booléens dans la partie non convexe

begin

  let $(f^m \wedge f^b) \wedge f^{nc} = f$ **in**

  if $f^m = f^b = vrai$ **then begin** { donc $f = f^{nc}$ }

  { On choisit une condition du support }

    return if bool **then** choisir_bool(f^{nc}) **else** choisir_num(f^{nc})

  else begin

    { $f^m \wedge f^b \neq vrai$ }

    if $f^m = vrai$ or not num and $f^b \neq vrai$ **then begin**

      { - On ne peut pas diviser sur une contrainte, }

      { - Ou on privilégie les booléens et on peut diviser sur les booléens }

      if $mon(f^b) \neq vrai$ **then**

        return if atom **then** choisir_bool($mon(f^b)$) **else** $mon(f^b)$

      else

        return if atom **then** choisir(f^b) **else** f^b

    end else

      { On divise sur une contrainte }

      return choisir_num(f^m)

  end

end

Algorithme 9.4 DivisePlace

Input: une structure de contrôle G, une place q et une condition $cond$.

Output: une structure de contrôle G modifiée par la division de q en q' et q''.

```
begin
  let G = (S, A) in
  S := S ∪ {q', q''};
  ast(q') := ast(q)∧cond; def(q') := def(q)⊓cond;
  ast(q'') := ast(q)∧¬cond; def(q'') := def(q)⊓¬cond;
  { Duplication des arcs sortants de q }
  for q ⇝ s ∈ A s.t. s ≠ q do begin
    A := A ∪ {q' ⇝ s, q'' ⇝ s};
    ast(q' ⇝ s) := ast(q ⇝ s)∧cond;
    seq(q' ⇝ s) := SequentialiseMicroEtat(ops,ast(q' ⇝ s),def(q'),def(s));
    ast(q'' ⇝ s) := ast(q ⇝ s)∧¬cond;
    seq(q'' ⇝ s) := SequentialiseMicroEtat(ops,ast(q'' ⇝ s),def(q''),def(s));
  end
  { Duplication des arcs entrants de q }
  for p ⇝ q ∈ A s.t. p ≠ q do begin
    A := A ∪ {p ⇝ q', p ⇝ q''};
    ast(p ⇝ q') := ast(p ⇝ q); seq(p ⇝ q') := seq(p ⇝ q);
    RaffineMicroEtat(seq(p ⇝ q'),ast(p ⇝ q'),def(p),def(q'));
    ast(p ⇝ q'') := ast(p ⇝ q); seq(p ⇝ q'') := seq(p ⇝ q);
    RaffineMicroEtat(seq(p ⇝ q''),ast(p ⇝ q''),def(p),def(q''));
  end
  { Duplication de la boucle, si elle existe }
  if q ⇝ q' ∈ A then begin
    A := A ∪ {q' ⇝ q', q'' ⇝ q'', q' ⇝ q'', q'' ⇝ q'};
    ast(q' ⇝ q') := ast(q' ⇝ q'') := ast(q ⇝ q)∧cond;
    ast(q'' ⇝ q'') := ast(q'' ⇝ q') := ast(q ⇝ q)∧¬cond;
    seq(q' ⇝ q') := SequentialiseMicroEtat(ops,ast(q' ⇝ q'),def(q'),def(q'));
    ...
  end
  G := G \ {q};
end
where RaffineMicroEtat(seq,ast,initial,final) ≡ begin
  propage_arriere(seq)(final)
  insat := Insat(seq)
  if insat≠faux then begin
    ast := ast and not insat;
    seq := SequentialiseMicroEtat(ops,ast,initial,final)
  end
end
```

9.5 Raffinement de séquence

Le raffinement de séquence que nous envisageons est celui fondé sur la détection des mauvaises unions convexes, et plus précisément sur la détection de la perte d'égalités ou du gain de droites, introduit au §9.1.1. Les bornes supérieures prenant place dans les séquences sont toutes binaires, ce qui simplifie grandement l'analyse. Comme d'habitude, nous n'essaierons de raffiner qu'après convergence des analyses.

Il y a deux cas à examiner : celui du calcul avant et celui du calcul arrière. Nous utiliserons la séquence de la figure 9.8 pour illustrer la discussion, et plus particulièrement la sous-séquence définie par le test sur la condition d'état c. X_1 et X_2 sont les micro-états initiaux des deux branches, et Y_1 et Y_2 sont leur micro-états finaux. X_1 et X_2 satisfont respectivement les conditions c et $\neg c$. On suppose que la séquence est associée à un arc $p \rightsquigarrow s$.

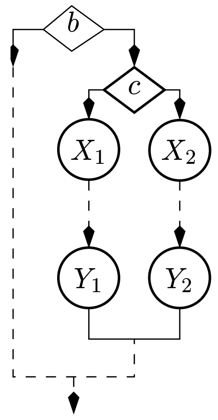


FIG. 9.8 –

9.5.1 Calcul avant

Supposons, après convergence que la borne supérieure des Y_i soit mauvaise au sens du §9.1.1. Pour éviter cette borne supérieure, il faut diviser la place successeur s en deux places s' et s'', avec des définitions identiques et $ast(p \rightsquigarrow s') = ast(p \rightsquigarrow s) \wedge c$ et $ast(p \rightsquigarrow s'') = ast(p \rightsquigarrow s) \wedge \neg c$. Bien sûr, les définitions initialement identiques de s' et s'' vont se différencier après les analyses dynamiques qui vont suivre.

Le raffinement de séquence passe donc par un raffinement de la place s. Le problème est que cette division de s ne se fait par sur des conditions du programme, or nous avons décidé au §9.1.4 de nous restreindre à de tels raffinements.

Nous ne pouvons donc pas raffiner explicitement les séquences sur ce critère .

9.5.2 Calcul arrière

En revanche, la suppression d'une mauvaise borne supérieure en arrière nous fournit un critère acceptable. Supposons que la borne supérieure des X_i soit mauvaise. On peut la supprimer en divisant la place origine p en p' et p'' avec $def(p') = def(p) \sqcap c$ et $def(p'') = def(p) \sqcap \neg c$. c étant une variable booléenne ou une contrainte, il n'y a pas d'extraction de sous-formule binaire à effectuer.

Pratiquement, notre outil procède de la manière suivante : pour chaque place on examine les conditions associées aux mauvaises unions convexes en arrière dans toutes les séquences sortantes. À chacune de ces conditions est associée un poids, qui est la somme de la perte d'égalité et du gain de droite. La place est ensuite divisée suivant une condition de poids maximal. Comme d'habitude, on ne divise qu'une fois une place entre deux analyses.

9.6 Assertions non convexes

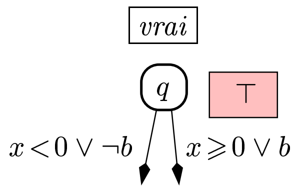
Lorsque notre heuristique principale échoue, c'est-à-dire que la propriété n'est pas prouvée et qu'aucun critère de division n'est trouvé, on peut essayer de rendre les assertions plus "convexes", par division de place.

9.6.1 Choix d'une condition

Soit f une formule non convexe. On peut réécrire f en $f^b \wedge f^m \wedge f^{nc}$, cf. §6.1.2, avec $f^{nc} \neq vrai$ portant sur des contraintes et éventuellement des booléens, et $mon(f^{nc}) = vrai$. Il suffit de prendre comme condition la racine c de f^{nc}, qui est nécessairement une contrainte, pour séparer f en union de deux formules "plus convexes" $f \wedge c$ et $f \wedge \neg c$. Notons que la conversion d'une formule en union exacte d'états convexes procède ainsi de manière récursive, cf. §6.1.3.

9.6.2 Application aux assertions de place et aux assertions d'arc

On peut appliquer le paragraphe précédent aux assertions de place non convexes, en les décomposant en deux assertions qui vont permettre de diviser leur place.



Il peut être intéressant de l'appliquer aussi aux assertions d'arc : imaginons la situation de la figure ci-contre. Les assertions des deux arcs sortants étant complémentaires, l'assertion de place (formule encadrée) ne contient aucune information. Dans un tel cas, une condition de division extraite d'une assertion d'arc permettra de mieux différencier les états considérés à la place q.

Concernant le choix entre assertions de place et assertions d'arc, on essaie d'abord d'extraire une condition de division de l'assertion de place, et en cas d'échec on parcourt les arcs sortants jusqu'à trouver une condition de division adéquate.

9.7 Combinaison des heuristiques

Nous avons défini dans ce chapitre trois heuristiques :

1. celle consistant à diviser une place sur une condition discriminante, en avant ou en arrière ;
2. l'heuristique évitant les mauvaises unions dans les séquences lors d'un calcul arrière ;
3. enfin, l'heuristique cherchant à rendre les assertions de place et d'arc convexes.

Lorsqu'on combine ces heuristiques, le nombre de possibilités est à peu près infini. Nous avons distingué celles qui suivent, et qui appliquent les choix généraux du §9.2.2, car elles se révèlent les plus efficaces. Entre deux cycles d'analyses :

- Par défaut, on cherche d'abord à appliquer l'heuristique 2 pour diviser des places. Dans la pratique, il est rare que cette heuristique nous donne une condition de division. On considère ensuite les places non encore divisées. Pour chacune de ces places, l'algorithme 9.1 va nous donner différentes conditions de division, en avant et/ou en arrière. On privilégie celles qui coupe les sous-composantes fortement connexes de faible profondeur, et à profondeur égale on favorise les conditions en avant aux conditions en arrière (souvent moins précises). Enfin, si *aucune* place n'a pu être divisée, on utilise l'heuristique 3 sur chaque place.
- Une première option consiste à appliquer l'heuristique 3 même si les autres heuristiques ont permis la division de places. On l'applique alors aux places non encore divisées. Ceci n'est qu'une option, car cette heuristique est la plus hasardeuse : elle peut engendrer des divisions totalement inutiles.

– Une seconde option, indépendante de la précédente, est d'utiliser conjointement les conditions de division en avant et en arrière pour extraire une condition de division binaire, lors du parcours des places par l'algorithme 9.1.

9.8 Conclusion

Le principe du raffinement que nous avons mis en œuvre consiste à séparer, dans chaque place, les états suivant leur comportement futur ou leur comportement passé. Ceci se fait en analysant respectivement les préconditions des arcs sortants de la place ou les postconditions des arcs entrants. Ce principe a été appliqué dans un cadre différent par [SUM96]. Nous utilisons également des heuristiques secondaires, mais elles sont plus rarement nécessaires et souvent beaucoup moins efficaces.

L'originalité de notre outil est de l'appliquer un cadre différent, et sur des programmes important, ce qui nécessite une automatisation complète. En particulier nous proposons

– un algorithme pour choisir automatiquement les places à diviser et les conditions de division à utiliser, fondé sur la notion de sous-composantes fortement connexes (§9.2), qui se révèle expérimentalement efficace dans des conditions "difficiles" (lorsque pour une place donnée, le nombre de conditions possibles est élevé) ;
– une méthode pour extraire une condition de division adéquate à partir d'une condition discriminante plus générale (§9.3) ; cette méthode est généralisable à d'autres contextes : en effet lorsque la borne supérieure perd de l'information dans un treillis abstrait, généralement la complémentation d'une valeur abstraite est difficile.

Une autre caractéristique importante de notre raffinement est qu'il induit automatiquement un raffinement des opérateurs de pré- et post-condition, par spécialisation de ces opérateurs sur leur domaine et leur codomaine (§7.6), bien qu'il soit fondé sur la division des places,

La partie III qui suit montre que ces heuristiques sont efficaces : elles sont d'une part assez puissantes pour effectuer suffisamment de divisions et réussir à prouver la plupart des propriétés, et d'autre part elles ne génèrent pas trop de divisions sur des programmes importants.

Troisième partie

Implémentation et résultats expérimentaux

Chapitre 10

Implémentation

Ce chapitre présente l'outil implémenté durant la thèse et qui met en œuvre les idées de la partie II.

NOUS décrivons ici l'implémentation de notre outil de vérification, appelé NBAC (Numerical and Boolean Automaton Checker) par référence à l'outil BAC [Hal94]. Nous avons également du implémenter une nouvelle bibliothèque de manipulation de polyèdres convexes, ainsi qu'un traducteur du langage LUSTRE vers le langage d'entrée de notre outil, que nous appelerons aussi NBAC. Tous nos outils ont été programmés avec l'outil NOWEB de programmation littérale [Ram94] et sont donc relativement documentés.

Le §10.1 discute du choix de la bibliothèque de manipulation de BDDs et des fonctionnalités apportées par la bibliothèque de polyèdres que nous avons écrite. Le §10.2 présente l'outil NBAC, son format d'entrée et les options implémentées. Enfin le §10.3 présente le traducteur LUS2NBAC.

10.1 Procédures de décision

Notre outil de vérification manipule des formules et des états convexes, qui sont la conjonction d'une formule portant sur des booléens et d'un polyèdre. Les manipulations de formules et de polyèdres sont effectuées respectivement par la bibliothèque de BDDs CUDD [Som] et la bibliothèque de polyèdres que nous avons développée spécifiquement et qui présente des perfectionnements comparée à celle développée à l'IRISA [Wil93] et celle qui était utilisée au laboratoire VERIMAG.

10.1.1 La bibliothèque de BDDs CUDD

Cette bibliothèque a été développée à l'université du Colorado en langage C. Elle permet en autre la manipulation de BDDs, (en fait des TDGs), ainsi que la manipulation d'ADDs (Algebraic Decision Diagrams [BFG$^+$93]) qui sont des diagrammes de décision binaires dont les feuilles sont des nombres flottants. Nous avons choisi d'utiliser cette bibliothèque car elle est reconnue comme très efficace, elle offre à peu près toutes les fonctionnalités que l'on peut imaginer, elle propose de nombreuses heuristiques de réordonnancement de variables, et les ADDs permettent de coder les MTBDDs dont nous avons besoin pour représenter les fonctions de transition numériques.

Pour la représentation des formules booléennes, nous avons utilisé directement les BDDs de la bibliothèque, et pour les MTBDDs nous avons réalisé une surcouche gérant la correspondance entre nombres flottants et expressions affines. La correspondance entre pseudo-variables booléennes et inégalités linéaires est en revanche gérée par l'outil NBAC.

Nous avons implémenté les opérateurs de cofacteurs améliorés, qui sont décrits au §6.2.1, à la fois pour les BDDs et les ADDs, qui bénéficient de la mécanique des caches offerte par la bibliothèque.

Comme en fait notre outil est programmé dans le langage CAML, nous avons également réalisé une interface permettant d'appeler les fonctions dont nous faisons usage depuis un programme CAML.

10.1.2 L'implémentation de la bibliothèque de polyèdres

Les premières versions de l'outil NBAC utilisait la bibliothèque de polyèdres développée par N. Halbwachs pour l'outil POLKA. Nous en avons développé une nouvelle au cours de la seconde année de thèse pour remédier à une limitation rédhibitoire, qui était que les dépassements de capacité se produisait fréquemment sur les entiers 32 bits utilisés, et en avons profité pour apporter des améliorations. Pour cela nous nous sommes inspirés de la bibliothèque de l'IRISA [Wil93] et de l'article [FP96].

La nouvelle bibliothèque, qui n'a pas encore trouvé de nom, a été programmée en C et manipule des polyèdres dans $\mathbb{Q}^n$ (les coefficients des contraintes et des générateurs sont des rationnels). Bien qu'elle n'apporte aucun résultat de recherche, elle représente une implémentation soignée. Outre la bibliothèque de l'IRISA, d'autres sont disponibles (cf. [Fuk]), mais en général elles ne contiennent que les algorithmes centraux de conversion et de minimisation. Il leur manque la gestion mémoire et les opérations dont nous avons besoin : enveloppe convexe, intersection, transformation linéaire, ...

