Methodes symboliques pour la verification de processus communicants : etude et mise en oeuvre

Alain Kerbrat

THÈSE

présentée par
Alain KERBRAT

pour obtenir le grade de DOCTEUR
de l'UNIVERSITÉ JOSEPH FOURIER - GRENOBLE I
(arrêtés ministériels du 5 juillet 1984 et du 30 mars 1992)

(Spécialité : Informatique)

Méthodes Symboliques pour la Vérification de Processus Communicants : étude et mise en œuvre

---

Date de soutenance : 29 novembre 1994

Composition du jury :	Président	J. Voiron
	Rapporteurs	R. De Simone
		P. Wolper
	Examinateurs	J.C. Fernandez
		C. Jard
		J. Mossière

Thèse préparée au sein du Laboratoire de Génie Informatique de Grenoble,
puis au sein de l'Unité Mixte de Recherche VERIMAG

Remerciements

Cette thèse a débuté dans le cadre du projet SPECTRE du Laboratoire de Génie Informatique de Grenoble, puis de l'unité mixte VERIMAG. Je tiens à remercier Joseph Sifakis, Directeur de Recherche au CNRS, de m'avoir accueilli dans cette équipe.

Jacques Voiron est professeur à l'Université Joseph Fourier. Je le remercie de m'avoir fait l'honneur d'accepter de présider le jury de cette thèse

Jean Claude Fernandez est Maître de Conférences à l'Université Joseph Fourier. Je le remercie d'avoir accepté d'encadrer cette thèse et de m'avoir soutenu pendant ces trois années

Robert de Simone est Directeur de Recherches à l'INRIA. Ses nombreuses remarques et commentaires lors de son évaluation de mon travail ont beaucoup contribué à améliorer ce document. D'autre part, certains de ses travaux ont été à la base de la première partie de cette thèse

Pierre Wolper est Professeur à l'Université de Liège. Je le remercie d'avoir accepté de juger ce travail

Claude Jard est Chargé de Recherche au CNRS. Je le remercie d'avoir accepté de participer à ce jury

Jacques Mossière est Professeur à l'Intitut National Polytechnique de Grenoble. Je le remercie d'avoir accepté de participer à ce jury

Une partie de ce travail a trouvé sa source dans les travaux passés et récents de Nicolas Halbwachs. Je tiens à lui exprimer ma reconnaissance pour toutes les discussions qui m'ont permis de développer ce qui est devenu la deuxième partie de cette thèse

Je tiens aussi à remercier mes premiers rapporteurs, qui ont eu à affronter les versions plus "brutes" de ce document ; en particulier Laurent Mounier, Pascal Raymond et Anne Rasse, dont les remarques éclairées m'ont permis d'affiner cette présentation

J'ai trouvé un peu de mon inspiration et beaucoup de ma motivation grâce à l'ambiance stimulante qui règne dans l'équipe SPECTRE. J'ai particulièrement bénéficié de cet ambiance dans le bureau que j'ai partagé avec Hubert Garavel et Laurent Mounier. Leur bonne humeur et humour constants m'ont permis d'entretenir un élan parfois un peu hésitant, leurs compétences m'ont permis de focaliser cet élan et de le faire aboutir

Je veux aussi remercier toutes les personnes de l'Unité Mixte, que j'ai pu découvrir et apprécier que ce soit au travers de nos relations professionnelles, sportives ou amicales. En particulier, je tiens à faire une passe amicale à Fix, Patrick, Riadh et Sergio, à saluer Mihaela et Radu, à donner une poignée de main énergique à Alain, Pascal et Yann-Erick, et à exprimer toute mon affection à Anne, Claire, Fabienne, Muriel et Suzanne.

1994 restera pour moi une étape essentielle, avec l'aboutissement de ce travail de thèse, qui est un premier pas dans une vie professionnelle que je veux enrichissante. Je suis aussi devenu père, ce qui me donne d'ores et déjà une vie personnelle épanouissante, que je partage avec bonheur avec Anne Cécile ...

Table des matières

Liste des figures

Introduction

les systèmes distribués

Les systèmes distribués sont la base de la plupart des protocoles de communication qui sont utilisés dans les réseaux d'ordinateurs et dans les systèmes de télécommunications, ainsi que dans les architectures multi-processeurs. L'importance croissante des protocoles de communication dans les systèmes informatiques modernes imposent des contraintes de sûreté et de fiabilité très sévères : le moindre dysfonctionnement d'un système de télécommunications peut affecter des millions de personnes et provoquer la perte de sommes énormes, voire mettre en cause la sécurité de personnes.

Il est donc crucial de pouvoir garantir leur bon fonctionnement avant leur mise en service, c'est-à-dire de valider l'ensemble des comportements possibles du système par rapport aux services qui lui sont demandés. Ce genre de validation est en général hors de portée des techniques classiques de conception de programmes :

- la description du service demandé est souvent donnée en langage naturel, complétée par des diagrammes de transitions. Ce mode de description semi-formel conduit à des définitions imprécises, ambiguës et volumineuses.
- les méthodes de validation classiques sont généralement basées sur une couverture plus ou moins grande des exécutions possibles, par génération et validation de séquences de test. Ces méthodes ne peuvent en général pas être exhaustives, et nécessitent de se placer au plus près des conditions réelles d'utilisation pour être réalistes. Comment tester dans ce cas un protocole devant gérer les communications de plusieurs centaines de machines?

Pour répondre à ces problèmes, des formalismes de description ont été développés, associés à des méthodes de vérification formelle. Ces formalismes, désignés sous le nom de *Formal Description Techniques* (FDT) sont conçus pour permettre la *spécification* des systèmes distribués. Ils sont d'autre part exécutables et peuvent aussi servir à la *programmation* de ces mêmes systèmes.

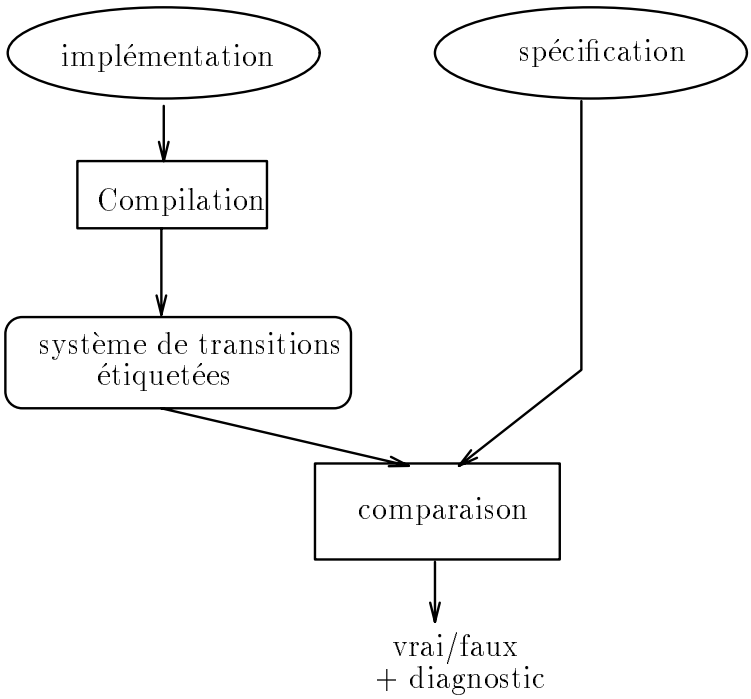


Figure 0.1: Vérification basée sur les modèles

Vérification formelle

La vérification formelle consiste à comparer la description d'un système (l'*implémentation*) avec la description des services attendus (les *spécifications*). Cette comparaison s'effectue selon une relation de satisfaction définie elle aussi formellement.