La taille cumulée des fichiers C extraits des sources NOWEB est d'environ 3000 lignes.

Quelques rappels sur les polyèdres

Comme nous l'avons mentionné au §2.4.4, un polyèdre peut alternativement être représenté par une conjonction d'inégalités linéaires :

$$P = \{x \mid \bigwedge_{1\leqslant j\leqslant m} a_j x \geqslant b_j\} = \{x \mid Ax \geqslant B\}$$

ou par ses éléments générateurs :

$$P = \left\{ \sum_{s_i \in S} \lambda_i s_i + \sum_{r_j \in R} \mu_j r_j \mid \lambda_i \geqslant 0, \mu_j \geqslant 0, \sum_i \lambda_i = 1 \right\}$$

où S est un ensemble de *sommets* et R un ensemble de *rayons*. (A, B) et (S, R) sont respectivement un système de contraintes et un système générateur de P.

Comme travailler dans un cadre linéaire est plus simple que travailler dans un cadre affine, on plonge de manière classique $\mathbb{Q}^n$ dans $\mathbb{Q}^{n+1}$ avec la transformation $x \mapsto \begin{pmatrix} \xi x \\ \xi \end{pmatrix}, \xi \geqslant 0$, comme l'explique [Wil93]. Les polyèdres que la bibliothèque manipule sont donc des cônes. La transformation inverse consiste à intersecter un cône avec le plan $\xi = 1$. Ceci nous permet de nous concentrer sur les cônes, dont les représentation duales sont

$$P = \{x \in \mathbb{Q}^{d+1} \mid Ax \geqslant 0\} = \{x \in \mathbb{Q}^{d+1} \mid x = R\mu, \mu \geqslant 0\}$$

L'algorithme de Chernikova [Che68, LeV92], également appelée méthode de la double description, est un algorithme permettant de passer d'une représentation à l'autre et de minimiser la taille de la représentation. La conversion permet d'obtenir très facilement l'intersection de polyèdres, en prenant l'union des contraintes des polyèdres concernés, ou l'enveloppe convexe, en prenant l'union des générateurs des polyèdres concernés. Cet algorithme est incrémental et son principe est le suivant : supposons que l'on veuille calculer les générateurs d'un polyèdre (plus précisément, d'un cône) connu par ses contraintes. L'algorithme démarre avec un système de générateurs représentant le cône univers, puis à chaque étape va intersecter le cône courant avec une nouvelle contrainte et modifier conformément le système de générateurs, de façon à ce qu'à tout instant le polyèdre décrit par le système de générateurs corresponde à celui défini par les contraintes prises en compte. Le système de générateurs obtenu est sous forme minimale. En général, on procède ensuite à la simplification du système de contraintes, qui est relativement facile une fois que l'on connaît un système minimal de générateurs. Les deux représentations étant duales, la conversion d'un système de générateurs vers un système de contraintes se fait exactement par le même algorithme .

Améliorations apportées par la bibliothèque

Nous décrivons ici les caractéristiques de notre bibliothèque qui n'étaient pas disponibles dans celle de l'IRISA ni dans celle de l'outil Polka. Certaines de ces caractéristiques sont disponibles dans d'autres bibliothèques.

Représentation des coefficients. — La motivation initiale pour réécrire une nouvelle bibliothèque était les problèmes de dépassement de capacité des coefficients rencontrés lorsque ceux-ci sont représentés par des entiers machines, en général sur 32 bits. Il faut savoir que tous les coefficients d'un vecteur utilisent le même dominateur commun. [Avi98] a observé expérimentalement que cette représentation était plus efficace que celle consistant à prendre un nombre rationnel pour chaque coefficient, sans doute parce que l'addition est plus aisée. Dans ces conditions, lorsque la dimension des polyèdres est importante (même de l'ordre de la dizaine), les coefficients explosent, en particulier lors de l'algorithme central de conversion entre contraintes et générateurs qui combine les vecteurs entre eux. La bibliothèque que nous avons écrite permet donc de choisir la représentation des entiers. Actuellement elle peut utiliser soit des entiers machines `long int` (usuellement 32 bits), soit des entiers `long long int` (usuellement 64 bits) proposés par le compilateur Gcc [GNU], ou enfin les entiers multi-précisions de la bibliothèque GMP (Gnu Multi-Precision Library [GNU]). L'interface de manipulation des entiers est générique et il est aisé pour un utilisateur d'interfacer la bibliothèque avec une autre bibliothèque d'entiers multi-précision (cela a été fait avec la bibliothèque Leda).

Calculs paresseux. — La conversion d'une représentation à l'autre et la minimisation peuvent être retardées. La bibliothèque de l'IRISA garde toujours les deux représentations en mémoire, alors que nous pouvons n'en garder qu'une et n'effectuer une conversion que lorsque cela est nécessaire, par exemple lorsque nous voulons intersecter des polyèdres dont les contraintes ne sont pas disponibles. Quand les deux représentations sont disponibles, elles sont nécessairement minimales, mais ce n'est pas le cas lorsque seulement une est disponible.

Matrice de saturation en mémoire. — La *matrice de saturation* d'un ensemble de contraintes et de générateurs est une matrice qui indique quelles contraintes *saturent* quels générateurs. Une contrainte sature un générateur lorsque leur produit scalaire est nul, ce qui signifie que le générateur appartient à l'hyperplan défini par la contrainte. La matrice de saturation est utilisée par l'algorithme de conversion et de minimisation. Lorsque nous disposons des deux représentations d'un polyèdre, nous avons choisi de l'incorporer dans la représentation, ce qui évite d'avoir à la recalculer lorsque par exemple l'on ajoute une contrainte ou un générateur à un polyèdre. Par conséquent les opérations d'intersection ou d'enveloppe convexe sont accélérées et sont encore plus incrémentales. Ceci est particulièrement utile lorsque nous effectuons un parcours descendant des BDDs et MTBDDs pour éliminer les chemins insatisfaisables (§5.2), puisque lors d'un tel parcours les contraintes sont ajoutées une par une.

Tri des contraintes et générateurs. — Les lignes des matrices sont triées lexicographiquement (et paresseusement, comme la conversion). Ceci présente deux avantages : [FP96] observe que ceci accélère l'algorithme de conversion ; dans notre cas, lorsque par exemple nous fusionnons les contraintes de deux polyèdres lorsque nous calculons leur intersection, ceci permet d'éliminer facilement les contraintes identiques.

Conclusion

Concernant l'efficacité réelle de la bibliothèque, nous n'avons malheureusement pas eu le temps de comparer l'efficacité de cette bibliothèque avec l'ancienne (celle de l'outil POLKA, très proche de celle de l'IRISA), car ceci nécessite une interface commune.

Étant donné le nombre de courriers électroniques reçus en un an par l'auteur de ses lignes ou plus fréquemment par son directeur de thèse à propos de l'outil POLKA, il est clair que l'utilisation des polyèdres en analyse statique et en vérification de programme se répand, et nous espérons que cette bibliothèque sera utile à d'autres équipes.

10.2 L'outil NBAC

10.2.1 Choix du langage de programmation

Nous avons choisi le langage CAML [CAM] pour notre outil. Ce langage fonctionnel d'orde supérieur à typage statique, qui est une variante de ML, permet une programmation très concise et offre la sécurité et les facilités d'utilisation classiques de cette famille de langage : gestion de la mémoire transparente (ramasse-miettes), système de modules puissant, facilité de définir et de manipuler des structures de données complexes, pretty-printing, ... Comme nous avons expérimenté de nombreux algorithmes, ces avantages ont été déterminants. Concernant l'efficacité du code généré, le compilateur OCAML compile vers du code natif pour de nombreuses architectures, et les procédures de décision coûteuses que nous utilisons (BDDs et polyèdres) sont codées en C.

L'interface avec ces bibliothèques est propre et gère la mémoire : le ramasse-miette de OCAML est en effet capable d'appeler des fonctions de finalisation sur les objets de type externe.

La taille cumulée des fichiers CAML constituant l'outil NBAC proprement dit est de 10000 lignes, auxquelles il faut ajouter le code des interfaces avec les bibliothèques de BDDs et de polyèdres (750 lignes de CAML et 2000 lignes de C).

10.2.2 Langage d'entrée de l'outil

La grammaire de notre langage est la suivante :

⟨automaton⟩ ::= ⟨declarations⟩ ⟨definitions⟩ ⟨transitions⟩
  ⟨initial⟩ [⟨assertion⟩] ⟨invariant⟩

⟨declarations⟩ ::= ⟨state_declarations⟩ [⟨input_declarations⟩]
  [⟨local_declarations⟩]
⟨state_declarations⟩ ::= **state** ⟨vars_type_list⟩
⟨input_declarations⟩ ::= **input** ⟨vars_type_list⟩
⟨local_declarations⟩ ::= **local** ⟨vars_type_list⟩
⟨vars_type_list⟩ ::= ⟨vars_type⟩ | ⟨vars_type⟩ ⟨vars_type_list⟩
⟨vars_type⟩ ::= ⟨ident_list⟩ ":" ⟨type⟩ ";"
⟨ident_list⟩ ::= ⟨ident⟩ | ⟨ident⟩ ⟨ident_list⟩
⟨type⟩ ::= **bool** | **int**

⟨definitions⟩ ::= **definition** ⟨definition_list⟩
⟨definition_list⟩ ::= ⟨definition⟩ | ⟨definition⟩ ⟨definition_list⟩
⟨definition⟩ ::= ⟨ident⟩ "=" ⟨expression⟩ ";"
⟨transitions⟩ ::= **transition** ⟨transition_list⟩
⟨transition_list⟩ ::= ⟨transition⟩ | ⟨transition⟩ ⟨transition_list⟩
⟨transition⟩ ::= ⟨ident⟩"'" "=" ⟨expression⟩ ";"

⟨expression⟩ ::= "`true`" | "`false`" | ⟨num⟩ | ⟨ident⟩ |
  "(" ⟨expression⟩ ")" | **not** ⟨expression⟩ |
  ⟨expression⟩ ⟨binop⟩ ⟨expression⟩ |
  "#"(⟨expression_list⟩ ")" |
  if ⟨expression⟩ **then** ⟨expression⟩ **else** ⟨expression⟩
⟨binop⟩ ::= **or** | **and** | **xor** | **eq** | **xor** | "=>" |
  "=" | "<" | "<=" | ">=" | ">" |
  "+" | "-" | "*"

⟨initial⟩ ::= **initial** ⟨expression⟩ ";"
⟨assertion⟩ ::= **assertion** ⟨expression⟩ ";"
⟨invariant⟩ ::= **invariant** ⟨expression⟩ ";"

"*" a une priorité plus forte que "+" et "-", qui sont prioritaires sur "<", "<=", "=", ">=" et ">", qui ont priorité plus forte que **not** , qui a une priorité plus forte que **eq** et **xor** , prioritaires sur **and** , prioritaire sur **or** , prioritaire sur "=>". Ces opérateurs ont la signification habituelle. "#" est l'opérateur ou exclusif : il retourne vrai si et seulement si au plus une de ses opérandes est vraie.

Les définitions donnent la valeur des variables locales en fonction de variables non primées. Sémantiquemement ce sont des macros. Les dépendances cycliques dans les définitions sont détectées et rejetées. Les transitions donnent la valeur suivante d'une variable d'état en fonction des variables non primées.

Il existe trois classes de variable : les variables d'état, les variables d'entrées, et les variables locales. Un automate possède au minimum une variable d'état. Chaque classe est encore divisée suivant le type des variables : celles-ci peuvent être de type booléen ou numérique. Le domaine numérique est pour l'instant restreint aux entiers relatif, mais devrait être étendu aux rationnels.

Hormis les variables locales qui sont des macros, la sémantique du langage a déjà été donnée au §3.3.3.

10.2.3 Options implémentées

Nous décrivons le mode de l'emploi de notre outil et les options que nous avons implémentées.

`-bdd (1|2|m|p|s)` : options concernant les BDDs. Rappelons que les contraintes sont *toujours* au dessus des variables booléennes dans les BDDs et MTBDDs.

- `s` : effectue une analyse d'accessibilité booléenne du programme telle que décrite au §8.1.2.
- `1` : effectue un réordonnancement de variables dans les BDDs après substitution des variables locales. Les BDDs qui sont alors en mémoire sont les fonctions de transition et les formules *init*, *inv*, *bad* initiales, ainsi que le "*care set*" sur les contraintes.
- `2` : en conjonction avec `s`, effectue un second réordonnancement de variables après l'analyse d'accessibilité booléenne.
- `m` : autorise le mélange des variables booléennes d'entrée et d'état. Par défaut les entrées sont placées sous les variables d'état.
- `p` : partitionne les contraintes en ensembles de contraintes parallèles ; lors d'un réordonnancement, ce sont les partitions qui sont réordonnées. Ceci minimise la taille du "*care set*".

`-simpl (s<n>|r<n>)` : simplification des diagrammes interprétés et cofacteurs généralisés (§6.2.1 et §6.2.2). `n` signifie :

- `0` : utilisation de l'opérateur "restrict"
- `1` : utilisation de l'opérateur "restrict" amélioré (effectuant un parcours descendant)
- `2` : utilisation d'opérations polyédriques (élimination de *tous* les chemins insatisfaisables)

`s` spécifie l'opérateur à utiliser dans l'algorithme de séquentialisation, et `r` celui utilisé dans les heuristiques de raffinement.

`-widening (d|p|f<n>|r<n>)` : type d´élargissement utilisé (§8.4) :

- `d` : détermination dynamique des points d'élargissement.
- `p` : élargissement paramétré par des contraintes "importantes"
- `f<n>` : élargissement partiel :
  - `0` : pas d´élargissement partiel ;
  - `1` : élargissement partiel lors de l'activation d'un nouvel arc entrant ;
  - `2` : élargissement partiel lors de l'activation d'une nouvelle branche d'une séquence entrante.
- `r<n>` : effectue jusqu'à `n` itérations décroissantes après convergence.

`-widening_freq <n>` : fréquence d'application de l'élargissement ; `-widening_freq 2` signifie par exemple utilisation toutes les deux itérations.

`-widening_limit <n>` : nombre de contraintes au delà duquel l'élargissement partiel n'est plus appliqué ; cette option est utile lorsque l'élargissement partiel ne simplifie pas assez les polyèdres et que le nombre de contraintes augmente dangereusement.

`-div (numeric|bool|atom|combine)` : extraction d'une condition de division adéquate (§9.3 et §9.7) :

- `n` : favorise les conditions numériques sur les booléennes dans les monômes en facteur (par défaut, l'inverse)
- `b` : favorise les conditions booléennes sur les numériques dans les parties non convexes (par défaut, l'inverse) ;
- `a` : divise les places seulement sur des atomes ;
- `c` : combine les conditions avant et arrière ;
- `s` : cherche toujours à utiliser une heuristique secondaire pour diviser plus de places.

`-seq (bool|num|exact|vec)` : options de séquentialisation :

- `b` : pour les fonctions de transition booléennes, considère non seulement les conditions numériques, mais aussi les conditions booléennes apparaissant également dans les fonctions numériques.
- `n` : donne la priorité absolue aux conditions numériques dans le choix des conditions ;
- `e` : ne referme aucun test, i.e., génère des séquences n'effectuant pas d'approximation[1].
- `v` : lorsqu'on ouvre un test sur une condition numérique $c(\vec{x}) \geqslant b$, sélectionne sous le test *toutes* les opérations dépendant d'une condition de type $c(\vec{x}) \geqslant b'$, plutôt que seulement celle dépendant de $c(\vec{x}) \geqslant b$.

Les autres options sont des options d'affichage que nous ne détaillons donc pas ici.

10.2.4 Les options efficaces

Nous indiquons quelles sont les combinaisons d'options les plus efficaces. Concernant les options `-bdd`, on a toujours intérêt à utiliser `-bdd 2ms`, i.e., on effectue une analyse d'accessibilité booléenne, on réordonne les variables en utilisant l'algorithme du "*sifting*" [Rud93], et on autorise le mélange des deux types de variables booléennes. Pour les options de simplification et d'évaluation partielle, nous n'utilisons jamais d'opérations polyédriques, et les deux combinaisons efficaces sont `-simpl s0r1` et `-simpl s1r1`. L'inconvénient de l'opérateur "restrict" procédant de haut en bas est son coût en temps, mais il est toujours meilleur que le "restrict" standard. Les options `-widening` sont à peu près toutes utilisables, elles ont en fait une faible influence sur la vérification globale. Il faut juste limiter le nombre d'étapes décroissantes, qui rendent les polyèdres compliqués. Nous utilisons le plus souvent `-widening dpr1f1`. Concernant la fréquence d'application de l'élargissement, nous élargissons à chaque itération. Concernant le raffinement, les divisions sur des atomes sont presque toujours plus efficaces que les divisions sur des formules plus générales, et nous utilisons en général `-div a` ou `-div ac`. Nous ne positionnons usuellement aucune option de séquentialisation.

[1] dans ce cas, on pourrait améliorer les calculs de pré- et post-condition, ce que nous n'avons pas fait

10.3 L'outil LUS2NBAC

Nous avons donné la syntaxe et la sémantique de notre format d'entrée dans le paragraphe précédent. Ce format est cependant de bas niveau, aussi c'est seulement pour de petits exemples que nous l'avons utilisé directement. Pour les exemples plus importants, nous avons utilisé le langage LUSTRE. Nous avons donc programmé un traducteur LUS2NBAC qui génère un programme NBAC à partir d'un programme LUSTRE et de l'observateur d'une propriété.

Le traducteur réutilise la partie frontale du compilateur LUSTRE V4, qui fait des vérifications syntaxiques et sémantiques, détecte les variables d'état syntaxiquement équivalentes, et effectue une analyse de cône d'influence (ou de dépendance).

Le programme NBAC généré ne comporte donc que des variables influençant la propriété ou l'assertion du programme, et les analyses effectuées par la partie frontale du compilateur LUSTRE V4 ont permis de diminuer le nombre de variables à prendre en compte.

Nous avons également programmé un traducteur DC2NBAC du format DC vers le format NBAC. Le format DC, qui signifie Declarative Code, est le format intermédiaire commun aux langages LUSTRE, ESTEREL et SIGNAL. Nous avons cependant laissé ce traducteur dans un état instable, car le compilateur LUSTRE V5 qui compile dans ce format n'est pas encore complet et ni le compilateur ni l'outil DRAC manipulant le code DC n'implémentent à l'heure actuelle les analyses effectuées par le compilateur LUSTRE V4 et qui se révèlent bien utiles.

Chapitre 11

Études de cas

Nous illustrons dans ce chapitre l'utilisation de notre outil pour vérifier diverses propriétés de sûreté sur un certain nombre de programmes, et analysons également son comportement interne sur quelques-uns de ces exemples.

Un certain nombre de programmes d'école vont d'abord être présenté au §11.1, illustrant le comportement de notre outil dans différentes situations. Nous présentons ensuite au §11.2 un contrôleur d'ascenseur assez réaliste, sur lequel nous vérifions plusieurs propriétés. Le §11.3 présente une étude de cas du projet de recherche européen VHS (Verification of Hybrid Systems) que nous avons traitée. Enfin le §11.4 analyse plus en détail le comportement interne de l'outil et l'effet de différentes options sur la vérification de propriété.

Les résultats que nous présentons ici correspondent à ceux obtenus en positionnant au mieux les options de notre outil, lesquelles seront précisées pour chaque exemple. Les temps d'exécution et la consomnation mémoire sont mesurés avec la commande `time`. La machine sur laquelle nous avons effectué les expériences est un PC pentium III biprocesseur tournant à une fréquence de 600 Mhz et possédant 2 Go de mémoire vive. L'outil n'utilise bien sûr qu'un seul des processeurs. À titre de comparaison, une telle machine donne sur notre outil des performances trois fois plus élevées qu'une station SUN UltraSparc 10 plus répandue. Nous avons toujours utilisé la version de l'outil qui utilise des entiers multi-précision pour les polyèdres, même si pour certains exemples ce n'est pas nécessaire. Mentionnons que la version utilisant des entiers 64 bits permet de gain d'un facteur 2 en temps d'exécution, et un peu moins en mémoire.

Ces résultats sont présentés sous la forme d'un tableau du type suivant :

	Complexité				Vérification					
	bool	num	cont	auto	bdd	ana	div	taille max	temps	mem
	s/i	s/i				b/n	p/a			

Dans les sous-colonnes de la colonne **Complexité**, **bool** et **num** indiquent respectivement le nombre de variables booléennes et numériques, en distinguant variables d'état et d'entrées. **cont** donne le nombre de contraintes du programme[1]. **auto** donne la taille de l'automate booléen réduit par bisimulation par l'outil Ocmin, lorsqu'il a pu être construit.[2]

[1] ce nombre peut être surestimé, car certaines contraintes introduites lors de l'analyse du source Nbac n'apparaissent plus après substitution des variables locales.

[2] rappelons que cet automate ne permet pas nécessairement la preuve d'une propriété, puisque quelquefois il est nécessaire de séparer un état booléen d'après une contrainte numérique.

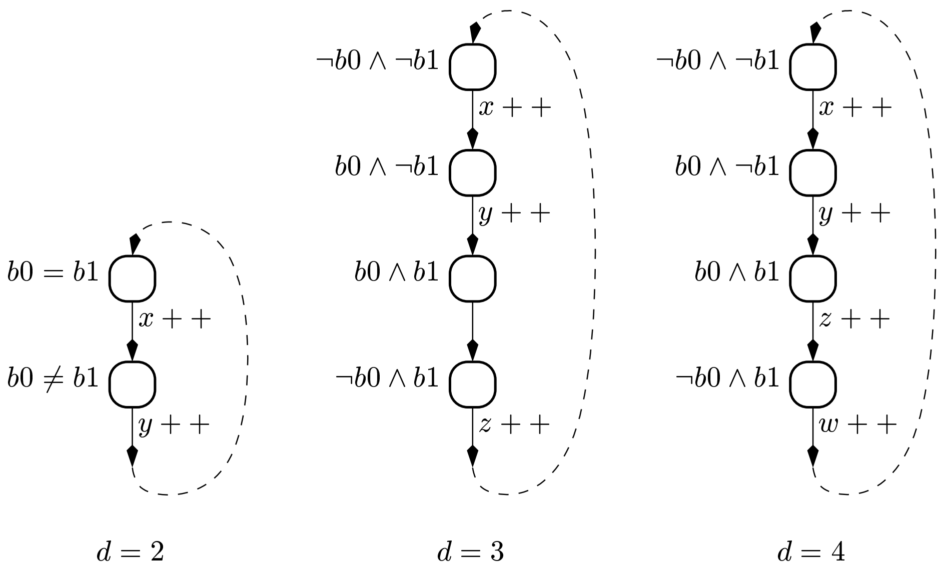


FIG. 11.1 – Cycles de base pour un programme `incalt-`d

Dans les sous-colonnes de la colonne **Vérification**, **bdd** indique le nombre maximum de nœuds de BDDs actifs au cours de la vérification[3], **ana** donne le nombre de combinaisons analyse avant/analyse arrière effectuées, **div** donne le nombre de divisions de place effectuées, en distinguant division sur des booléens ou des contraintes, et **taille max** la taille maximale de la structure de contrôle, en nombre de places et nombre d'arcs. **temps** donne le temps total en en minutes et secondes, et **mem** l'occupation mémoire maximale en Mo.

11.1 Cas d'école

11.1.1 Incrémentation cyclique et alternée de compteurs

Présentation de l'exemple

Nous allons d'abord considérer notre programme d'exemple introduit au chapitre 4, page 33, qui consiste à effectuer deux cycles d'incrémentation alternée de deux compteurs.

Nous l'avons généralisé dans deux sens : en augmentant le nombre de cycles, et en augmentant le nombre de variables. `incalt-`d`-`n indique un tel programme avec d compteurs et 2^n cycles. La figure 11.1 donne les cycles de base pour $d \in \{2, 3, 4\}$. Dans tous les cas, la propriété sera "retardée" dans une variable booléenne ok, de sorte que la structure de contrôle initiale ait 3 places et pour compliquer un peu la preuve.

Les vérifications ont été menées avec les options `-bdd 2ms -simpl s0r0 -widening dpr1f1 -div a`. Le tableau 11.1 montre les résultats obtenus pour la vérification de la propriété $x \geqslant y$. Le tableau 11.2 montre les résultats obtenus en vérifiant la propriété $y \geqslant z$, lorsque celle-ci a un sens. Enfin le tableau 11.3 compare les résultats obtenus lorsqu'on vérifie que $x \geqslant y \geqslant z[\geqslant w]$.

[3]tel qu'indiqué par la bibliothèque CUDD.

	Complexité				Vérification				
	bool	num	cont	auto	ana	div	taille max	temps	mem
incalt-2-1	3/0	2/0	1	5	2	0/1	4/4	$\approx$ 0.3s	-
incalt-2-2	4/0	2/0	1	9	2	0/1	4/4	$\approx$ 0.3s	-
incalt-2-4	6/0	2/0	1	33	2	0/1	4/4	$\approx$ 0.4s	-
incalt-3-1	4/0	3/0	1	9	2	0/1	4/4	$\approx$ 0.3s	-
incalt-3-2	5/0	3/0	1	17	2	0/1	4/4	$\approx$ 0.3s	-
incalt-3-4	7/0	3/0	1	65	2	0/1	4/4	$\approx$ 0.6s	-
incalt-4-1	4/0	4/0	1	9	2	0/1	4/4	$\approx$ 0.3s	-
incalt-4-2	5/0	4/0	1	17	2	0/1	4/4	$\approx$ 0.4s	-
incalt-4-4	7/0	4/0	1	65	2	0/1	4/4	$\approx$ 0.7s	-

TAB. 11.1 – Vérification de la propriété $x \geqslant y$

	Complexité				Vérification				
	bool	num	cont	auto	ana	div	taille max	temps	mem
incalt-3-1	4/0	3/0	1	9	2	0/1	4/4	$\approx$ 0.3s	-
incalt-3-2	5/0	3/0	1	17	2	0/1	4/4	$\approx$ 0.3s	-
incalt-3-4	7/0	3/0	1	65	2	0/1	4/4	$\approx$ 0.6s	-
incalt-4-1	4/0	4/0	1	9	2	0/1	4/4	$\approx$ 0.3s	-
incalt-4-2	5/0	4/0	1	17	2	0/1	4/4	$\approx$ 0.4s	-
incalt-4-4	7/0	4/0	1	65	2	0/1	4/4	$\approx$ 0.7s	-

TAB. 11.2 – Vérification de la propriété $y \geqslant z$

	Complexité				Vérification				
	bool	num	cont	auto	ana	div	taille max	temps	mem
incalt-3-1	4/0	3/0	2	9	6	2/4	9/12	$\approx$ 0.5s	-
incalt-3-2	5/0	3/0	2	17	6	2/4	9/12	$\approx$ 0.6s	-
incalt-3-4	7/0	3/0	2	65	6	2/4	9/12	$\approx$ 1.5s	-
incalt-4-1	4/0	4/0	3	9	9	3/6	10/14	$\approx$ 1.0s	-
incalt-4-2	5/0	4/0	3	17	9	3/6	10/14	$\approx$ 1.3s	-
incalt-4-4	7/0	4/0	3	65	9	3/6	10/14	$\approx$ 3.5s	-

TAB. 11.3 – Vérification de la propriété $x \geqslant y \geqslant z[\geqslant w]$

	Complexité				Vérification				
	bool	num	cont	auto	ana	div	taille max	temps	mem
incalt-3-4	7/0	3/0	2	65	6	1/1	5/9	$\approx$ 0.9s	-
incalt-4-4	7/0	4/0	3	65	5	4/3	9/11	$\approx$ 1.9s	-

TAB. 11.4 – Vérification de la propriété $x \geqslant y \geqslant z[\geqslant w]$ avec l'option `-div ab`

Commentaires

1. Les trois tables montrent que le nombre de cycles n'a ici aucune influence sur les divisions effectuées. En revanche il a une influence sur le temps d'exécution. Cela provient du fait que le nombre d'itérations des analyses augmente. Par exemple pour **incalt3-1** le nombre d'itérations des analyses avant est de 9, alors que pour **incalt-3-4** il est de 65. Il s'agit tout simplement du diamètre booléen du système qui augmente.
2. Le fait de vérifier la propriété $x \geqslant y$ ou $y \geqslant z$ ne change pas le comportement de l'outil. Dans les deux cas, la division selon la contrainte concernée suffit à la preuve.
3. Lorsqu'on vérifie que $x \geqslant y \geqslant z[\geqslant w]$, les choses deviennent plus compliquées, car les possibilités de division sont plus étendues et il faut extraire des conditions de division à partir de formules discriminantes plus générales. Pour ce dernier exemple, si on remplace l'option `-div a` par `-div ab`, on obtient la table 11.4.

Sur cet exemple jouet, on peut donc observer que la complexité de la vérification dépend plus de la propriété que du programme, ce qui est un des objectifs revendiqués de notre méthode.

11.1.2 Un contrôleur de lecteur de disque

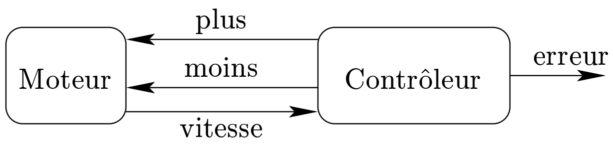


FIG. 11.2 – Contrôleur de disque

Prenons maintenant un exemple dans lequel le contrôle numérique est important et un peu subtil.

Nous considérons un contrôleur de lecteur de disque (simplifié) relié à un moteur, figure 11.2. Le moteur réagit à deux signaux, **plus** et **moins**, qui le fait tourner respectivement plus ou moins vite. En l'absence de ces signaux, la vitesse varie entre deux bornes (ici , -4 et +4). Un capteur donne la vitesse courante **vitesse**. Le contrôleur essaie de maintenir la vitesse dans un intervalle spécifié (ici [8,12]) en envoyant les signaux appropriés **plus** et **moins**. Quand la vitesse a été en dehors de cet intervalle pendant un délai donné (ici, 15), compté par un nombre de pas, un signal **erreur** est envoyé.

On veut vérifier que le contrôleur du programme de la figure 11.3 n'émet jamais un message d'erreur. Ceci signifie que si la vitesse est initialement nulle, elle doit atteindre à temps l'intervalle correct et être correctement régulée après.