Une approche possible de vérification formelle est l'utilisation de démonstrateurs automatiques ou de systèmes de réécriture. L'utilisation de ces systèmes est en général ardue et au mieux semi-automatisable. L'approche que nous considérons dans ce travail est basée sur la comparaison de modèles (*"model checking"*), dont le principe est donné par la figure 0.1. Le modèle couramment considéré pour la vérification est un *système de transitions étiquetées*, c'est-à-dire un automate dont les transitions entre états sont étiquetées par les actions du système. Le processus de vérification se décompose alors en deux phases : compilation d'un programme exprimé dans un langage FDT vers un *système de transitions étiquetées*, puis vérification des spécifications sur ce système de transitions étiquetées. Un intérêt pratique de cette approche est qu'elle est complètement automatisable. Elle est par contre limitée par la taille des modèles générés.

La technique de vérification appliquée dépend du formalisme utilisé pour les spécifications; on distingue en particulier deux types de spécifications :

Les spécifications logiques :
Elles permettent l'expression de propriétés globales sur le fonctionnement du pro-

gramme (comme l'absence de blocages, la terminaison, l'équité, ...) à l'aide d'un ensemble de formules d'une logique temporelle. Dans ce cas, la vérification consiste à évaluer la validité de l'ensemble de ces formules sur le système de transitions étiquetées modélisant le programme.

Les spécifications comportementales :
Elles décrivent un aspect du *comportement attendu* du programme sous la forme d'un système de transitions étiquetées. Ce système de transitions étiquetées peut provenir lui-même d'une spécification écrite dans un langage FDT. La vérification consiste alors à comparer les deux systèmes de transitions par rapport à une *relation d'équivalence* ou *d'inclusion*. Ces relations sont basées sur la *bisimulation* et la *simulation* et sont modulées par une notion d'*observabitilité* des actions du programme. Le choix de la relation à utiliser dépend essentiellement du type des propriétés que l'on souhaite préserver lors de la comparaison.

L'application de ces techniques au sein du projet SPECTRE a donné lieu (entre autres) à la création d'une boîte à outils dédiée à la vérification de programmes LOTOS. Le cœur de cette boîte à outils est constitué des logiciels suivants :

CÆSAR :
Ce compilateur, réalisé par H. Garavel [Gar89] permet de générer à partir d'un programme LOTOS un système de transitions étiquetées qui modélise son comportement.

ALDÉBARAN :
ALDÉBARAN est un outil de vérification de spécifications comportementales réalisé par J.C. Fernandez [Fer88, Mou92]. Il permet de réduire ou comparer des systèmes de transitions étiquetées modulo différentes relations d'équivalence basées sur la notion de *bisimulation*.

Explosion des états

Un problème lié aux méthodes de vérification basée sur les modèles est liée à la taille de ce modèle. Dès que les programmes étudiés sont de complexité réaliste, la taille du modèle correspondant devient rapidement prohibitive; ce problème est connu sous le nom de *problème de l'explosion des états*.

Ce problème s'avère crucial dans la pratique des outils "classiques" de vérification, qui travaille sur le modèle complètement généré. Ces outils se trouvent de fait limités à la vérification de modèles ayant environ 1 million d'états, ce qui est largement insuffisant pour des cas réalistes.

Une certain nombre de méthodes ont été étudiées et mise en œuvre pour tenter de traiter différents aspects du problème de l'explosion des états. Ces méthodes jouent sur un ou plusieurs des paramètres suivants :

La génération du modèle

L'idée est alors d'essayer de générer un modèle *réduit*, mais équivalent au modèle original du point de vue vérification des propriétés à valider. D'autre part, nous pouvons aussi avoir des méthodes plus ou moins efficaces de *représentation* du modèle, en fonction de ses caractéristiques.

L'exploration du modèle

Il est possible d'améliorer considérablement les performances, en temps ou en mémoire, lors de l'exploration du modèle, en n'effectuant qu'une exploration *partielle* ou en ne mémorisant qu'une *partie* du modèle.

La qualité du résultat

Le but de la vérification formelle est de démontrer qu'une propriété est correcte ou fausse pour un système donné. Mais quand une propriété est fausse, il est souvent possible d'appliquer des méthodes plus simples qui se contenteront de la recherche d'un contre exemple, comme par exemple les différentes méthodes de *test* d'un programme. Ce sont des méthodes de vérification partielle, permettant de démontrer la *non satisfaction* d'un propriété, mais pas leur satisfaction.

Les méthodes couramment employées sont les suivantes :

Vérification à la volée [FJJM92, Val93]

Plutôt que de construire complètement le modèle, *puis* de vérifier la propriété sur ce modèle, la propriété est vérifiée *à la volée*, pendant un parcours du modèle. Pour parcourir tout le modèle, il est en général suffisant de ne garder en mémoire que la pile permettant un parcours en profondeur. Pour accélérer le calcul, la plupart des algorithmes à la volée utilisent au maximum la mémoire pour garder le plus possible des états déjà atteints.

Les mêmes algorithmes sont souvent utilisés pour une vérification *partielle* , grâce à des méthodes de codage astucieuses des ensembles d'états [Hol89] qui ne garantissent pas forcément l'exploration de tout le modèle.

Ordres partiels [GW93, Val93]

Une des causes reconnues de l'explosion des états est lié à la représentation du parallélisme par l'*entrelacement* d'actions parallèles asynchrones. Une idée consiste alors à éviter de parcourir *tous* les chemins construits par entrelacement d'actions parallèles, pour n'en parcourir qu'un sous ensemble caractéristique pour la propriété à vérifier. Le modèle ainsi parcouru peut être beaucoup plus petit que le modèle complet, surtout si le système à vérifier contient beaucoup d'actions asynchrones.

Cette technique se combine très bien avec la vérification à la volée, chaque méthode tirant parti des avantages offerts par l'autre.

Réduction

Puisque la taille du modèle pose un problème, une idée fréquemment utilisée est de réduire cette taille en tenant compte soit de la relation de comparaison utilisée, soit des

spécifications à vérifier. S'il s'agit de spécifications comportementales, la comparaison s'effectue alors pour une relation d'équivalence R. La réduction du modèle consiste alors à effectuer la vérification sur un modèle équivalent pour R, mais plus petit en nombre d'états (de préférence minimal). De nombreux travaux ont abouti à des algorithmes de réduction performants[PT87, BFH90, LY92].

Abstraction

L'abstraction consiste à construire un modèle abstrait à partir du modèle d'origine dit modèle concret, tel que chaque action du modèle concret peut être simulé par une action dans le modèle abstrait. Cette approche est maintenant utilisée dans certains travaux [Kur89, CGL92, Lon93, Loi94], pour effectuer une vérification partielle de propriétés. Les propriétés vérifiables par cette méthode doivent être préservées par la méthode d'abstraction. Dans ce cas, une propriété vérifiée sur le modèle abstrait est valide sur le modèle concret. Si la propriété n'est pas vérifiée sur le modèle abstrait, nous ne pouvons tirer aucune conclusion, car la méthode d'abstraction peut éventuellement être trop grossière pour la vérification de cette propriété.

Composition

Un système distribué étant construit par mise en parallèle de sous systèmes, des méthodes ont été proposées pour tirer parti de cette composition.

En particulier, la construction d'une abstraction du modèle peut se faire par la composition d'abstractions de chacun des composants parallèles du modèle, l'abstraction de chaque composante étant éventuellement plus facile à réaliser. De même, nous pouvons utiliser cette technique pour la réduction d'un modèle, en construisant le modèle global d'un système à partir du modèle minimal pour une relation d'équivalence R de chacune des composantes de ce système.

Une autre utilisation de la composition consiste à vérifier une propriété séparément sur chaque composant, et à en déduire la validité d'une propriété globale.

Un inconvénient de l'utilisation de la composition est lié une fois de plus au problème de l'explosion des états. Même si chaque composant ne décrit qu'une partie du comportement du système, le modèle correspondant est souvent de taille prohibitive. En fait, lorsque certains composants sont modélisés sans tenir compte de leurs interactions avec les autres (relâchement de contraintes de synchronisation) leur modèle peut même être de taille largement supérieure au modèle du système complet.