On utilise les mêmes options que dans l'exemple précédent, ce qui donne les résultats suivants :

	Complexité				Vérification				
	bool	num	cont	auto	ana	div	taille max	temps	mem
lecteur	3/0	2/1	9	5	3	2/0	5/13	0.6s	-

Les deux divisions booléennes qui ont eu lieu ont été effectuées sur les valeurs de `pre plus` et `pre moins`. Si on avait divisé sur tous les booléens et les contraintes d'état, on aurait obtenu une structure de contrôle avec 4 x 5 x 2 = 40 places (4 états booléens, 5 intervalles pour **pre vitesse** et 2 intervalles pour **pre cpt**). L'outil se comporte donc bien même sur de petits exemples pour lequel il n'est pas conçu.

```
node Environnement(diff: int; plus,moins: bool) returns (ok: bool);
let
  ok = (-4 <= diff and diff <= 4) and
     (if (true -> pre plus) then diff >= 1 else true) and
     (if (false -> pre moins) then diff <= -1 else true);
tel

node Controleur(diff: int) returns (vitesse: int; plus,moins: bool);
let
  vitesse = 0 -> pre(vitesse)+diff;
  plus = vitesse <= 9;
  moins = vitesse >= 11;
tel

node Propriete(vitesse: int) returns (ok: bool);
var cpt: int;
    acceptable: bool;
let
  acceptable = 8 <= vitesse and vitesse <= 12;
  cpt = 0 -> if acceptable then 0 else pre(cpt)+1;
  ok = true -> (pre cpt<15);
tel

node Lecteur(diff:int) returns (ok: bool);
var vitesse: int;
    plus,moins,realiste: bool;
let
  (vitesse,plus,moins) = Controleur(diff);
  realiste =  Environnement(diff,plus,moins);
  assert(realiste);
  ok = Propriete(vitesse);
tel
```

FIG. 11.3 – Le programme du lecteur de disque

11.1.3 Un régulateur de métros

Présentation de l'exemple

Nous terminons par un exemple un peu plus compliqué, qui est le régulateur de métro de [HPR97] qui nous a servi d'exemple au §3.3. Nous incitons le lecteur à se référer au §3.3.1 pour une description détaillée. Aux hypothèses qui y sont décrites, nous ajoutons les deux suivantes :

- un métro ne peut être dans l'état "en avance" pendant plus de 10 pas, ce qui est modélisé par le fait que dans ce cas il ne reçoit plus de signaux **Balise** ;
- le fait qu'un métro soit dans l'état "en retard" gèle l'horloge centrale envoyant le signal **Seconde** à tous les métros.

Notons que la seconde hypothèse fait que les métros ne sont pas indépendants.

Nous allons faire varier le nombre de métros. Dans `metro-`n`-`p, n indique le nombre de métros et p la propriété. Les propriétés que nous allons montrer sont :

1. Un métro ne peut passer en un seul pas de l'état "en retard" à l'état "en avance" et vice-versa. Quand il y a plusieurs métros, nous ne la vérifions que sur le premier.
2. L'écart $\#Balise - \#Seconde$ est compris entre -10 et 20 (rappelons que le régulateur doit maintenir cette valeur autour de 0).
3. Lorsqu'il y a au moins deux métros, supposés être sur deux voies parallèles, l'écart du premier avec le second est inférieur ou égal 40 balises.
4. Idem que la propriété précédente, sauf qu'on montre cette fois-ci la borne réelle, qui est de 30 balises.
5. Même propriété que précédemment, mais cette fois-ci c'est *la valeur absolue* de leur écart est inférieur ou égal 30 balises. Ceci signifie que s'ils sont sur la même voie et initialement distants d'au moins 30 balises, ils ne se rencontreront jamais.

Les vérifications ont été menées avec les options `-bdd 2ms -simpl s1r1 -widening dpr0f1 -div a` et donnent les résultats de la table 11.5.

	Complexité				Vérification					
	bool	num	cont	auto	bdd	ana	div	taille max	temps	mem
metro-1-1	4/2	3/0	11	8	2.6e3	3	0/3	6/13	0.9s	-
metro-1-2	3/2	4/0	13	5	2.5e3	3	0/2	5/9	1.3s	-
metro-2-1	6/3	5/0	22	29	1.2e4	3	1/2	6/14	12.6s	46
metro-2-2	5/3	6/0	24	17	1.3e4	4	1/5	9/27	15.8s	54
metro-2-3	5/3	5/0	23	17	1.1e4	4	1/6	10/37	9.1s	37
metro-2-4	5/3	5/0	23	17	1.1e4	5	3/10	16/67	14.3s	39
metro-2-5	5/3	5/0	24	17	6.2e4	échec après 47/556 divisions			23m14s	148
metro-3-1	8/4	7/0	33	113	1.6e5	3	1/2	6/14	5m27s	188
metro-3-2	7/4	8/0	35	65	9.6e4	4	2/4	9/29	4m53s	192
metro-3-3	7/4	7/0	34	65	9.3e4	4	2/5	10/39	10m10s	$\approx$ 200
metro-3-4	7/4	7/0	34	65	9.3e4	5	6/8	17/93	14m53s	$\approx$ 200

TAB. 11.5 – Vérification du régulateur de métro

Commentaires

1. On constate d'abord que le temps de vérification augmente de manière dramatique avec le nombre de métros considérés, même lorsque la propriété ne porte que sur un seul des métros. Ceci provient du fait que pour que le signal **Seconde** de l'horloge centrale puisse être émis, il faut qu'*aucun* métro ne soit en retard. Par conséquent, même si la propriété ne porte que sur un seul métro, il faut prendre en compte le comportement de tous les métros, et en particulier le nombre de variables numériques croît rapidement.

   Mentionnons aussi que même si ce système est petit, les fonctions de transition sont assez compliquées et comporte de nombreux tests sur des contraintes. Le §11.4.1 analyse plus en détail le comportement interne de l'outil, et montre qu'une grande partie du temps de calcul est consacré à la séquentialisation et aux duplications d'arcs lors des raffinements.

2. La comparaison de la preuve des propriétés 3 et 4 est instructive. Il s'agit à chaque fois de montrer le respect d'une borne supérieure pour la même quantité, mais dans le second cas la borne est la borne réelle, ce qui rend la vérification un peu plus difficile, surtout lorsqu'il n'y a que 2 métros.
3. De manière curieuse, la preuve de la propriété 4 réclame moins de divisions avec 3 métros qu'avec 2 métros. Ceci provient du fait que lorsqu'on a le choix entre plusieurs variables ou contraintes pour diviser une place, on choisit celle dont le rang est le plus petit dans les BDDs ; un changement de l'ordre des variables dans les BDDs influence par conséquent le processus de raffinement.
4. Une observation intéressante est que la preuve de la propriété 5 est beaucoup plus difficile que celle de la propriété 4. La propriété 4 consiste à montrer que pour une certaine quantité Δ, $\Delta \leqslant 30$. Montrer $\Delta \geqslant -30$ se fait exactement de la même manière. En revanche la propriété 5 consiste à montrer en même temps $-30 \leqslant \Delta \leqslant 30$, et on n'y arrive pas avec les heuristiques qui ont été définies[4], même pour deux métros. Ceci est sans doute du au fait que la violation de la propriété peut être causée par deux raisons symétriques mais opposées, et que dans ce cas les approximations sont très pénalisantes ; notons cependant que ce phénomène ne se produit pas pour la propriété 2. Ainsi, encore plus que pour les méthodes de vérification exacte, il peut être déterminant de décomposer une propriété en sous-propriétés indépendantes.

11.2 Un contrôleur d'ascenseur

11.2.1 Présentation du programme

Nous abordons ici la vérification d'un programme sensiblement plus gros — et plus réaliste — que les cas d'école qui précèdent. Il s'agit d'un contrôleur d'ascenseur qui sert d'exemple illustratif dans la thèse [Bou99]. L'ascenseur a n niveaux. À chaque niveau se trouvent des boutons de montée et de descente, et dans la cabine n boutons permettant à l'utilisateur de spécifier l'étage auquel il veut se rendre. Les demandes multiples sont mémorisées, ce qui produit un contrôle complexe. Par ailleurs le contrôleur s'occupe de positionner de manière adéquate divers voyants lumineux. Concernant la cabine, l'ouverture et la fermeture des portes sont modélisées, ainsi que les mouvements, décomposés par demi-étage pour le démarrage et l'arrivée.

Nous avons modifié le programme original en rajoutant un compteur introduisant un délai pour l'ouverture et la fermeture des portes de la cabine, ainsi qu'un second compteur pour les délais de montée et de descente de la cabine.

11.2.2 Description des propriétés à prouver et résultats

Dans `asc`-n-p, n indique le nombre d'étages et p la propriété à montrer. Les propriétés que nous voulons montrer sont :

1. L'ascenseur est au plus dans un des états suivants : en train d'ouvrir la porte de la cabine, de la fermer, de monter ou de descendre.
2. Nous montrons une borne inférieure pour le délai entre le début de la fermeture de la porte au niveau 0 et la fin de son ouverture au niveau 1.

[4] bien sûr des divisions subséquentes “au hasard” devraient permettre de conclure

3. Idem que la précédente, exceptée que la borne est la borne réelle.
4. Nous montrons une borne inférieure entre deux départs de l'ascenseur du niveau 0 (mais qui n'est pas la borne exacte). Ceci implique qu'entre-temps, l'ascenseur est monté puis redescendu.
5. Idem que la précédente, excepté que le niveau 0 est remplacé par le niveau 1, à partir duquel l'ascenseur peut monter *ou* descendre.

La taille du programme et des différentes propriétés est d'environ 700 lignes de LUSTRE. Les vérifications ont été menées avec les options `-bdd 2ms -simpl s0r1 -widening dpr1f1 -div a` et les résultats obtenus sont donnés par la table 11.6.

	Complexité				Vérification					
	bool	num	cont	auto	bdd	ana	div	taille max	temps	mem
asc-3-1	26/7	2/0	5	455	1.1e5	sans objet			1.8s	-
asc-3-2	26/7	3/0	6	497	3.8e5	6	18/9	30/75	16.9s	52
asc-3-3	26/7	3/0	6	497	3.8e5	8	44/26	73/188	21.5s	53
asc-3-4	26/7	3/0	6	489	3.9e5	2	1/0	4/6	15.7s	72
asc-3-5	26/7	3/0	6	489	2.5e5	4	3/4	10/26	21.2s	69
asc-4-1	32/10	2/0	5	3235	4.6e5	sans objet			11.5s	-
asc-4-2	32/10	3/0	6	$\approx$	8.3e5	4	5/2	10/26	1m03s	$\approx$ 90
asc-4-3	32/10	3/0	6	$\approx$	8.3e5	8	29/15	47/103	1m12s	$\approx$ 90
asc-4-4[5]	32/10	3/0	6	$\approx$	9.7e5	4	4/2	9/19	1m02s	$\approx$ 110
asc-4-5	32/10	3/0	6	$\approx$	1.6e6	6	12/10	25/67	1m34s	$\approx$ 150
asc-5-1	38/13	2/0	5	?	1.7e6	sans objet			1m12s	73
asc-5-2	38/13	3/0	6	?	3.4e6	4	5/2	10/26	8m35s	$\approx$ 130
asc-5-3	38/13	3/0	6	?	3.4e6	11	87/30	118/350	9m30s	$\approx$ 140
asc-5-4	38/13	3/0	6	?	5.0e6	6	10/6	19/44	9m57s	$\approx$ 310
asc-5-5	38/13	3/0	6	?	5.3e6	6	17/7	27/68	11m13s	$\approx$ 310

TAB. 11.6 – Vérification de l'ascenseur

11.2.3 Commentaires

Ce contrôleur fait typiquement partie des programmes pour lesquels notre technique devrait être efficace : le contrôle fini est très important, alors que la partie numérique est simple, et il est clair que la mémorisation des requêtes de l'ascenseur font que beaucoup d'états peuvent être équivalents vis-à-vis d'une propriété sans être bisimilaires. Ceci se voit sur le fait que la taille maximale de la structure de contrôle au cours du raffinement est bien en-deça de celle de l'automate de contrôle booléen réduit par bisimulation. Notons également que la production et la réduction par bisimulation de l'automate de l'ascenseur lorsqu'il y a 4 étages a nécessité 40 minutes. Nous n'avons donc pas pu mesurer la taille de l'automate booléen correspondant à 5 étages.

Par ailleurs les propriétés de délai représentent un bon test pour nos heuristiques de raffinement. En effet leur preuve nécessite la séparation automatique des états successifs du contrôleur et de l'ascenseur : fermeture des portes, montée ou descente par étage et demi-étage, ouverture de portes, ...

[5] on a du prendre l'option `-simpl s0r0` pour cet exemple

Voici d'autres commentaires plus ciblés :

1. La propriété 1 ne nécessite pas d'analyse dynamique pour être prouvée. Une simple analyse booléenne y parvient (en fait, même en ne prenant pas en compte les contraintes incompatibles).
2. Encore une fois, montrer une propriété de délai avec la borne inférieure *réelle* est plus difficile que montrer le respect d'une borne inférieure plus petite (propriétés 2 et 3) : le raffinement nécessaire dépend de la propriété à prouver.
3. L'ajout d'un quatrième et d'un cinquième étage fait augmenter la consomnation méméoire de manière raisonnable ; les temps de vérification en revanche augmente plus sensiblement.

Bien que nous ne les ayons pas expériemtées, il est très probable que les techniques de [GS97, BLO98], consistant à partitionner l'espace d'état d'après les gardes apparaissant dans le programme, soient ici inopérantes :

- Si l'on ne considère que les gardes des fonctions de transition *numériques*, le partitionnement obtenu n'est pas assez précis, car certains compteurs sont utilisés plusieurs fois dans les propriétés de délai, et il faut alors décomposer la formule gardant l'incrémentation de ces compteurs pour distinguer les différentes utilisations du compteur.
- Si l'on considère les gardes de *toutes* les fonctions de transition, on obtient une structure de contrôle qui est au moins aussi grosse que l'automate de contrôle booléen réduit par bisimulation.

11.3 Modélisation d'une usine chimique et de son contrôleur

Nous abordons ici la modélisation d'une usine chimique contrôlée par un programme et la vérification de propriétés sur ce système, qui est une des quatre études de cas du projet européen ESPRIT 26270, intitulé "Verification of Hybrid Systems" (VHS).

11.3.1 Présentation

La fonction principale de l'usine est la production d'une solution de chlorure de sodium d'une concentration fixée, obtenue par mélange d'une solution très concentrée et d'eau pure, ainsi que la séparation d'une solution diluée en ses composants de départ. Le processus est décrit en détail dans [Kow98], et des mesures quantitatives plus détaillées dans [Dep99]. La figure 11.4 montre le diagramme de l'usine.

Modélisation de l'usine

Nous avons modélisé tous les éléments de la figure 11.4, à savoir les 29 valves, les 2 pompes, l'évaporateur (modélisation de la température et de l'ebullition), et la quantité de liquide dans chaque réservoir. Les débits sont supposés constants, alors que dans [Dep99] leur variation en fonction de la quantité de liquide du réservoir concerné varie dans une proprortion de +35% au maximum.

La période d'échantillonage choisie est d'une seconde, et les volumes sont donnés en cm^3, afin d'avoir une discrétisation fine.

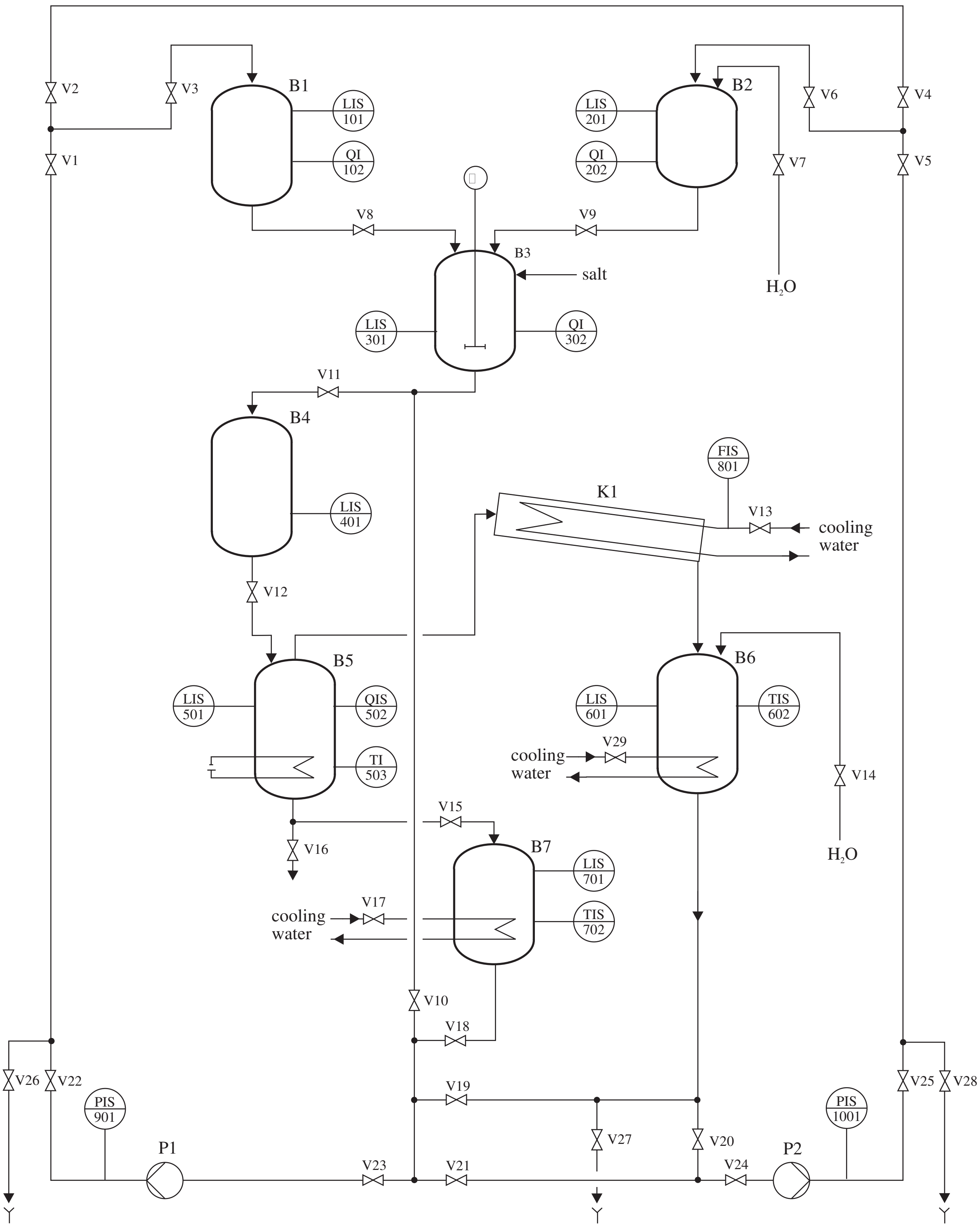


FIG. 11.4 – Diagramme de l'usine chimique

Les réservoirs ont en général plusieurs valves pour les remplir et les vider. Nous n'avons pas modélisé ce qui se passe quand un réservoir est plein. En revanche, à cause de la discrétisation du comportement, il se pourrait que le niveau d'un réservoir deviennent négatif. Pour éviter ceci tout en prenant en compte la conservation de la matière, la quantité de liquide qui quitte un réservoir pour aller dans un autre réservoir en une seconde est le minimum du débit et de la quantité de liquide dans le réservoir qui se vide. Une telle modélisation n'est alors correcte que si plusieurs valves "sortantes" d'un même réservoir ne sont jamais simultanément ouvertes. Ceci est une propriété à vérifier.

Outre la quantité de liquide dans les réservoirs, nous avons aussi modélisé la température des réservoirs B5, B6 et B7, avec la simplification suivante : l'évolution de la température d'un réservoir, par chauffage et ou refroidissement, ne dépend pas de la quantité de liquide présente dans le réservoir, une telle dépendance n'étant pas linéaire.[6]

La concentration ou la qunatité de sel dans chaque réservoir n'est pas modélisée.

Il est à noter qu'une telle modélisation est hors de portée de ce que permettent les automates temporisés, et correspond à une version discrète des automates hybrides linéaires. On peut bien sûr incorporer les entiers bornés représentant l'état des réservoirs dans le contrôle, et alors utiliser des automates temporisés, mais ceci nécessite d'utiliser de petites constantes numériques, sans quoi il y a explosion du nombre d'états de contrôle.

Modélisation du contrôleur

Le contrôleur se décompose en fait en un contrôleur global, s'occupant de la coordination de sous-contrôleurs plus simples et attachés à un réservoir et à ses vannes. Nous avons modélisés les sous-contrôleurs correspondant aux phases d'initialisation du système et de production, à l'exclusion des opérations de rinçage.

Phase d'initialisation. — Avant initialisation, tous les réservoirs sont vide. L'initialisation consiste à produire une solution initiale d'eau salée. Cette phase utilise les programmes suivants :

- B2 Remplissage du réservoir B2 par de l'eau, via la valve V7.
- B3K Production d'une solution concentrée dans le réservoir B3.
- B3U Pompage du contenu du réservoir B3 dans le réservoir B1.

Phase de production. — La "production" fonctionne en fait en vase clos ; à partir d'une solution diluée, on produit par évaporation une solution concentrée. Cette phase utilise les programmes suivants :

- B3 Production d'une solution diluée dans B3.
- B5 Remplissage du réservoir B5 à partir du réservoir B3, chauffage et évaporation partielle, puis transfert dans le réservoir B7.
- B6 Refroidissement de l'eau issue de la condensation dans le réservoir B6.
- B6U Recyclage de l'eau contenue dans le réservoir B6 par pompage dans le réservoir B2 via la pompe P2.

[6] si l'on opère un changement de variable, en modélisant explicitement le temps des opérations, on peut représenter cette dépendance, mais la modélisation physique n'est alors plus indépendante du comportement des contrôleurs.

B7 Refroidissement de la solution concentrée du réservoir B7.

B7U Pompage du contenu du réservoir B7 dans le réservoir B1 via la pompe P1.

Coordination des contrôleurs. — L'ordonnancement des contrôleurs que nous avons choisi est donné par la figure 11.5, dans laquelle les barres verticales sont des barrières de synchronisation. Il y a peu de parallélisme dans le contrôle à cause des contraintes d'exclusion mutuelle : deux contrôleurs ne peuvent commander la même vanne par exemple.

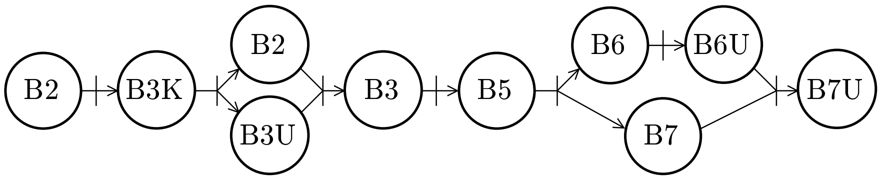


FIG. 11.5 – Contrôleur global

Autres modélisations

Le projet VHS étant un projet européen impliquant de nombreuses équipes, de nombreuses modélisations ont été faites à des fins de vérification de propriétés.

Modélisations non temporisées. — [Luk99] effectue une modélisation non temporisée du système et de *tous les contrôleurs*. Pour le type de propriétés vérifiées, il n'y a pas besoin de modéliser un coordinateur de contrôleurs. Les propriétés vérifiées avec l'outil SMV sont du type suivant :

- lorsqu'un contrôleur termine, les valves, les pompes et le radiateur sont tous fermés ou éteints ;
- lorsque le radiateur est en marche, la valve 13 communiquant avec le condensateur est ouverte et toutes les autres sont fermées ;
- les pompes ne sont jamais en marche lorsqu'une valve en aval est fermée ;
- ...

Dans ce type de modèle, la modélisation physique est donc limitée à l'état des vannes, pompes et radiateur. L'état des réservoirs (quantité de liquide, temprérature, salinité) est très abstrait le réservoir est soit vide, soit plein ; de ce fait les contrôleurs qui utilisent des capteurs de niveaux et de température sont également abstraits (par indéterminisme). Ces abstractions sont correctes du moment qu'on ne cherche à prouver que des propriétés d'incompatibilités de certains évènements ou des contraintes d'ordre entre évènements.

Modélisations temporisées. — [Huu99] modélise le comportement du système d'évaporation, prouve un certain nombre de propriétés fines sur ce modèle, réduit par rapport au système global. [KLPW99] effectue une modélisation assez proche de la nôtre avec des automates temporisés. Leur modèle est plus simple dans le sens qu'il ne considère que 7 valves, tandis que les quantités dans les réservoirs sont représentées en litres par des entiers de petites taille (< 10) de façon à pouvoir les intégrer dans le contrôle discret, et les débits sont inversés : ils sont utilisés sous la forme “secondes par litre”. Ils ne modélisent pas le fait que *plusieurs* valves sortantes d'un réservoir *pourraient* être simultanément ouvertes. Dans notre cas, la prise en compte de cette possibilité complique les fonctions de transition. Parmi les diverses propriétés prouvées :

- propriétés de cohérence : quantités de liquide dans les réservoirs positives et inférieures au maximum (grâce à l'action des contrôleurs), et conservation de la masse.
- pour chaque réservoir, il n'y a qu'une valve (entrante ou sortante) ouverte à un instant donné ;
- les propriétés prouvées sur le modèle non temporisé cité ci-dessus.

Les limitations de ce modèle, comparé au nôtre, sont les suivantes :

- On ne considère que 7 valves, ce qui oblige aussi à abstraire les contrôleurs ;
- La discrétisation des quantités de liquide est beaucoup plus grossière (elles sont mesurées en litres, au lieu de millilitres dans notre modèle) ; en particulier, quand un réservoir se vide, la quantité de liquide va être décrémentée tous les d secondes ; si on ferme la vanne concernée en cours de comptage, la quantité n'est pas décrémentée.

Comparaison avec notre modèle. — Notre modèle correspond à celui de [KLPW99], avec les différences suivantes :

- Nous modélisons *toutes* les valves, et le système des tuyaux ; de plus notre modélisation du système physique est beaucoup plus indépendante de celle des contrôleurs ;
- Les quantités numériques ne sont pas incorporées dans le contrôle et la discrétisation est plus fine (les constantes sont de l'ordre de 10^3 ou 10^4) ;
- En revanche les débits sont constants dans notre modèle, alors que [KLPW99] distingue deux modes, suivant que le réservoir qui se vide est plus ou moins rempli.

11.3.2 Description des propriétés à prouver

Les propriétés prouvées dans la modélisation non temporisée [Luk99] s'effectuent toutes aisément avec notre outil, car elles se prouvent sur l'abstraction booléenne de notre programme, même sans prise en compte des gardes statiquement insatisfaisables. Il en est de même pour certaines propriétés montrées dans [Huu99], comme celle-ci : pour chaque réservoir, il n'y a qu'une valve ouverte en même temps.

Notons aussi que comme la modélisation temporisée de [Huu99] représente le contenu des réservoirs par des entiers finis incorporés dans le contrôle, certaines vérifications sont en fait des vérifications de type fini ; par exemple la vérification de la conservation de la masse peut se faire sans construire de zones, uniquement en construisant l'automate temporisé global et en supprimant les transitions statiquement insatisfaisables.

Propriétés de contrôle — Les propriétés suivantes sont des propriétés portant sur les contrôleurs et qui ont un sens dans une modélisation sans quantités numériques. En principe, elles peuvent être prouvées sur une abstraction booléenne du programme.

1. Lorsque les contrôleurs sont tous inactifs, les vannes, pompes et radiateur sont tous fermés.
2. Les contrôleurs actionnant au moins un dispositif commun (vanne, pompe ou radiateur) sont en exclusion mutuelle.

3. Lorsque le radiateur fonctionne, la valve 13 communiquanrt avec le condensateur est ouverte et toutes les autres sont fermées.
4. Pour chaque réservoir, il n'y a qu'une valve (entrante ou sortante) ouverte à un instant donné.

Propriétés numériques — Les propriétés qui suivent nécessitent une analyse du comportement numérique du système.

1. Le contenu des réservoirs est toujours positif ou nul.
2. Après la phase d'initialisation, il y a conservation de la quantité de liquide dans le système. On décompose cette propriété d'égalité en deux propriétés d'inégalités.
3. (a) Durée de la la phase d'initialisation supérieure à 1200s
   (b) Durée de la la phase d'initialisation inférieure à 1300s
4. (a) Durée de la phase de production supérieure à 1500s
   (b) Durée de la phase de production supérieure à 3000s
   (c) Durée de la la phase d'initialisation inférieure à 4000s

   La durée réelle se situe entre ces deux dernières valeurs.
5. On considère maintenant la durée d'une partie de la phase de production, s'arrêtant avec la fin du sous-contrôleur B5.
   (a) Durée supérieure à 1500s
   (b) Durée inférieure à 3000s

11.3.3 Résultats et commentaires

Vérifications sur le système complet. — Les résultats sont donnés table 11.7. Les options utilisées sont `-bdd 2ms -simpl s1r1 -widening dpr1f1 -div ac`.

	Complexité				Vérification					
	bool	num	cont	auto	bdd	ana	div	taille max	temps	mem
vhs-a	97/0	10/0	26	34	2.7e4	sans objet			2.4s	-
vhs-b	84/0	10/0	26	34	2.7e4	sans objet			2.6s	-
vhs-c	97/0	10/0	26	34	2.7e4	sans objet			2.3s	-
vhs-d	91/0	10/0	26	34	2.7e4	sans objet			1.9s	-
vhs-1	85/0	10/0	33	66	7.7e5	9	10/9	19/32	31m34s	$\approx$ 350
vhs-2	85/0	11/0	30	34	7.3e5	21	84/86	78/209	29m01s	$\approx$ 200
vhs-3a	85/0	11/0	27	35	7.5e5	6	12/5	20/40	3m00s	156
vhs-3b	85/0	11/0	27	35	7.4e5	11	45/32	65/197	8m29s	200
vhs4a	85/0	11/0	27	35	?	?	?	?	> 30m	
vhs4b	85/0	11/0	27	35	?	?	?	?	> 30m	
vhs4c	85/0	11/0	27	35	?	?	?	?	> 30m	
vhs-5a	85/0	11/0	27	34	4.5e6	10	49/55	107/333	10m47s	$\approx$ 300
vhs-5b	85/0	11/0	27	34	?	?	?	?	> 30m	

TAB. 11.7 – Vérification de l'usine chimique

Ces expérimentations appellent plusieurs commentaires :

1. Cet exemple est a-priori mal adapté à notre outil. En effet, malgré sa complexité apparente, le contrôle est petit, car les sous-contrôleurs sont ordonnancés en séquence, à part deux exceptions (cf. figure 11.5), et il n'y a aucun indétermininsme. L'automate booléen réduit par bisimulation est en fait petit. En revanche les variables numériques sont nombreuses, et comme en début d'analyse les polyèdres manipulés sont de type hypercubes, les calculs polyédriques sont très coûteux. Cet exemple est surtout un test pour les heuristiques de raffinement, qui doivent retrouver automatiquement la structure de contrôle des différents sous-contrôleurs, et montre l'intérêt qu'il aurait de pouvoir spécifier une structure de contrôle initiale[7].
2. Les propriétés de contrôle se montre comme attendu sans utiliser d'analyse numérique.
3. Il est assez difficile de montrer les propriétés **vhs-1** et **vhs-2** (en terme de temps d'exécution, mais pas en nombre d'analyses). En principe, il n'y aurait pas besoin d'effectuer des divisions de place pour prouver ces propriétés, car elles découlent en fait des fonctions de transition. Mais en raison des unions convexes effectuées dans les séquences, les analyses sur la structure de contrôle initiale ne parviennent pas à montrer ces propriétés. Nous avons essayé de les montrer avec l'option `-seq e` de séquentialisation sans approximation, mais l'outil n'arrive alors pas à construire les séquences en un temps raisonnable.
4. On arrive beaucoup plus facilement à montrer les propriétés de délai pour la phase d'initialisation, et ce de manière précise.
5. En revanche on ne parvient pas à montrer en un temps raisonnable des propriétés de délai précises sur la phase de production complète ou même partielle. Seule la propriété **vhs-5a** est montrée. Notons que ceci est difficile, car l'outil doit d'abord décomposer les opérations de la phase d'initialisation pour connaître précisément l'état de départ des réservoirs pour la phase de production.