Techniques symboliques

La représentation *énumérative* d'un modèle pâtit directement du problème de l'explosion des états, la taille de la représentation mémoire étant proportionnelle à la taille du modèle lui-même. Une idée est alors de ne plus représenter chaque état individuellement, mais d'adopter des méthodes de représentation efficaces d'*ensembles* d'états, et d'adapter les opérateurs utilisés dans les algorithmes de vérification classiques pour travailler directement sur des ensembles.

Cette approche a véritablement pris son essor avec la définition par Bryant des Diagrammes de Décision Binaires (BDDs)[Bry92] , qui permettent la manipulation efficace

de fonctions booléennes. De nombreux outils de vérification utilisent les BDDs avec succès, en particulier dans le domaine de la vérification de circuits. D'autres représentations symboliques plus numériques sont aussi utilisées dans la vérification de systèmes temporisés et de systèmes hybrides [HNSY92, ACHH93, HPR94] ; il s'agit de versions plus ou moins particulières des *polyèdres convexes*, que nous présentons plus loin.

Objectifs de ce travail

Le travail que nous présentons a pour objectif l'étude et la mise en œuvre de certaines méthodes de représentation symbolique, et d'algorithmes adaptés à ces méthodes. La combinaison de ces deux approches va nous permettre de travailler sur un modèle qui se situe en amont des causes principales de l'explosion des états, puisque nous pourrons traiter les variables du programme symboliquement et non plus en extension.

Un algorithme : Génération de modèle minimal

Nous nous intéressons en particulier à l'algorithme de génération de modèle minimal proposé par [BFH90]. Cet algorithme permet à partir d'une représentation symbolique d'un modèle de construire directement un modèle réduit, suivant une relation d'équivalence donnée. Nous voulons déterminer les conditions nécessaires pour une mise en œuvre efficace de cet algorithme, en tenant compte des caractéristiques des représentations symboliques du modèle que nous comptons utiliser. Nous voulons d'autre part utiliser cet algorithme pour plusieurs relations d'équivalence.

Deux méthodes de représentation symbolique

Nous nous intéressons à deux méthodes de représentation symbolique du modèle, une méthode basé sur la représentation de fonctions booléennes, les *Diagrammes de Décision Binaires* (BDDs) et une méthode basé sur la représentation d'inégalités linéaires sur les variables rationnelles, les *polyèdres convexes*.

Les Diagrammes de Décision Binaire

Les BDDs sont maintenant utilisés avec succès dans de nombreux domaines informatiques. Certaines des réalisations les plus notables sont du domaine de la vérification de circuits et font maintenant l'objet d'une utilisation industrielle.

Beaucoup d'outils de vérification basés sur les BDDs traitent de la vérification par rapport à des spécifications logiques [BCM+90, CMB90, McM92, Lon93] . Des travaux s'intéressent aussi à leur application pour la vérification de spécifications comportementales [BdS92] . C'est le cas des travaux présentés ici [FKM93] ; nous nous intéressons à l'application des BDDs pour la génération de modèle minimal, mais aussi pour la comparaison d'un modèle

avec une spécification pour une relation d'équivalence donnée.

Les polyèdres convexes

Les polyèdres convexes peuvent être représentés par des ensembles d'inégalités et égalités linéaires, définies sur des variables rationnelles. Un intérêt de cette théorie est la possibilité de traiter *symboliquement* les variables entières d'un programme et les transformations sur ces variables. Une caractéristique supplémentaire des polyèdres est qu'ils permettent de traiter des modèles ayant un espace d'états infinis.

Les polyèdres ont déjà été utilisés pour l'*analyse sémantique* de programmes impératifs [Hal79]. Leur utilisation pour la vérification et l'analyse de systèmes distribués [BW94] , mais aussi de systèmes temporisés [Hal93, HPR94] est un phénomène récent. Nous présentons ici une adaptation de ces travaux au modèle qui nous intéresse [Ker94]. Cette adaptation nous permet de caractériser, par une *analyse en avant approchée*, une *approximation supérieure* des états accessibles d'un système. Ce sur-ensemble des états accessibles est donné sous la forme d'un ou plusieurs polyèdres et constitue un *invariant* du système. Nous pouvons en déduire des informations intéressantes sur le système modélisé, comme les bornes de variation de variables et l'existence de relations linéaires entre variables. Ceci constitue une méthode de vérification partielle, qui permettra de démontrer la satisfaction d'une formule exprimée sur les variables du système, si cette formule contient l'invariant calculé.

Un outil : MAGEL

L'ensemble des méthodes présentées dans ce document ont été mises en œuvre au sein d'un même outil, MAGEL. Cet outil permet la vérification, la comparaison et l'analyse de protocoles. Son format d'entrée a été adapté pour accepter des Réseaux de Petri générés par le compilateur CÆSAR à partir de description de protocoles en LOTOS. L'outil MAGEL permet actuellement la génération d'un modèle minimal pour différentes relations de bisimulation, directement à partir d'un Réseau de Petri restreint à des variables booléennes. Cette partie correspond à la mise en œuvre de l'algorithme de génération de modèle minimal avec les BDDs et constitue un module de vérification comportementale. MAGEL permet d'autre part l'analyse de ces Réseau de Petri étendus par des variables entières, par calcul d'approximations supérieures avec les polyèdres. Cette partie constitue un module de vérification partielle de propriétés.

L'algorithme de génération de modèle minimal avec les BDDs de MAGEL a été d'autre part intégrée avec d'autres algorithmes de vérification comportementale à l'outil de vérification ALDÉBARAN et constitue un équivalent symbolique à l'ensemble des méthodes énumératives existantes dans ALDÉBARAN.

L'ensemble des méthodes a pu ainsi être expérimenté sur une gamme d'exemples allant de jeux d'essais et de "benchmarks" à des exemples de spécification de protocoles réels.

Organisation du document

L'organisation de ce document reflète les différents thèmes que nous avons choisis. Pour chacun de ces thèmes, nous avons une partie de présentation de l'existant, et une partie d'adaptation et de mise en œuvre qui constitue l'apport de notre travail. Parmi les chapitres de description de l'existant, nous trouvons les chapitres 1, 3 et 6, ainsi que le début du chapitre 2 (description de l'algorithme de génération de modèle minimal). Les différents apports de ce travail sont décrits dans les chapitres 4, 5, 7 et 8, ainsi que la fin du chapitre 2 (optimisation et mise en œuvre de l'algorithme de génération de modèle minimal). Nous donnons une description plus précise du contenu de ces chapitres :

Le chapitre 1 définit les deux principaux modèles sur lesquels sont basés les travaux présentés dans ce document :

- le modèle système de transitions étiquetées. C'est le modèle sur lequel sont définies la plupart des méthodes de vérification que nous utiliserons
- le modèle Réseau de Petri. C'est le modèle à partir duquel nous construirons un modèle symbolique pour les BDDs, ou semi symbolique pour les polyèdres

Dans le même chapitre, nous présentons le modèle semi symbolique *automate interprété* utilisé avec les polyèdres.

Le chapitre 2 introduit une méthode de vérification de modèles pour des relations d'équivalence basées sur la notion de *bisimulation*. Cette méthode consiste à calculer une partition des états du modèle compatible avec sa relation de transition. Un algorithme efficace pour ce genre de calcul connu sous le nom de *raffinement de partition* existe [KS83, PT87] et a fait l'objet de mises en œuvre efficaces dans certains outils de minimisation et comparaison de modèles comme ALDÉBARAN, AUTO,....

Dans la suite du chapitre, nous présentons un algorithme élaboré dans l'équipe, l'algorithme de génération de modèle minimal[BFH90]. Cet algorithme permet de combiner le calcul d'accessibilité des états d'un modèle avec le calcul de raffinement de partition, permettant de minimiser un modèle *pendant* sa génération, contrairement à l'algorithme de raffinement de partition plus classique, qui est dissocié du calcul de l'accessibilité des états du modèle.

Nous présentons une mise en œuvre de cet algorithme amélioré par certaines optimisations liées à l'utilisation de représentations symboliques du modèle. Nous définissons enfin l'adaptation de l'algorithme à différentes relations de bisimulation.

Le chapitre 3 introduit la première méthode de représentation symbolique que nous allons utiliser. Il s'agit de *Diagrammes de Décision Binaires* (BDDs), qui permettent de représenter et manipuler efficacement des expressions booléennes. Nous définissons les méthodes de codage et opérateurs de base que nous utilisons par la suite.

Dans le chapitre 4 , nous définissons le codage de notre modèle Réseau de Petri sous la forme d'un ensemble de BDDs, et les avantages que peuvent apporter les BDDs pour la

représentation de modèles, du point de vue compacité de la représentation et efficacité lors des calculs de points fixes.

Le chapitre 5 contient la présentation de toutes les applications que nous avons réalisées avec le modèle symbolique à base de BDDs. En particulier, nous présentons l'application à la minimisation et comparaison par la mise en œuvre de l'algorithme de génération de modèle minimal avec les BDDs; nous appliquons aussi une méthode de comparaison pour des préordres de simulation, par exploration du produit synchrone de deux modèles à comparer.

Le chapitre 6 présente la deuxième méthode de représentation symbolique utilisée dans ce travail. Il s'agit de *polyèdres convexes*, qui peuvent être représentés par des systèmes d'inégalités linéaires sur les variables du système. Associés avec les polyèdres, nous présentons une méthode de calcul d'*approximations* de points fixes, utilisée couramment dans des systèmes d'analyse sémantique de programmes impératifs ou fonctionnels. Ces approximations vont permettre de forcer la convergence de calcul de points fixes éventuellement infinis (puisque le treillis des polyèdres est de hauteur infinie).

Le chapitre 7 présente une utilisation classique de ces polyèdres, pour une *analyse approchée en avant* d'un programme, que nous adapterons à notre modèle. Cette analyse va nous permettre de déterminer certaines propriétés invariantes sur les variables d'un programme.

Le chapitre 8 est consacrée à l'outil MAGEL, qui est le résultat de la mise en œuvre des méthodes présentées dans ce document. Nous présentons l'architecture et les choix de conception de l'outil, ainsi que son adaptation au Réseaux de Petri produits par le compilateur CÆSAR. Nous présentons ensuite l'adaptation d'une partie des algorithmes de MAGEL à ALDÉBARAN, et la création d'une version *symbolique* d'ALDÉBARAN.

Partie I

Modèles et algorithmes

Notations

0.1 Ensembles

Soient deux ensembles Q et Q', nous utiliserons pour les opérateurs ensemblistes les notations suivantes :

- ensemble vide : $\emptyset$
- cardinal : $\mid Q \mid$
- union : $Q \cup Q'$
- intersection : $Q \cap Q'$
- inclusion : $Q \subseteq Q'$
- inclusion stricte : $Q \subset Q'$
- produit cartésien : $Q \times Q'$
- différence : $Q \setminus Q'$

0.2 Relations

Soit R une relation binaire de Q sur Q ($R \subseteq Q \times Q$), R peut vérifier les propriétés suivantes :

Réfléxivité :
$\forall q \in Q, (q, q) \in R$

Symétrie :
$\forall q, q' \in Q, (q, q') \in R \equiv (q', q) \in R$

Antisymétrie :
$\forall q, q' \in Q, [(q, q') \in R \wedge (q', q) \in R] \Rightarrow q = q'$

Transitivité :
$\forall q, q', q'' \in Q, [(q, q') \in R \wedge (q', q'') \in R] \Rightarrow (q, q'') \in R$

Nous utiliserons indifféremment les notations $(q, q') \in R$ et qRq'.

Relation d'ordre et treillis

Une relation réflexive et transitive est une relation de *préordre*.

Une relation réflexive, antisymétrique et transitive est une relation d'*ordre partiel*. Si cette relation est telle que deux éléments sont toujours comparables, alors la relation est dite *totale*.

Définition 0.2-1 (Treillis complet)
Un treillis complet $(Q, \subseteq, \cup, \cap)$ *est un ensemble ordonné tel que tout partie* X *de* Q *admet une* borne supérieure *et une* borne inférieure. *En particulier, l'ensemble* Q *admet* $\top = \cup Q$ *comme élément maximal et* $\bot = \cap Q$ *comme élément minimal.* ■

Pour toute fonction totale, croissante, $\mathcal{F}$ de 2^Q vers 2^Q on appellera :

- *point fixe* de $\mathcal{F}$ l'ensemble X tel que $X = \mathcal{F}(X)$
- *post-point fixe* de $\mathcal{F}$ l'ensemble X tel que $\mathcal{F}(X) \subseteq X$
- *pre-point fixe* de $\mathcal{F}$ l'ensemble X tel que $X \subseteq \mathcal{F}(X)$

On notera

- $\mu\pi.\mathcal{F}(\pi)$ le plus petit point fixe de $\mathcal{F}$ par rapport à $\subseteq$
- $\nu\pi.\mathcal{F}(\pi)$ le plus grand point fixe de $\mathcal{F}$ par rapport à $\subseteq$

Relation d'équivalence et partition

Une relation réflexive, symétrique et transitive est une relation d'*équivalence*.

Une relation d'équivalence $\sim$ peut se définir à partir d'une relation de préordre $\subseteq$:

$$\forall q, q' \in Q \;.\; q \sim q' - (q \subseteq q') \wedge (q' \subseteq q)$$

Définition 0.2-2
On appelle *partition* d'un ensemble Q, un ensemble ρ de parties de Q tel que :

- Les éléments de ρ sont *deux à deux disjoints* :

$$\forall X, Y \in \rho \;.\; X \neq Y \Rightarrow X \cap Y = \emptyset$$

- ρ définit un *recouvrement* de Q :

$$\bigcup_{X \in \rho} X = Q$$

■

A toute relation d'équivalence $\sim$ définie sur Q, on peut associer une partition $\rho_\sim$ de Q telle que :

$$\forall p, q \in Q, p \sim q - (\exists X \in \rho_\sim p \in X \wedge q \in X)$$

Réciproquement, à toute partition définie sur Q, il est possible d'associer une relation d'équivalence $\sim_\rho$ telle que deux éléments de Q soient équivalents par $\sim_\rho$ si et seulement si ils appartiennent au même élément de ρ.

Les éléments de $\rho_\sim$ sont appelés les *classes d'équivalence* de $\sim$. Pour tout élément p de Q, nous noterons $[p]_\sim$ la classe de $\rho_\sim$ qui le contient.

0.2.1 Treillis des relations

Nous noterons $(Q \times Q, \subseteq, \cup, \cap)$ le treillis des relations d'équivalence sur $Q \times Q$. C'est un treillis complet dont l'élément maximum est la relation *universelle* $(Q \times Q)$ et l'élément minimum est l'*identité* $(\{(p,p) \mid p \in Q\})$.

On dira qu'une relation R_1 est *plus forte* qu'une relation R_2 si et seulement si R_1 *est contenue* dans R_2 $(R_1 \subseteq R_2)$. Clairement, si R_1 est plus forte que R_2, alors deux éléments de Q équivalents modulo R_1 seront équivalents modulo R_2 :

$$R_1 \subseteq R_2 - \forall (p,q) \in Q \times Q \,.\, p\ R_1\ q \Rightarrow p\ R_2\ q$$

0.2.2 Treillis des partitions

Soit $\mathcal{P} \in 2^{2^Q}$ l'ensemble des partitions sur Q, nous pouvons définir un treillis complet $(\mathcal{P}, \sqsubseteq, \sqcup, \sqcap)$ de la manière suivante :

Définition 0.2-3
Soient $\rho, \rho' \in \mathcal{P}$ des partitions définies sur Q.