Vérifications sur le système décomposé par phase. — Concernant les propriétés numériques, nous avons aussi cherché à les montrer séparément sur chaque phase, ce qui facilite bien sûr la preuve. Cependant le modèle physique reste identique. Pour la phase de production, nous initialisons les réservoirs avec des valeurs proches de celles obtenues par la phase d'initialisation.

Nous avons utilisé les mêmes options que précédemment. La table 11.8 montre les résultats pour la phase d'initialisation, et la table 11.9 ceux concernant la phase de production.

	Complexité				Vérification					
	bool	num	cont	auto	bdd	ana	div	taille max	temps	mem
vhs-1	63/0	8/0	29	24	5.4e4	3	0/2	4/4	7s	35
vhs-3a	68/0	9/0	23	17	5.1e4	6	9/8	20/31	13s	38
vhs-3b	68/0	9/0	23	17	6.7e4	8	26/20	41/90	33s	57

TAB. 11.8 – Vérification del'usine chimique, phase d'initialisation

On constate que les propriétés se montrent beaucoup plus facilement. C'est surtout les temps de calculs qui sont diminués (et moins le nombre d'analyses) , en particulier lors-

[7] Ceci n'est pas possible à partir du langage LUSTRE ; cependant le langage et l'outil MATOU [MRR00] qui implémente la notion de *mode* pour les langages synchrones flot de données devrait le permettre [MR98]

	Complexité				Vérification					
	bool	num	cont	auto	bdd	ana	div	taille max	temps	mem
vhs-1	76/0	10/0	33	44	1.8e5	4	0/3	4/4	5m54s	182
vhs2	76/0	11/0	30	23	1.1e5	18	20/26	50/69	9m50s	112
vhs4a	75/0	11/0	27	24	2.1e5	10	23/13	39/74	7m40s	≈ 200
vhs4b	75/0	11/0	27	24	Échec car un polyèdre atteint plus de 5000 sommets ou contraintes[8]				24m02s	
vhs4c	75/0	11/0	27	24	?	?	?	?	> 30m	
vhs5a	75/0	11/0	27	23	1.4e5	9	12/11	26/53	2m44s	≈ 110
vhs5b	75/0	11/0	27	23	Échec car un polyèdre atteint plus de 5000 sommets ou contraintes				17m32s	

TAB. 11.9 – Vérification del'usine chimique, phase de production

qu'on ne considère que la phase de production, car dans ce dernier cas beaucoup de variables numériques restent égales à une constante, ce qui diminue la dimension réelle des polyèdres.

Les propriétés de délai sur la phase de production **vhs-4b**, **vhs-4c** et **vhs-5b** ne peuvent cependant pas prouvées.

Conclusion.— Cette étude de cas, bien qu'elle ne soit pas favorable à la technique du partitionnement dynamique, montre que notre outil est capable de vérifier des propriétés de sûreté sur des systèmes dont le contrôle numérique est important et complexe, et dont le contrôle booléen, bien que relativement simple, est défini par un nombre de variables élevé.

La modélisation du système physique que nous avons faite est la plus détaillée qui ait été effectuée, à notre connaissance.

Bien que certaines propriétés de délai n'aient pu être prouvées, *toutes* les propriétés montrées sur des modélisation discrètes ont pu être très aisément prouvées, ainsi que des propriétés numériques générales. Notons que si l'on décompose le contrôleur global plus finement, par sous-contrôleur et non seulement par phase, tout en gardant le modèle physique complet, les propriétés de délai deviennent réellement faciles à montrer.

11.4 Comportement interne et effet de quelques options

Nous nous attachons dans ce paragraphe à montrer les effets des options de notre outil sur le processus du raffinement et le temps de vérification nécessaire. Nous montrons également de manière plus détaillée le comportement interne de l'outil.

Nous prendrons nos exemples parmi quelques-uns des problèmes de vérification décrits précédemment dans ce chapitre. Dans tous les cas, nous prendrons les options qui ont servis à prouver les propriétés dans les paragraphes précédents, et ne ferons varier que l'option spécifiée.

[8] Limite que nous avons fixée à la taille des polyèdres pour toutes ces expériences

11.4.1 Séquentialisation

Nous examinons dans ce paragraphe l'effet des options de séquentialisation sur les séquences produites, ainsi que le temps passé dans cette opération. Nous n'avons pas considéré ici les exemples d'incrémentation alternée de compteur, car ils sont trop simples pour que les options aient une réelle influence.

Les résultats sont donnés par un tableau du type suivant :

	Complexité			Opt	Séquences			Temps	
	bool	num	cont	`-seq`	début	1 cycle	fin	total	seq
					acts, b/n, of	acts, b/n, of	acts, b/n, of		

La colonne **Séquences** donne la taille moyenne des séquences à différents moments de l'analyse ; la sous-colonne **début** indique la taille moyenne des séquences avant la première analyse, **1 cycle** donne la même mesure après le premier cycle d'analyse, et **fin** toujours la même mesure sur l'automate final. Une telle taille est donnée par différents nombres : **acts** indiquent le nombre d'actions, **b/n** le nombre d'ouvertures de test, en distinguant les tests sur une variable booléenne (**b**) et ceux portant sur une contrainte numérique (**n**). Enfin **fo** (pour fermeture/ouverture) indique le nombre de fois qu'une cascade de fermetures de test est suivie par une cascade d'ouvertures de test. Plus ce nombre est élevé, plus l'approximation induite par les séquences est importante.

Nous redonnons les temps d'exécution totaux, ainsi que le temps passé spécifiquement dans les algorithmes de séquentialisation et de raffinement de séquence. L'examen du tableau 11.10, qui donne les résultats obtenus en faisant varier les options de séquentialisation, appelle les commentaires suivants :

1. Tout d'abord, l'intérêt d'utiliser une méthode approchée pour calculer pré- et post-conditions apparaît très clairement. En effet, l'option `seq -e` qui consiste à ne pas refermer de tests dans les séquences et donc à ne pas approximer donne des résultats nettement moins bons que lorsqu'on autorise la fermeture de tests. Nous avons interrompu l'expérience sur le problème **metro-3-2**, alors que l'outil en était encore à la construction de la structure de contrôle initiale. De plus, on observe[9] que les approximations induites par les séquences ne sont pas pénalisantes pour le raffinement de la structure de contrôle.

2. Les options `-seq b` ou `-seq n` n'ont pas une influence déterminante sur les séquences obtenues et les temps d'exécution.

3. Il arrive pour certains exemples que la taille moyenne des séquences soit plus importante après le premier cycle d'analyse. Ceci est en réalité du au fait que des arcs dont les séquences étaient simples ont été éliminées, ce qui a fait augmenter la moyenne. En général, le premier cycle d'analyse permet de réduire sensiblement la taille des séquences et le nombre de tests ; l'exemple de l'ascenseur le montre bien.

4. Pour certains exemples, comme celui du métro ou de l'usine, la majeure partie du temps est passé dans la séquentialisation et le raffinement de séquence. Il conviendrait d'analyser plus en détail ce phénomène et d'y remédier.

[9] mais ceci n'apparaît pas dans la table.

	Complexité			Opt	Séquences			Temps	
	bool	num	cont	`-seq`	début	1 cycle	fin	total	seq
lecteur	3/0	2/1	9		19.7, 4.7/0.7, 1	23.6, 6.3/0, 1	7.3, 1.1/0, 0	0.59s	0.24s
				`b`	20.0, 4.7/0.2, 0.7	23.6, 6.3/0, 1.0	7.3, 1.1/0, 0	0.58s	0.24s
				`n`	19.7, 4.7/0.7, 1	23.6, 6.3/0, 1	7.3, 1.1/0, 0	0.59s	0.24s
				`e`	24.7, 4.7/0.2, 0	29.6, 6.3/0, 0	8.1, 1.1/0, 0	0.62s	0.26s
metro-2-2	5/3	6/0	24		105, 37/21, 7.8	136, 54/28, 10.7	39, 10/7, 2.2	16.6s	11.7s
				`b`	115, 40/20, 7.1	147, 57/26, 9.7	43, 10/7, 1.9	15.4s	10.3s
				`n`	98, 39/26, 7.3	117, 55/30, 9.5	65.6, 26/16, 4.3	12.8s	8.8s
				`e`	491, 78/162, 0	492, 87/165, 0	109, 15/32, 0	56.1s	42.4s
metro-3-2	7/4	8/0	35		473, 248/67, 35	637, 359/90, 48	233, 84/38, 16.2	295s	216s
				`b`	482, 249/62, 35	651, 361/84, 48	267, 87/43, 16.2	476s	195s
				`n`	532, 336/79, 25	643, 468/93, 36	233, 84/38, 16.2	313s	214s
				`e`	?	?	?	> 900s	
asc-3-3	26/7	3/0	6		292, 8/106, 4.3	169, 9/7, 2.3	36, 0.2/0, 0	17.8s	8.7s
				`b`	227, 11/9, 2.0	192, 11/7, 2.0	36, 0.2/0, 0	17.3s	8.4s
				`n`	292, 8/106, 4.3	169, 9/7, 2.3	36, 0.2/0, 0	27.7s	8.5s
				`e`	628, 8/59, 0	361, 8/20, 0	29, 0/0, 0	47.4s	28.7s
asc-4-3	32/10	3/0	6		397, 8/176, 4.6	200, 8/7, 2.3	40, 0.2/0.1, 0	70.0s	17.3s
				`b`	246, 8/8, 2.3	200, 8/7, 2.3	40, 0.2/0.1, 0	64.5s	15.1s
				`n`	397, 8/176, 4.6	200, 8/7, 2.3	40, 0.2/0.1, 0	65.4s	17.4s
				`e`	860, 8/82, 0	425, 8/20, 0	36, 0.0/0.2, 0	80.7s	25.7s

TAB. 11.10 – Influence de la séquentialisation

11.4.2 Élargissement

Nous nous intéressons ici sur l'effet de l'élargissement sur la précision des analyses. Nous allons en fait mesurer cette précision indirectement, en examinant le nombre de divisions nécessaires pour prouver une propriété sur un programme et en faisant varier les options d'élargissement. Nous avons dans ce paragraphe remplacé l'exemple **metro-n-2** par **metro-n-3**, qui exige une analyse plus fine du comportement numérique.

Annonçons-le tout de suite, les différents perfectionnement de l'élargissement standard décrits au §8.4 ont *très peu d'influence* sur la vérification de nos exemples. Aussi avons-nous comparé les résultats obtenus avec l'élargissement de base obtenu par `-widening r0f0` et l'élargissement assez évolué obtenu par `-widening dpr1f2`. Nous élargissons à chaque étape.

	Complexité			Vérification		
	bool	num	cont	ana	div	temps
lecteur	3/0	2/1	9	3	2/0	échec
				3	2/0	0.5s
metro-2-4	5/3	6/0	23	5	3/10	14.4s
				5	3/10	25.4s
metro-3-4	7/4	8/0	34	5	6/8	14m55s
				5	6/8	15m11s
asc-3-3	26/7	3/0	6	9	42/24	17.9s
				9	44/23	18.5s
asc-4-3	32/10	3/0	6	10	53/25	1m14s
				8	29/15	1m11s

TAB. 11.11 – Influence de l'élargissement

Le tableau 11.11 compare les deux options pour les différents exemples. On n'observe une différence que pour **lecteur**, **asc-3-3** et **asc-4-3**, et dans les deux derniers cas très faible en temps. De manière générale, quelques divisions supplémentaires permettent de compenser l'approximation plus grande engendrée par l'élargissement standard. Il faut par ailleurs être conscient que sur des exemples dont le comportement numérique est compliqué et/ou qui possèdent un nombre élevé de variables numériques, l'élargissement évolué engendre des polyèdres plus compliqués, ce qui ralentit les calculs polyédriques. Ceci est faiblement visible sur l'exemple **metro-3-4**.

11.4.3 Algorithmes de raffinement

Nous allons maintenant examiner les différentes options de raffinement. Le nombre de combinaisons étant élevé (les options ne sont pas exclusives), nous avons distingué les suivantes :

- `-div a` : c'est l'option standard, qui consiste à ne diviser que selon des littéraux ;
- pas d'option : on autorise la division selon des formules quelconques (lorsqu'elles ne contiennent pas de contraintes numériques) ;
- `-div ac` : on combine les conditions avant et arrière pour extraire un littéral des conditions discriminantes ; cela a plutôt tendance à favoriser les divisions suivant des booléens ;

- `-div ab` : lorsque l'on extrait un littéral de la partie non convexe d'une condition discriminante, on favorise les booléens au lieu des contraintes numériques ;
- `-div an` : lorsque l'on extrait un littéral du monôme en facteur d'une condition discriminante, on favorise les contraintes numériques au lieu des booléens.

	Complexité			Opt	Vérification		
	bool	num	cont	`-div`	ana	div	temps
incalt-3-2	5/0	3/0	2	`a`	6	2/4	0.6s
					6	2/4	0.6s
				`ac`	7	9/10	1.0s
				`ab`	3	1/1	0.4s
				`an`	6	2/4	0.6s
incalt-4-2	5/0	4/0	3	`a`	9	3/6	1.4s
					9	3/6	1.4s
				`ac`	8	15/11	2.0s
				`ab`	5	4/3	0.8s
				`an`	9	3/6	1.4s
lecteur	3/0	2/1	9	`a`	3	2/0	0.5s
					2	1/0	échec
				`ac`	3	2/0	0.5s
				`ab`	3	2/0	0.5s
				`an`	3	2/0	0.5s
metro-2-2	5/3	6/0	24	`a`	4	1/5	28.8s
					4	1/5	30.5s
				`ac`	6	6/23	5m33s
				`ab`	7	8/25	3m36s
				`an`	4	1/5	31.3s
asc-3-3	26/7	3/0	6	`a`	9	44/23	17.5s
					?	> 660	$>$ 5m26s
				`ac`	8	33/24	16.9s
				`ab`	12	126/58	34.0s
				`an`	13	171/22	43.1s
asc-4-3	32/10	3/0	6	`a`	8	29/15	1m12s
					?	> 600	$>$ 6m
				`ac`	11	143/72	1m43s
				`ab`	8	32/13	1m15s
				`an`	13	254/26	2m45s

TAB. 11.12 – Influence de l'heuristique de raffinement

La table 11.12 donne les résultats obtenus. On constate que l'option standard `-div a` donne toujours des résultats satisfaisants, les autres possibilités donnant soit des résultats proches, soit des résultats franchement mauvais ou des échecs.