- $\rho \sqsubseteq \rho'$ si et seulement si :

  $$\forall X \in \rho \,.\, \exists X' \in \rho' \,.\, X \subseteq X'$$

  On dit alors que ρ est un *raffinement* de ρ'.

- $$\rho \sqcap \rho' = \{X \cap X' \mid X \in \rho \wedge X' \in \rho'\}$$

- $$\rho \sqcup \rho' = \bigsqcap_{\rho''} \rho'' \text{ avec } \rho \sqsubseteq \rho'' \wedge \rho' \sqsubseteq \rho''$$

■

L'élément maximum de ce treillis est la *partition universelle* $\rho = \{Q\}$.

L'élément *minimum* est la *partition identité* $\rho = \{\{p\} \mid p \in Q\}$.

Si ρ est un raffinement de ρ', alors la relation d'équivalence associée à ρ est plus forte que la relation d'équivalence associée à ρ' :

$$\rho \sqsubseteq \rho' - \quad \sim_\rho \subseteq \sim'_\rho$$

Chapitre 1

Modèle

Nous présentons dans ce chapitre l'ensemble des modèles qui sont utilisés dans ce travail. Nous commençons cette présentation par le modèle de base que nous considérons pour la vérification de systèmes ; il s'agit de *systèmes de transitions étiquetées*. C'est sur ce modèle que seront définies les méthodes de vérification que nous présentons dans le chapitre suivant.

Les autres modèles que nous présentons dans ce chapitre peuvent être considérées comme des formes intermédiaires de représentation du programme. Ces formes intermédiaires sont couramment utilisées dans des compilateurs. Nous nous intéressons en particulier à une forme intermédiaire basée sur les *Réseaux de Petri* . Les Réseaux de Petri que nous présentons sont de plus *interprétés*, c'est-à-dire que ses transitions sont décorés par des tests et des actions sur les variables du programme représenté. Les méthodes symboliques que nous présentons en deuxième partie seront appliquées à partir de ce modèle.

Le dernier modèle que nous présentons est situé à mi chemin entre de Réseau de Petri et le système de transitions étiquetées. Il s'agit d'un *automate interprété*, que nous utiliserons essentiellement comme forme intermédiaire supplémentaire pour une des représentations symboliques utilisées.

1.1 Système de transitions étiquetées

Définition 1.1-1
Un système de transitions étiquetées S est un quadruplet $S = (Q, A, T, init)$, où :

- Q est l'ensemble des états de S,
- A est l'ensemble des actions de S. Nous notons A_τ l'ensemble $A \cup \{\tau\}$ où τ est l'étiquette représentant une action non observable,
- T est la relation de transition, $T \subseteq Q \times A_\tau \times Q$,
- $init$ est l'*état initial* de S.

■

Pour un système de transitions étiquetées $S = (Q, A, T, init)$ et p un élément de Q, nous utiliserons les notations suivantes :

- La transition $(p, a, q) \in T$ sera notée $p \xrightarrow{a}_T q$ (ou $p \xrightarrow{a} q$, s'il n'y a pas ambiguïté sur la relation de transition). L'état q est un *successeur* de p par l'action a. Un état sans successeur est appelé un état *puits*.

- Nous étendrons la notation précédente à une séquence d'actions $\lambda = a_1 \ldots a_n \in {A_\tau}^n$; $p \xrightarrow{\lambda}_T q$ est le prédicat défini par :

$$\lambda \ni a_1 \cdots a_n \wedge \exists q_1, \cdots, q_{n-1} \in Q \ . \ p \xrightarrow{a_1} q_1 \wedge q_1 \xrightarrow{a_2} q_2 \wedge \ \cdots \ \wedge q_{n-1} \xrightarrow{a_n} q.$$

- Nous noterons $\mathcal{A}ct(p)$ l'ensemble des actions pour lesquelles p admet un successeur par T (i.e, les actions qui peuvent être effectuées à partir de p) :

$$\mathcal{A}ct(p) = \{a \in A \mid \exists q \in Q \ . \ p \xrightarrow{a}_T q\}$$

et, pour $B \subseteq Q$,

$$\mathcal{A}ct(B) = \bigcup_{p \in B} \mathcal{A}ct(p)$$

Les fonctions de *pre-* et *post-conditions* (ou *transformateurs de prédicats*) de 2^Q vers 2^Q sont définies comme suit :

$$\begin{aligned} pre_\lambda(B) &= \{q \in Q \mid \exists p \ . \ q \xrightarrow{\lambda} p \wedge p \in B\} \\ post_\lambda(B) &= \{q \in Q \mid \exists p \ . \ p \xrightarrow{\lambda} q \wedge p \in B\} \end{aligned}$$

- Enfin, si Λ dénote un ensemble de langages λ définis sur $\mathcal{A}^*$ ($\Lambda \subseteq 2^{\mathcal{A}^*}$), l'ensemble $\mathcal{A}ct_\Lambda(p)$ des langages (ou actions abstraites) de Λ pour lesquelles p admet un successeur est défini par :

$$\mathcal{A}ct_\Lambda(p) = \{\lambda \in \Lambda \mid \exists q \in Q \ . \ p \xrightarrow{\lambda}_T q\}$$

et, pour $B \subseteq Q$,

$$\mathcal{A}ct_\Lambda(B) = \bigcup_{p \in B} \mathcal{A}ct_\Lambda(p)$$

1.1.1 Propriétés d'un système de transitions étiquetées

Définition 1.1-2
Soit $S = (Q, A, T, init)$ un système de transitions étiquetées :

- S est dit *fini* si et seulement si Q est fini

- S est dit *à branchement fini* si et seulement si chacun de ses états a un nombre fini de successeurs.

  Pour un système S à branchement fini, on appelle *facteur de branchement* c de S, le rapport entre les cardinaux de sa relation de transition et de l'ensemble de ses états :

  $$c = \frac{|T|}{|Q|}$$

- S est *déterministe* si et seulement si chaque état de Q a au plus un successeur par une action donnée :

  $$\forall p \in Q \,.\, \forall a \in \mathcal{A} \,.\, |post_a(\{p\})| \leq 1.$$

  Là encore, cette notation sera étendue aux langages d'actions de la façon suivante : si Λ est un ensemble de langages définis sur $\mathcal{A}$, S est dit *déterministe pour* Λ si et seulement si

  $$\forall p \in Q \,.\, \forall \lambda \in \Lambda \,.\, |post_\lambda(\{p\})| \leq 1.$$

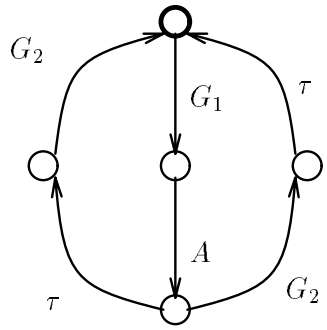
■

1.1.2 Syntaxe graphique

Nous représenterons graphiquement les systèmes de transitions étiquetées en tenant compte des règles suivantes :

- un état est représenté par un petit cercle de trait fin. L'état initial est représenté par un petit cercle de trait épais.
- une transition est représentée par une simple flèche, son étiquette lui étant accolée

Exemple 1-1
Le comportement d'un *cycler* de l'exemple du *scheduler de Milner*(voir chapitre 8) est représenté par la figure suivante :



■

1.2 Réseau de Petri Interprété

Le modèle que nous utiliserons comme base avec les méthodes de représentations symboliques présentées en deuxième partie est dérivé du modèle *Réseau de Petri Interprété* utilisé par CÆSAR dans son processus de compilation d'un programme LOTOS. Ce modèle est particulièrement adapté à la mise en œuvre des méthodes de représentation symbolique; en effet, le modèle Réseau de Petri a les caractéristiques suivantes :

- il permet la modélisation du parallélisme de manière concise
- les variables du programme et opérations sur ces variables sont représentées en *compréhension*, et pas en *extension*, ce qui se prête mieux à une représentation symbolique
- il existe des compilateurs (CÆSAR, QUASAR) qui peuvent générer un modèle dans ce formalisme à partir d'un langage de haut niveau (LOTOS, ...) .
- il n'est pas lié à un langage en particulier.