11.4.4 Autres observations

Nous terminons par quelques observations importantes, et du reste assez intuitives.

D'abord, pour les exemples de taille importante, le réordonnancement de variables dans les BDDs est *vital* : les fonctions de transitions sont plus réduites, ce qui accélère beaucoup l'algorithme de séquentialisation, *gros* consommateur d'opérations sur les BDDs, ainsi que bien sûr les analyses proprement dites. Le réordonnancement prend parfois quelque temps : de l'ordre de 10 secondes dans le cas de l'ascenseur à 4 étages, et de l'ordre de 50 secondes avec 5 étages.

L'analyse booléenne prenant partiellement en compte les contraintes incompatibles joue également un rôle majeur. Elle est peu coûteuse (jamais plus de 10 secondes sur nos exemples), et si on ne l'effectue pas la construction des séquences sur la structure de contrôle initiale est extrêmement laborieuse, car la séquence produite prend en compte des combinaisons de conditions que l'analyse booléenne élimine. Bien sûr, après une premier cycle d'analyse, tout rentre dans l'ordre grâce au raffinement de séquence. Cette remarque est du reste assez commune, car il est bien connu qu'il est important d'effectuer des analyses moins puissantes et moins coûteuses avant d'attaquer une analyse coûteuse et sophistiquée.

Concernant les options de simplification et d'évaluation partielle (`-simpl`), le bon choix dépend beaucoup de l'exemple. L'option **s0** est nécessaire lorsque les BDDs des fonctions de transition sont gros, mais ne permet pas une très bonne prise en compte du "*care set*" sur les contraintes (cf. §6.2.1). **r1** est de rigueur dans la plupart des cas, mais il se trouve que pour l'exemple **lecteur**, l'utilisation de cette option ne permet pas la preuve, car l'algorithme de raffinement ne trouve plus de condition de division.

11.5 Conclusion

Nous avons montré dans ce chapitre l'efficacité de notre outil de vérification NBAC sur des exemples non triviaux, et même complexes dans le cas de l'ascenseur ou de l'étude de cas du projet VHS. Nos idées directrices ont donc pu être mises en œuvre avec succès, ce qui nous amène aux commentaires suivants :

1. Les approximations, loin d'être pénalisantes, en particulier pour le processus de raffinement, nous permettent de changer d'échelle dans la taille des systèmes traités.
2. Les heuristiques de raffinement automatique se révèlent effectives, ce qui était loin d'être évident au départ, surtout sur des systèmes importants pour lesquels quelques "mauvaises" divisions initiales peuvent pénaliser la suite du processus de raffinement.
3. L'objectif de pouvoir prouver plus facilement des propriétés simples que des propriétés subtiles est atteint : nous avons montré sur plusieurs exemples que lorsque l'on affine la propriété, la preuve réclame plus de divisions de place et plus de temps.
4. D'un point de vue plus technique, le fait d'approximer les calculs de pré- et post-conditions n'est pas pénalisant pour le raffinement de la structure de contrôle, et bien au contraire réduit considérablement la complexité en temps et en espace de l'analyse, ce qui permet en définitive de traiter de plus gros exemples.

Chapitre 12

Conclusion

12.1 Bilan

12.1.1 Motivation initiale et démarche suivie

L'objectif de ce travail était la conception et la mise au point d'une technique de vérification algorithmique, capable de vérifier des propriétés de sûreté sur des systèmes temps-réels exhibant à la fois un comportement booléen complexe et un comportement numérique non trivial.

Pour parvenir à cet objectif, nous sommes parti des deux convictions suivantes :

1. Il est nécessaire de pouvoir traiter les deux aspects, booléen et numérique, de manière *couplée et symbolique* ; en effet généralement dans un programme le comportement des variables booléennes et numériques s'influencent mutuellement, et lorsque le nombre de variables est un peu élevé les techniques symboliques, qui manipulent des ensembles de valeurs, sont souvent nécessaires.
2. Il est tout aussi nécessaire de faire usage d'*approximations* pour maîtriser la complexité du problème de vérification, au lieu de s'en servir uniquement pour surmonter son caractère indécidable.

Devant ce constat, nous nous sommes très naturellement placé dans le cadre théorique de l'interprétation abstraite, qui propose une méthode générale pour l'analyse approchée de programme. Pour satisfaire la première exigence, nous avons choisi un treillis abstrait dont les éléments sont la conjonction d'un ensemble booléen et d'un ensemble numérique, et qui permettent d'établir un lien relativement grossier entre les deux types de propriétés. Pour satisfaire la seconde exigence, nous nous sommes servi de la structure de contrôle pour contrôler d'une manière fine le degré des approximations

L'usage d'approximations est cependant délicat, car il faut trouver un compromis adapté entre précision et complexité : une précision élevée permet de prouver un nombre plus grand de propriétés, au détriment de la complexité des systèmes vérifiés, tandis qu'une faible précision permet d'aborder la vérification de systèmes plus complexes. La solution que nous proposons a donc été influencée par ces observations :

1. La précision requise pour la preuve d'une propriété dépend fortement de celle-ci ; en particulier il est souhaitable que la preuve de propriétés simples soit plus facile que celle de propriétés subtiles ;

2. Les propriétés n'impliquent souvent de manière déterminante que de petites parties du programme vérifié. Cela signifie que la précision requise pour leur preuve n'est pas uniforme : certaines parties du programme doivent être analysées très précisément, tandis que des approximations grossières suffisent pour les autres.

C'est pourquoi la "bonne" structure de contrôle pour la preuve d'une propriété donnée ne peut être trouvée a-priori, et doit être construite progressivement par raffinement d'une structure de contrôle initiale grossière.

12.1.2 Réalisation

La mise en œuvre des idées précédentes s'est faite dans le cadre de la vérification de programmes synchrones, en utilisant les BDDs pour représenter les ensembles booléens et les fonctions de transition, et les polyèdres convexes pour représenter de manière approchée les ensembles numériques. Nous avons conçu et implémenté dans l'outil de vérification NBAC :

- Une méthode originale pour calculer les pré- et post- conditions de valeurs abstraites de manière approchée, de façon à ne pas perdre un temps superflu dans ces opérations lorsque la structure de contrôle limite la précision finale des calculs de point fixe. Les approximations induites par cette méthode s'affinent progressivement, en coordination avec le raffinement de la structure de contrôle.
- Des algorithmes d'analyse combinant analyses avant et arrière. Cette combinaison, qui est classique, est ici très efficace du fait qu'elle est utilisée conjointement avec le raffinement de la structure de contrôle. C'est cette combinaison qui permet la mise en œuvre de l'observation 2.
- Enfin, des heuristiques de raffinement, qui cherchent à séparer les états selon leur comportement futur ou passé et qui se révèlent expérimentalement efficaces : assez puissantes pour trouver des critères de raffinement, tout en ne générant pas trop de divisions inutiles. Ces heuristiques représentent la mise en œuvre de l'observation 1.

Une contribution de caractère plus pratique est la réalisation d'une nouvelle bibliothèque de polyèdres.

Nous avons également évalué de manière détaillée l'outil NBAC sur un certain nombre de programmes, en testant ses capacités et en analysant son comportement interne. La conclusion de ces expériences est que nous arrivons effectivement à vérifier des propriétés non triviales sur des systèmes de taille importante, constituant pour l'un d'entre eux une étude de cas d'un projet européen. Mentionnons que nous n'avons pas connaissance de travaux traitant ces problèmes avec le même niveau de modélisation.

Ceci confirme notre idée selon laquelle les approximations sont *nécessaires* pour la preuve de systèmes de taille moyenne, (et même de petite taille comparée à la taille des programmes LUSTRE industriels, qui est de l'ordre de plusieurs dizaines de milliers de lignes). La démonstration que nous en faisons est intéressante dans la mesure où, dans le domaine de la vérification de systèmes temps-réels, les approximations sont beaucoup plus souvent regardées comme *un mal nécessaire* que comme un moyen permettant de changer d'échelle dans la taille des systèmes vérifiés. Il y a certes dans cette communauté de recherche une culture de l'*abstraction*, mais il s'agit souvent d'abstractions finies et donc relativement "rigides", ou d'abstractions infinies, mais exactes [BG96, ABJ98].

12.1.3 Perfectionnements souhaitables

Deux améliorations seraient souhaitables, à court ou moyen terme.

Une limitation inhérente aux méthodes approchées de vérification est que celles-ci ne peuvent que garantir qu'une propriété est vraie, ou répondre "je ne sais pas". En cas d'échec de la vérification, l'outil s'arrête avec une structure de contrôle décrivant l'ensemble des états dangereux, qui est inexploitable en l'état actuel, mais qui pourrait permettre un diagnostic plus fin. Il faudrait concevoir une méthode cherchant à extraire une trace de contre-exemple à partir de cette structure de contrôle. L'extraction réussie d'une telle trace assurerait que la propriété est fausse et la trace obtenue constituerait un diagnostic d'erreur.

Nous avons aussi vu dans la partie expérimentale que pour un certain nombre de cas, 90% du temps de vérification est consacré à la séquentialisation de la fonction de transition et la spécialisation des séquences. Une analyse plus détaillée de l'algorithme s'impose. Il est possible que cet algorithme soit plus sophistiqué que nécessaire et que sa simplification serait à la fois peu pénalisante pour la précision des analyses et très appréciable pour sa rapidité.

12.2 Perspectives

Les perspectives qu'offre ce travail sont de deux types ; d'une part on peut étendre et améliorer sous différents aspects notre méthode en demeurant dans le même cadre d'application ; d'autre part on peut la généraliser et appliquer la technique du partitionnement dynamique à d'autres contextes.

12.2.1 Extensions

Plusieurs extensions seraient souhaitables pour réduire la complexité des analyses et traiter des programmes plus importants. La complexité des analyses est en grande partie due à la complexité des opérations dans le treillis numérique utilisé.

Il est par exemple bien connu qu'il vaut mieux précéder les analyses sophistiquées et coûteuses par des analyses moins puissantes mais plus économiques. Typiquement une analyse utilisant pour treillis numérique celui des intervalles pourrait précéder les analyses que nous faisons en utilisant le treillis des polyèdres convexes, de la même manière que nous faisons précéder la construction de la structure de contrôle initiale par une analyse d'accessibilité booléenne.

Une autre possibilité pour diminuer le coût en temps et en espace des opérations sur les polyèdres serait de traiter spécifiquement certains polyèdres, à savoir ceux qui sont le produit cartésien de polyèdres de dimension plus petite. Par exemple il est fréquent dans les polyèdres que la valeur admissible de certaines variables ne soient définies que par des contraintes d'intervalle ; dans un tel cas le polyèdre peut se décomposer en le produit d'un hypercube et d'un polyèdre de dimension plus réduite. Or il est important de traiter de manière particulière les hypercubes, car ceux-ci sont définis par un nombre de générateurs exponentiel en leur dimension.

Par ailleurs on pourrait dans un certain nombre de cas remplacer le treillis des polyèdres par un treillis moins coûteux. Nous pensons en particulier que l'emploi du treillis des zones, ou l'emploi du treillis abstrait résultant de la combinaison du treillis des intervalles et de celui des égalités linéaires, seraient tous deux très intéressants. Ces deux treillis permettraient la preuve de beaucoup de propriétés prouvées dans le chapitre 11, en particulier des propriétés de délai qui nécessitent la synchronisation de compteurs et l'encadrement de leur valeur. Une inconnue demeure : ces treillis sont-ils assez précis pour permettre l'obtention de conditions de division pertinentes ? Le fait qu'ils autorisent la preuve de propriétés évoluées sur une structure de contrôle suffisamment détaillée n'implique pas qu'ils puissent apporter suffisamment d'information pour guider le raffinement d'une structure de contrôle plus grossière.

12.2.2 Généralisation

Une première généralisation serait d'appliquer notre outil à la vérification des automates hybrides linéaires [ACH+95], en codant les états discrets des automates par des variables booléennes. Il suffirait juste d'ajouter l'opérateur de passage du temps, ce qui est très simple sur les polyèdres convexes [HPR97]. Notre algorithme de séquentialisation est appliquable à ce type de système, ainsi que nos heuristiques de raffinement. Bien sûr, des ajustements seraient sans doute nécessaires. Nous avons l'intention de mener prochainement un tel travail.

De manière plus générale, notre technique consistant à combiner le treillis booléen avec le treillis des polyèdres convexes peut se généraliser à la combinaison du treillis booléen avec un autre treillis de cardinal infini, pas nécessairement numérique. L'algorithme de séquentialisation et les heuristiques de raffinement sont utilisables avec un treillis de cardinal infini, du moment que l'on dispose sur ce treillis d'un opérateur d'élargissement, et que les conditions apparaissant dans les fonctions de transition et concernant ce treillis soient "complémentables" au sens suivant : pour toute valeur abstraite $a \in A$, il existe un ensemble fini de valeurs abstraites $(a_i^*)_{1 \leqslant i \leqslant n}$ tel que

$$\gamma(a) \cap \left(\bigcup_i \gamma(a_i^*) \right) = \emptyset \quad , \quad \gamma(a) \cup \left(\bigcup_i \gamma(a_i^*) \right) = C$$

Par exemple le treillis des congruences [Gra89] ne pourrait être utilisé, car la formule $x \equiv y[p]$ avec $x, y \in \mathbb{Z}$ et $p \in \mathbb{N}$ n'est pas complémentable.

Une piste intéressante serait ainsi d'appliquer notre technique à la vérification d'automates communiquant par file d'attente. Les travaux existants [BG96, ABJ98] procèdent par construction pour chaque file de son ensemble exact d'états accessibles, et y réussissent lorsque ces ensembles sont représentables par le type de représentation choisie et que les techniques d'accélération convergent. Seul le contenu des files est représenté de manière symbolique. Nous pourrions essayer notre technique sur ce type de systèmes, en utilisant un treillis abstrait pour la représentation des files, sur lequel on dispose d'opérations adaptées de borne inférieure et supérieure et d'élargissement. [Mau99] ouvre des pistes pour la conception d'un tel treillis. Notons que la vérification de ces systèmes est un enjeu industriel, car ils correspondent à la sémantique du langage SDL [IT94] défini par l'industrie et dédié à la programmation de protocoles de télécommunication.

Bibliographie

[ABJ98] P. Aziz Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly analysis of systems with unbounded, lossy fifo channels. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, 1998.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. Henzinger, P. Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science B*, 138:3–34, janvier 1995.

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

[And96] C. André. Representation and analysis of reactive behaviors: a synchronous approach. In *IEEE-SMC'96, Computational Engineering in Systems Applications*, Lille, France, July 1996.

[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. *Optimization Methods and Software*, 10:107–124, 1998.

[BBM97] N. Bjørner, A. Browne, and Z. Manna. Automatic generation of invariants and intermediate assertions. *Theoretical Computer Science*, 173(1), 1997.

[BEF98] B. Büeller, A. Enge, and K. Fukuda. Exact volume computation for convex polytopes: A practical study. In G. Kalai and G. Ziegler, editors, *Polytopes - Combinatorics and Computation*, DMV-Seminar. Birkhäuser Verlag, 1998.

[BFG+91] A. Bouajjani, J.-C. Fernandez, S. Graf, C. Rodriguez, and J. Sifakis. Safety for branching time semantics. In *18th ICALP*. LNCS 510, Springer Verlag, juillet 1991.

[BFG+93] R. I. Bahar, E. A. Frohm, C. M. Gaona, G. D. Hachtel, E. Macii, A. Pardo, and F. Somenzi. Algebraic decision diagrams and their applications. In *ICCAD'93*, 1993.