Les deux premières caractéristiques expliquent l'intérêt que nous portons à ce formalisme. En effet, les Réseaux de Petri se situent en amont du problème de l'explosion des états. Dans le cas de la boîte à outils CÆSAR-ALDÉBARAN, c'est l'interprétation de ce réseau sous la forme d'un système de transitions étiquetées qui va provoquer l'apparition d'un grand nombre d'états.

Définition 1.2-1 (Réseau de Petri Interprété)
Un réseau de Petri Interprété est défini par le tuple $(\mathcal{Q}, \mathcal{U}_r, \mathcal{T}, \mathcal{G}, \mathcal{V})$ *où :*

- $\mathcal{Q}$ *est un ensemble de places*
- $\mathcal{U}_r$ *est une unité construite sur* $\mathcal{Q}$ *et appelée* unité racine
- $\mathcal{T}$ *est un ensemble fini de transitions construites sur* $\mathcal{Q}$, $\mathcal{A}$ *et* $\mathcal{V}$
- $\mathcal{A}$ *est un ensemble de d'étiquettes*
- $\mathcal{V}$ *est un ensemble de variables*

■

1.2.1 Unités

L'ensemble des places $\mathcal{Q}$ est partitionné en *unités* :
chaque unité est un ensemble de places correspondant à un comportement s'exécutant en parallèle avec d'autres unités. Chaque unité peut elle-même être composée de sous-unités.

Plus formellement, une unité U est un triplet $(\widetilde{Q}, Q_0, \widetilde{U})$ où :

- $\widetilde{Q}$ est l'ensemble des places *propres* de U, ce sont les places appartenant à U et à aucune sous-unités de U. Nous notons $places(U)$ l'ensemble des places propres de U.
- Q_0 est la place *initiale* de U
- $\widetilde{U}$ est l'ensemble des *sous-unités* de U. On note $units(U)$ l'ensemble des sous-unités de U et $units^*(U)$ l'ensemble des sous unités transitivement incluses dans U.

L'unité racine $\mathcal{U}_r$ contient l'ensemble de toutes les unités et donc toutes les places du réseau. L'ensemble $units^*(\mathcal{U}_r)$ de toutes les unités du réseau est tel que :

- les unités sont organisées en une structure arborescente, de racine $\mathcal{U}_r$ et tel que chaque nœud U a pour fils $Units(U)$. Nous appellerons *unités de base* les unités se situant aux feuilles de cette arborescence.
- les ensembles $places(U)$ pour tout U de $Units^*(\mathcal{U}_r)$ définissent une partition de l'ensemble des places $\mathcal{Q}$.

On notera $unit(p)$ la fonction qui associe à une place p son unité.

1.2.2 Transitions

Une *transition* t est un tuple $(\widetilde{Q_i}, \widetilde{Q_o}, G, \gamma, \alpha)$ où :

- $\widetilde{Q_i} \subseteq \mathcal{Q}$ est l'ensemble des *places d'entrée*, que nous noterons $\bullet$t
- $\widetilde{Q_o} \subseteq \mathcal{Q}$ est l'ensemble des *places de sortie*, noté t$\bullet$
- $G \in \mathcal{A}$ est une *étiquette*
- γ est un ensemble de gardes définies sur l'ensemble $\mathcal{V}$
- A est un ensemble d'affectations linéaires définies sur les variables de $\mathcal{V}$

Chaque transition t est composée de deux parties :

- les transformations du *contrôle* qui sont données par les ensembles $\bullet$t et t$\bullet$
- les transformations des *données*, qui sont modélisées par A.

Le franchissement de t est d'autre part conditionné par ces gardes γ.

Avant de détailler la syntaxe des actions, nous allons d'abord restreindre le domaine des données et définir les opérations autorisées sur les données.

Valeur

Les expressions de valeur utilisées dans les actions sont restreintes aux valeurs entières ou booléennes.

$$\begin{array}{rcl} S & \equiv & Bool \\ & | & Int \end{array}$$

De même, $\mathcal{V}$ est partitionnée en deux ensembles : l'ensemble $\mathcal{V}_{Bool}$ des variables booléennes et l'ensemble $\mathcal{V}_{Int}$ des variables entières ($\mathcal{V} = \mathcal{V}_{Bool} \cup \mathcal{V}_{Int}$). La syntaxe des expressions de valeur est la suivante :

Les non terminaux V, V_{Bool} et V_{Int} dénotent respectivement une variable, une variable booléenne et une variable entière. Le non terminal n dénote une valeur entière.

$$\begin{array}{rcl} E & \equiv & E_{Bool} \\ & | & E_{Int} \end{array}$$

Expression de valeur booléenne

$$\begin{array}{rcl} E_{Bool} & \equiv & \texttt{true} \\ & | & \texttt{false} \\ & | & V_{Bool} \\ & | & \textbf{not}\ \ E_{Bool} \\ & | & E_{Bool}\ \textbf{or}\ \ E_{Bool} \\ & | & E_{Bool}\ \textbf{and}\ \ E_{Bool} \\ & | & E_{Bool} = E_{Bool} \end{array}$$

Expression de valeur entière

$$\begin{array}{rcl} E_{Int} & \equiv & n \\ & | & V_{Int} \\ & | & n.V_{Int} \\ & | & E_{Int} + E_{Int} \\ & | & E_{Int} - E_{Int} \end{array}$$

Une expression de valeur pour les entiers est une combinaison linéaire des variables de $\mathcal{V}_{Int}$.

Gardes

La syntaxe des gardes de l'ensemble γ est donnée par les règles suivantes :

$$\begin{aligned} Gardes &\equiv \textbf{when}\ Garde \\ &\mid Gardes\ ;\ \textbf{when}\ Garde \end{aligned}$$

$$\begin{aligned} Garde &\equiv E_{Bool} \\ &\mid E_{Int}\ op-comp\ E_{Int} \end{aligned}$$

$$\begin{aligned} op-comp &\equiv = \\ &\mid > \\ &\mid < \\ &\mid \leq \\ &\mid \geq \end{aligned}$$

Affectations

La syntaxe des affectations de l'ensemble A est donnée par la règle suivante :

$$\begin{aligned} Affects &\equiv V := E \\ &\mid Affects\ ;\ V := E \end{aligned}$$

1.2.3 Sémantique opérationnelle du réseau

La sémantique opérationnelle du réseau est définie par un système de transitions étiquetées. Les *états* de ce système de transitions étiquetées sont des couples $\langle M\ ,\ C\rangle$ où M est un *marquage* et C est un *contexte*. Nous définissons d'abord les notions de marquage et de contexte et les relations de transition associées, puis nous définissons la relation de transition obtenue pour les couples marquage-contexte.

Marquage

Un marquage est un ensemble de places de $\mathcal{Q}$. Le *marquage initial* est le marquage M_0 dans lequel seule la place initiale de l'unité racine $\mathcal{U}_r$ est marquée. L'ensemble des marquages *accessibles* depuis le marquage initial est noté $\mathcal{M}$.

Construire la relation de transition $\longrightarrow_m$ entre marquages revient à considérer le réseau sans tenir compte des données. Cette relation de transition est définie par la règle suivante :

$$\frac{\bullet t \subseteq M, M' = M - \bullet t \cup t\bullet}{[M,t] \longrightarrow_m M'} \quad [M1]$$

Contexte

On appelle *contexte* une application partielle de l'ensemble des variables $\mathcal{V}$ vers l'ensemble des valeurs des variables de $\mathcal{V}$. La valeur d'une variable $V \in \mathcal{V}$ dans un contexte C est donné par l'expression $C(V)$. De même, on définit une fonction d'évaluation d'*expression de valeur* :

$$eval(E,C) = \begin{cases} C(V) \text{ si } E \equiv V \\ F(eval(E_1,C)) \text{ si } E \equiv F(E_1) \\ F(eval(E_1,C), eval(E_2,C)) \text{ si } E \equiv F(E_1,E_2) \end{cases}$$

Le *contexte initial* C_0 du réseau est le contexte où aucune variable n'a été affectée. La relation de transition entre contextes est alors définie par les règles suivantes :

$$\frac{C}{[C, \textbf{none}\ ] \longrightarrow_c C} \quad [C1]$$

$$\frac{eval(E,C) = true}{[C, \textbf{when}\ E] \longrightarrow_c C} \quad [C2]$$

$$\frac{C_2 = C_{1[eval(E,C_1)/V]}}{[C_1, V := E] \longrightarrow_c C_2} \quad [C3]$$

Etats

Les états sont constitués par la combinaison d'un marquage et d'un contexte. La relation de transition entre états est alors définie à partir des relations de transition entre marquages et contextes.

Chaque transition est notée $q_1 \xrightarrow{G} q_2$ et est définie comme suit :

$$\frac{[M_1,t] \longrightarrow_m M_2 \wedge [C_1, action(t)] \longrightarrow_c C_2]}{\langle M_1\ ,\ C_1 \rangle \xrightarrow{G} \langle M_2\ ,\ C_2 \rangle} \quad [E1]$$

où G est l'étiquette de la transition $t \in \mathcal{T}$ correspondante.

1.2.4 Propriétés du réseau

Le réseau que nous considérons vérifie de plus certaines propriétés, que nous exploiterons plus loin lors de l'utilisation de représentations symboliques.

Marquage sauf
A tout moment, chaque place contient au plus *une* marque : ces réseaux sont dits *saufs*.

Séquentialité des unités
Toute unité U vérifie la propriété suivante :

$$\forall M \in \mathcal{M}, card(M \cap places(U)) \leq 1$$

Les places propres de U contiennent au plus une marque.

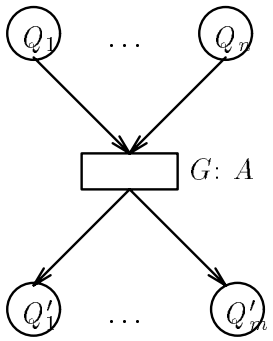
Imbrication des unités
Soient U_1 et U_2 deux unités distinctes telles que $U_1 \subset U_2$, alors pour tout marquage accessible $M \in \mathcal{M}$:

$$(M \cap places(U_1) = \emptyset) \text{ ou } (M \cap places(U_2) = \emptyset)$$

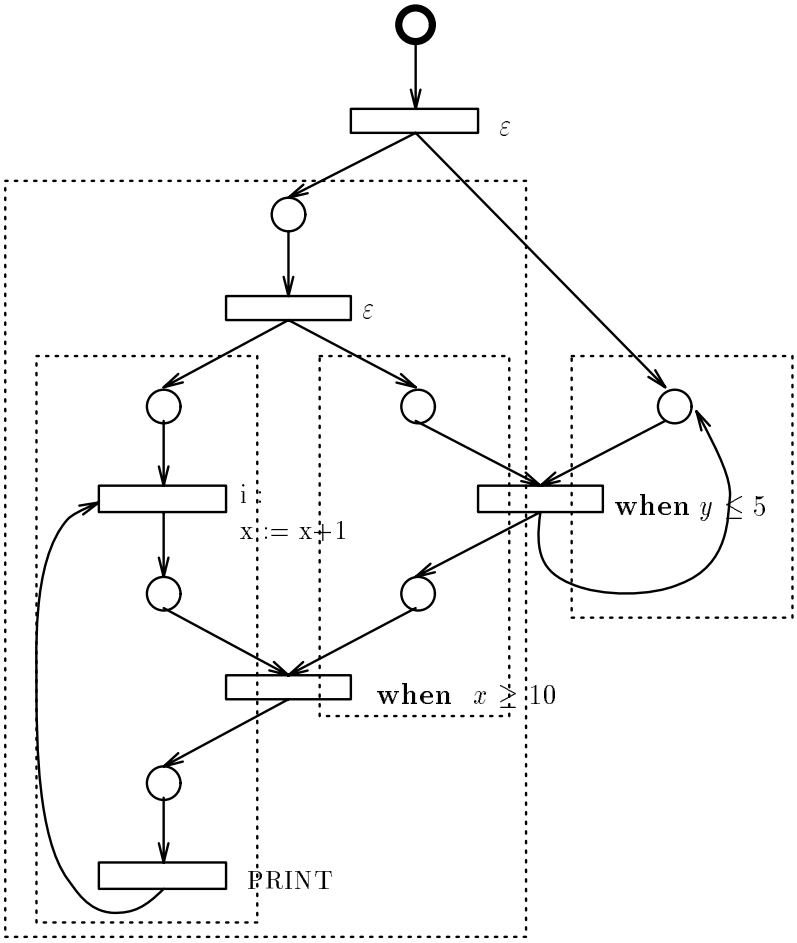
1.2.5 Syntaxe graphique

Nous représentons graphiquement les Réseaux de Petri en tenant compte des règles suivantes :

- une place est représentée par un cercle de trait fin. La place initiale est représentée par un cercle de trait épais
- une unité est représentée par un cadre en pointillé, qui contient les places et transitions de l'unité
- une transition est représentée par un rectangle décoré de plusieurs attributs. La relation entre une transition et ses places de départ et d'arrivée est représentée par une flèche. Une transition $(\widetilde{Q_i}, \widetilde{Q_o}, G, A)$ avec $\widetilde{Q_i} = \{Q_1, \ldots, Q_n\}$ et $\widetilde{Q_o} = \{Q'_1, \ldots, Q'_m\}$ est représentée par la figure :



Exemple 1-2
Un exemple de réseau structuré en cinq unités dont trois unités de base est donné par la figure suivante :



■

1.3 Automate interprété

Nous allons définir un modèle particulier, qui se situe entre le modèle réseau et le modèle système de transitions étiquetées; il s'agit d'un système de transitions où chaque transition est décorée de manière particulière, nous appelons ce système de transitions un *automate interprété*.

Nous utiliserons ce type d'automate dans un contexte particulier, qui est décrit dans le chapitre 7. La construction de cet automate se fera à partir d'un Réseau de Petri interprété ; par conséquent, nous donnons directement sa définition par rapport à un réseau.

Définition 1.3-1 (Automate interprété)
Soit un Réseau de Petri $R = (\mathcal{Q}, \mathcal{U}_l, \mathcal{T}, \mathcal{G}, \mathcal{V})$*, nous contruisons à partir de* R *un automate interprété tel que* $\mathcal{C} = (\mathcal{M}, \mathcal{T}, init)$ *tel que :*

- $\mathcal{M}$ *est un ensemble de marquages*

- $\mathcal{V}$ *est un ensemble des variables*
- $\mathcal{T}$ *est un ensemble de transitions. Chaque transition est un tuple* $(m_1, m_2, \gamma, \alpha)$ *tel que* $\exists t = (\widetilde{Q_i}, \widetilde{Q_o}, G, \gamma, \alpha) \in \mathcal{T}$ *où :*
  - $m_1, m_2 \in \mathcal{M}$ *et* $\bullet t \subseteq m_1$ *et* $m_2 = m_1 - \bullet t \cup t\bullet$ *(*$(m_1, m_2) \in \longrightarrow_m$ *où* $\longrightarrow_m$ *est la relation de transition entre les marquages du réseau R)*
- m_{init} *est le marquage initial.*