[BG92] G. Berry and G. Gonthier. The Esterel synchronous programming language: Design, semantics, implementation. *Science of Computer Programming*, 19(2):87–152, 1992.

[BG96] B. Boigelot and P. Godefroid. Symbolic verification of communication protocols with infinite state spaces using QDDs. In *Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, juillet 1996.

[BGP99] Tevfik Bultan, Richard Gerber, and William Pugh. Model-checking concurrent systems with unbounded integer variables: symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4):747–789, juillet 1999.

[BLO98] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, juillet 1998.

[BLP+99] G. Behrmann, K. G. Larsen, J. Pearson, C. Weise, and W. Yi. Efficient timed reachability analysis using clock difference diagrams. In *Computer Aided Verification, CAV'99*, volume 1633 of *LNCS*, juillet 1999.

[BLS96] S. Bensalem, Y. Lakhnech, and H. Saïdi. Powerful techniques for the automatic generation of invariants. In *Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, juillet 1996.

[Bou92a] F. Bourdoncle. Abstract interpretation by dynamic partitionning. *Journal of Functional Programming*, 2(4), 1992.

[Bou92b] F. Bourdoncle. Sémantique des langages impératifs d'ordre supérieur et interprétation abstraite. Thèse Ecole Polytechnique, Paris, 1992.

[Bou93] F. Bourdoncle. Efficient chaotic iteration strategies with widenings. In *International Conference on Formal Methods in Programming and their Applications*, volume 735 of *LNCS*, 1993.

[Bou98] A. Bouali. Xeve: an Esterel verification environment. In *Tenth International Conference on Computer-Aided Verification, CAV'98*, Vancouver (B.C.), June 1998. LNCS 1427, Springer Verlag.

[Bou99] L. Du Bousquet. Test fonctionnel statistique de systèmes spécifiés en LUSTRE. Thèse Université Joseph Fourier, Grenoble, 1999.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8):677–692, 1986.

[Bry91] R. E. Bryant. On the complexity of vlsi implementations and graph-based representations of boolean functions with application to integer multiplication. *IEEE Transactions on Computers*, C-40:205–213, 1991.

[BW96] B. Bollig and I. Wegener. Improving the variable ordering of OBDDs is NP-complete. *IEEE Transactions on Computers*, C-45(8):993–1002, 1996.

[CAM] The CAML language. http://caml.inria.fr/.

[CBM89a] O. Coudert, C. Berthet, and J. C. Madre. Verification of sequential machines using boolean functional vectors. In L. J. M. Claesen, editor, *Formal VLSI Correctness Verification*. North-Holland, novembre 1989.

[CBM89b] O. Coudert, C. Berthet, and J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. In *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[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, janvier 1977.

[CC79] P. Cousot and R. Cousot. Systematic design of program analysis frameworks. In *6th ACM Symposium on Principles of Programming Languages, POPL'79*, San Antonio, janvier 1979.

[CC92a] P. Cousot and R. Cousot. Abstract interpretation frameworks. *Journal of Logic and Computation*, 1992.

[CC92b] P. Cousot and R. Cousot. Comparing the Galois connection and widening/narrowing approaches to abstract interpretation. In M. Bruynooghe and M. Wirsing, editors, *PLILP'92*, Leuven (Belgium), janvier 1992. LNCS 631, Springer Verlag.

[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.

[CG99] C.Mauras and D. Gariou. Simulation symbolique de programmes réactifs synchrones : application à la mise au point et à la vérification du contrôle d'une chaudière. In Hermès, editor, *MSR'99*. ENS Cachan, 1999.

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

[CGS91] C. Courcoubetis, S. Graf, and J. Sifakis. An algebra for boolean processes. In *Conference on Computer-Aided Verification 91, Aalborg (Denmark)*, volume 575 of *LNCS*, juin 1991.

[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), janvier 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.

[CLR94] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to algorithms*. MIT Press, 1994.

[CMZ+93] E. M. Clarke, K. L. McMillan, X. Zhao, M. Fujita, and J. C.-Y. Yang. Spectral transforms for large boolean functions with application to technology mapping. In *30th ACM/IEEE Design Automaton Conference*, 1993.

[Cou77] P. Cousot. Asynchronous iterative methods for solving a fixpoint system of monotone equations. Technical Report IMAG-RR-88, Université scientifique et médicale de Grenoble, 1977.

[CU98] M. A. Colón and T. E. Uribe. Generating finite-state abstractions of reactive systems using decision procedures. In *Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, juillet 1998.

[Dan55] G. B. Dantzig. *Linear programming and extensions*. Princeton University Press, 1955.

[Dep99] A. Deparade. A switched continuous model of VHS case study 1. Draft, University of Dortmund, février 1999.

[Dil89] D. L. Dill. Timing assumptions and verification of finite state concurrent systems. In *Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, June 1989.

[DWT95] D. Dill and H. Wong-Toï. Verification of real-time systems by successive over and under approximations. In *Seventh Conference on Computer-Aided Verification, CAV'95*, Liège (Belgium), juillet 1995. LNCS 939, Springer Verlag.

[FP96] K. Fukuda and A. Prodon. Double description method revisited. In *Combinatorics and Computer Science*, volume 1120 of *LNCS*, pages 91–111. Springer-Verlag, 1996.

[Fuk] K. Fukuda. Web page. http://www.ifor.math.ethz.ch/ fukuda/fukuda.html.

[GJ79] M. R. Garey and D. S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness.* W. H. Freeman and Company, 1979.

[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.

[GS97] S. Graf and H. Saidi. Construction of abstract state graphs with pvs. In *Conference on Computer Aided Verification CAV'97, Haifa*, volume 1254 of *LNCS*, juin 1997.

[Hal79] N. Halbwachs. Détermination automatique de relations linéaires vérifiées par les variables d'un programme. Thèse de troisième cycle, Université de Grenoble, mars 1979.

[Hal93a] N. Halbwachs. Delay analysis in synchronous programs. In *Fifth Conference on Computer-Aided Verification, CAV'93*, Elounda (Greece), juillet 1993. LNCS 697, Springer Verlag.

[Hal93b] N. Halbwachs. *Synchronous programming of reactive systems.* Kluwer Academic Pub., 1993.

[Hal94] N. Halbwachs. BAC, a Boolean automaton checker. Technical report, Verimag Laboratory, February 1994.

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

[HFB93] N. Halbwachs, J.-C. Fernandez, and A. Bouajjanni. An executable temporal logic to express safety properties and its connection with the language LUSTRE. In *Sixth International Symp. on Lucid and Intensional Programming, ISLIP'93, Quebec*, April 1993.

[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, juin 1993. Workshops in Computing, Springer Verlag.

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

[HP85] D. Harel and A. Pnueli. On the development of reactive systems. In *Logic and Models of Concurrent Systems,* NATO *Advanced Study Institute on Logics and Models for Verification and Specification of Concurrent Systems.* Springer Verlag, 1985.

[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, août 1997.

[Huu99] Ralf Huuck. Verifying timing aspects of VHS case study 1. Technical Report (number to be assigned), Christian-Albrechts-Universität zu Kiel, 1999.

[IEE91] Another look at real-time programming. *Special Section of the Proceedings of the IEEE*, 79(9), septembre 1991.

[IT94] ITU-T. Recommendation Z.100. Specification and Description Language (SDL) and Annexes F.2: Static Semantics and F.3: Dynamic Semantics. Technical Report Z-100, International Telecommunication Union – Standardization Sector, Genève, 1994.

[JHR99] B. Jeannet, N. Halbwachs, and P. Raymond. Dynamic partitioning in analyses of numerical properties. In *Static Analysis Symposium, SAS'99*, Venezia (Italy), septembre 1999.

[JPV95] L. J. Jagadeesan, C. Puchol, and J. E. Von Olnhausen. Safety property verification of ESTEREL programs and applications to telecommunication software. In P. Wolper, editor, *7th International Conference on Computer Aided Verification, CAV'95*, Liege (Belgium), juillet 1995. LNCS 939, Springer Verlag.

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

[KLPW99] K. Kristoffersen, K. Larsen, P. Pettersson, and C. Weise. VHS Case Study 1 - Experimental Batch Plant using UPPAAL. BRICS, University of Aalborg, Denmark, mai 1999.

[Kow98] S. Kowalewski. Description of VHS case study 1 "Experimental Batch Plant". Draft. University of Dortmund, Germany, juillet 1998.

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

[LDBL93] M. Le Borgne, Bruno Dutertre, Albert Benveniste, and Paul Le Guernic. Dynamical systems over Galois fields. In *European Control Conference*, pages 2191–2196, Groningen, 1993.

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

[LGLL91] P. LeGuernic, T. Gautier, M. LeBorgne, and C. LeMaire. Programming real time applications with SIGNAL. *Proceedings of the IEEE*, 79(9):1321–1336, septembre 1991.

[Luk99] Ben Lukoschus. An abstract model of VHS case study 1 (experimental batch plant). Technical Report (number to be assigned), Christian-Albrechts-Universität zu Kiel, 1999.

[Mar90] F. Maraninchi. Argos, un langage graphique pour la conception, la description et la validation des systèmes réactifs. Thèse, Université Joseph Fourier, Grenoble, 1990.

[Mar92] F. Maraninchi. Operational and compositional semantics of synchronous automaton compositions. In *CONCUR'92*, Stony Brook, août1992. LNCS 630, Springer Verlag.

[Mau96] C. Mauras. Calcul symbolique et automates interprétés. Technical Report 10, IRCyN, novembre 1996.

[Mau98] L. Mauborgne. Abstract interpretation using typed decision graphs. *Science of Computer Programming*, 31:91–112, mai 1998.

[Mau99] L. Mauborgne. *Representation of sets of trees for abstract interpretation*. PhD thesis, École Polytechnique, 1999.

[MLAH99] J. Møller, J. Lichtenberg, H. R. Andersen, and H. Hulgaard. Difference decision diagrams. In *Computer Science Logic*, The IT University of Copenhagen, Denmark, septembre 1999.

[MR98] F. Maraninchi and Y. Rémond. Mode-automata: About modes and states for reactive systems. In *European Symposium On Programming*, Lisbon (Portugal), mars 1998. Springer verlag.

[MRR00] F. Maraninchi, Y. Rémond, and Y. Raoul. Matou : An implementation of mode-automata into dc. In *Compiler Construction*, Berlin (Germany), mars 2000. Springer verlag.

[MT98] C. Meinel and T. Theobald. *Algorithms and data structures in VLSI design*. Springer-Verlag, 1998.

[NSY92] X. Nicollin, J. Sifakis, and S. Yovine. Compiling real-time specifications into extended automata. *IEEE TSE Special Issue on Real-Time Systems*, 18(9):794–804, septembre 1992.

[PS88] A. Pnueli and M. Shalev. What is in a step. Technical report, Dept. of Applied Mathematics and Computer Science, The Weizmann Institute of Science, Israel, mai 1988.

[Ram94] Norman Ramsey. Literate programming simplified. *IEEE Software*, 11(5):97–105, 1994.

[Rat92] C. Ratel. Définition et réalisation d'un outil de vérification formelle de programmes Lustre: Le système Lesar. Thèse, Université Joseph Fourier, Grenoble, juin 1992.

[Ray91] P. Raymond. *Compilation efficace d'un langage déclaratif synchrone : Le générateur de code* LUSTRE-*V3*. Thèse, Institut National Polytechnique de Grenoble, novembre 1991.

[RHR91] C. Ratel, N. Halbwachs, and P. Raymond. Programming and verifying critical systems by means of the synchronous data-flow programming language LUSTRE. In *ACM-SIGSOFT'91 Conference on Software for Critical Systems*, New Orleans, décembre 1991.

[Rud93] R. Rudell. Dynamic variable ordering for ordered binary decision diagrams. In *ICCAD'93*, 1993.

[Som] F. Somenzi. CUDD: Colorado University Decision Diagram Package. ftp://vlsi.colorado.edu/pub.

[SUM96] H. Sipma, T.E. Uribe, and Z. Manna. Deductive model checking. In *8th International Conference on Computer Aided Verification, CAV'96*, volume 1102 of *LNCS*, juillet 1996.

[TBL99] R. Gerber T. Bultan and C. League. Composite model checking: Verification with type-specific symbolic representations. Technical Report TRCS99-02, University of California, Santa Barbara. Computer Science., janvier 1999. ftp://ftp.cs.ucsb.edu/pub/techreports/TRCS99-02.ps.

[THY93] S. Tani, K. Hamaguchi, and S. Yajima. The complexity of the optimal variable ordering problem of shared binary decision diagrams. In *International Symposium on Algorithms and Computations*. LNCS 762, Springer Verlag, 1993.

[Wil93] D. K. Wilde. A library for doing polyhedral operations. Rapport de Recherche 785, IRISA, décembre 1993.

[WT94] H. Wong-Toï. *Symbolic approximations for verifying real-time systems.* PhD thesis, Dept. of Computer Science, Stanford University, décembre 1994.

Résumé : Ce travail porte sur la vérification, par des méthodes algorithmiques, de propriétés de sûreté de systèmes synchrones ou plus généralement réactifs, présentant à la fois des aspects booléens et numériques. Il propose une technique originale permettant d'augmenter très sensiblement la taille des systèmes vérifiés.

Nous commençons par rappeler le cadre théorique utilisé, qui est celui de l'interprétation abstraite, et une de ses applications, l'analyse de relations linéaires, qui permet d'inférer des relations linéaires liant les variables numériques d'un programme en ses points de contrôle. Cette analyse a été appliquée à la vérification de programmes synchrones numériques et de systèmes hybrides linéaires. Dans ce contexte, deux problèmes se posent: le premier est que la structure de contrôle explicite de ces systèmes est souvent de taille prohibitive, et le deuxième est que la précision de l'analyse est quelquefois insuffisante pour la preuve de certaines propriétés.

Ces deux problèmes s'avérant liés, nous proposons une solution pour obtenir un compromis entre complexité et précision, consistant à utiliser un domaine abstrait combinant propriétés booléennes et numériques, et à se servir d'une structure de contrôle pour ajuster la précision de l'analyse. Celle-ci se sert de la propriété à montrer pour raffiner automatiquement une structure de contrôle initiale grossière, jusqu'à ce que la précision obtenue permette de conclure à la preuve de la propriété.

Un outil de vérification fondé sur ces principes a été développé et expérimenté sur plusieurs exemples significatifs, et pour certains d'entre eux inabordables par d'autres techniques automatiques existantes.

Mots-clés : vérification formelle, interprétation abstraite, analyse de relations linéaires, programmation synchrone

Abstract : This works deals with the verification by algorithmic methods of safety properties of synchronous systems, or more generally reactive systems, that exhibit both Boolean and numerical aspects. It proposes an original technique that allows the size of verified systems to be noticeably augmented.

We first recall the theoretical framework we use, which is the abstract interpretation theory, and one of its applications, the linear relation analysis, that discovers linear relations linking numerical variables of a program in each of its control points. This analysis was applied to the verification of numerical synchronous programs and linear hybrid systems. In this context, two problems arise: the first one is that the explicit control structure of these systems is often of prohibitive size, and the second one is that the analysis is not precise enough for the proof of some properties.

These two problems are in fact related, and we propose a solution to obtain a suitable tradeoff between complexity and precision. We use an abstract domain combining Boolean and numerical properties, and we exploit a control structure to adjust the precision of the analysis. The analysis uses the property to be proved in order to automatically refine an initial rough control structure, until the obtained precision enables the proof of the property.

A verification tool based on these principles was implemented and experimented on several and significative examples, some of which being unaffordable with other automatic verification techniques.

Keywords : formal verification, abstract interpretation, linear relations analysis, synchronous programming