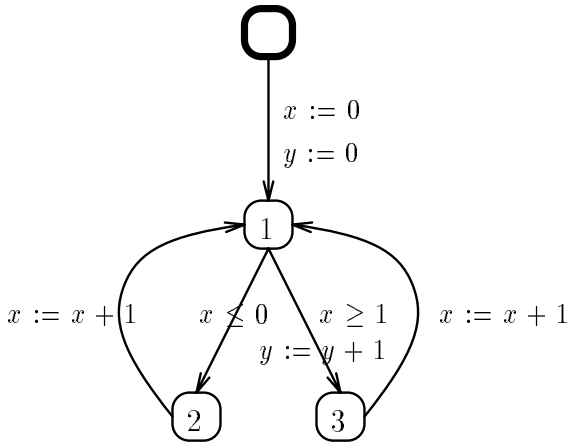
■

1.3.1 Syntaxe graphique

Nous représentons graphiquement les automates interprétés en tenant compte des règles suivantes :

- un point de contrôle est représentée par un rectangle aux coins arrondis. La place initiale est représentée par un rectangle de trait épais
- une transition est représentée par une flèche décorée par une étiquette, une liste de gardes et une liste d'affectations

Exemple 1-3



■

Chapitre 2

Vérification basée sur les modèles

Dans ce chapitre, nous nous intéressons à certaines méthodes qui permettent la vérification de systèmes parallèles par l'intermédiaire d'un modèle de ces systèmes. Nous présentons en particulier des méthodes de *comparaison* du modèle d'un système avec une spécification *comportementale*, et des méthodes de *minimisation* du modèle d'un système.

Pour pouvoir comparer ou minimiser des modèles, il faut déterminer une relation d'*équivalence* ou de *préordre* entre modèles. La définition de cette relation dépend de la nature des propriétés que l'on veut vérifier. Elle doit permettre d'identifier uniquement les modèles vérifiant les propriétés désirés, mais aussi de ne pas distinguer les modèles sur des critères autres que les propriétés voulues. La définition d'une telle relation introduit une notion d'*abstraction*, qui permet de ne prendre en compte que les comportements du système liés aux propriétés que l'on veut vérifier.

Un certain nombre de relations d'équivalences ont été proposées dans différents travaux. Nous nous intéressons en particulier aux relations entre *systèmes de transitions étiquetées*, qui constituent notre modèle de base. Ces relations sont basées sur :

- la comparaison de *séquences d'exécution*. C'est le cas de l'équivalence de trace, qui correspond à l'égalité de langages pour deux automates.
- la comparaison d'*arbres d'exécution*. Ce mode de comparaison contient le précédent et permet en plus de prendre en compte la *structure de branchement* des systèmes. C'est le cas des relations de *bisimulation* [Par81]

Les relations d'équivalence qui nous intéressent dans la suite sont de la famille des relations de bisimulation. Ces relations possèdent un certain nombre d'avantages :

- en faisant varier la notion d'observabilité des actions du programme, nous pouvons obtenir des relations de bisimulation adaptées à des classes différentes de propriétés.
- la prise en compte de la structure de branchement du programme permet la préservation de certaines classes des logiques temporelles basées sur une notion de temps *arborescent*.

- il existe un algorithme efficace pour la comparaison et la minimisation pour une relation de bisimulation particulière (la bisimulation forte) [PT87] qui a été étendu à d'autres relations de bisimulation [KS83, Fer88]

Dans la suite de ce chapitre, nous présentons une définition générale des relations de bisimulation entre systèmes de transitions étiquetées. Puis nous présentons le principe général sur lequel repose la plupart des algorithmes classiques de comparaison et de minimisation de modèles par rapport à une équivalence de bisimulation.

Nous décrivons ensuite un algorithme particulier de minimisation de modèle, basé sur le même principe, mais permettant la minimisation efficace d'un modèle *pendant* sa génération, l'algorithme de génération de modèle minimal [BFH90, BFH+92].

2.1 Relations de simulation et bisimulation

Nous présentons une définition paramétrique de la notion de bisimulation [Mou92].

Soient $S_i = (Q_i, A_i, T_i, init_i)_{(i=1,2)}$ deux systèmes de transitions étiquetées, soit Λ un ensemble de langages disjoints sur $\mathcal{A} \cup \{\tau\}$, et soit λ un élément de Λ ($\lambda \subseteq (\mathcal{A} \cup \{\tau\})^*$).

La relation de bisimulation est définie comme une famille de relations binaires sur $Q_1 \times Q_2$ paramétrée par Λ :

Définition 2.1-1 (Relation de bisimulation)
Soit l'opérateur $\mathcal{B}_\Lambda : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ *défini par :*

$$\begin{aligned}\mathcal{B}_\Lambda(R) \;=\; \{(p_1,p_2) \mid\; & \forall \lambda \in \Lambda \,.\, \forall q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \Rightarrow \exists q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \wedge (q_1,q_2) \in R)) \\ \wedge\; & \forall q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \Rightarrow \exists q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \wedge (q_1,q_2) \in R))\}\end{aligned}$$

Une relation R *sur* $Q_1 \times Q_2$ *est une bisimulation si et seulement si* $R \subseteq \mathcal{B}_\Lambda(R)$. ■

L'*équivalence de bisimulation* $\sim_\Lambda$ pour le langage Λ est alors définie comme le plus grand point fixe de l'opérateur $\mathcal{B}_\Lambda$ [Par81] :

$$\sim^\Lambda \;=\; \nu R.\, ((Q_1 \times Q_2) \;\cap\; \mathcal{B}_\Lambda(R))$$

La relation $\sim_\Lambda$ peut être définie comme l'intersection d'une suite décroissante de relations $\sim^k_{\Lambda(k \in \mathcal{N})}$ [Mil80] :

$$\begin{cases} \sim^0_\Lambda = Q_1 \times Q_2 \\ \sim^{k+1}_\Lambda = \mathcal{B}_\Lambda(\sim^k_\Lambda) \end{cases}$$

On a alors,

$$\sim_\Lambda \;=\; \bigcap_{i=0}^{\infty} (\sim^i_\Lambda)$$

La relation de simulation est obtenue à partir de la relation de bisimulation en considérant le préordre associé. Elle est définie comme une famille de relations sur $Q_1 \times Q_2$ paramétrée par Λ :

Définition 2.1-2 (Relation de simulation)
Soit l'opérateur $\mathcal{I}_\Lambda : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ *défini par :*

$$\mathcal{I}_\Lambda(R) = \{(p_1, p_2) \mid \forall \lambda \in \Lambda \,.\, \forall q_1 \,.\, (p_1 \xrightarrow{\lambda}_{T_1} q_1 \Rightarrow \exists q_2 \,.\, (p_2 \xrightarrow{\lambda}_{T_2} q_2 \wedge (q_1, q_2) \in R))\}$$

Une relation R *sur* $Q_1 \times Q_2$ *est une simulation si et seulement si* $R \subseteq \mathcal{I}_\Lambda(R)$. ■

On appelle *préordre de simulation* $\sqsubseteq_\Lambda$ pour le langage Λ le plus grand point fixe de l'opérateur $\mathcal{I}_\Lambda$:

$$\sqsubseteq_\Lambda = \nu R.\, ((Q_1 \times Q_2) \cap \mathcal{I}_\Lambda(R))$$

Comme dans le cas précédent, la relation $\sqsubseteq_\Lambda$ peut être définie comme l'intersection d'une suite décroissante de relations $\sqsubseteq^k_{\Lambda(k \in \mathcal{N})}$:

$$\begin{cases} \sqsubseteq^0_\Lambda = Q_1 \times Q_2 \\ \sqsubseteq^{k+1}_\Lambda = \mathcal{I}_\Lambda(\sqsubseteq^k_\Lambda) \end{cases}$$

On a alors,

$$\sqsubseteq_\Lambda = \bigcap_{i=0}^{\infty} (\sqsubseteq^i_\Lambda)$$

2.2 Raffinement de partition

La comparaison et la réduction de systèmes de transitions étiquetées pour une relation de bisimulation sont basées sur un même principe; il s'agit de partitionner l'ensemble des états du système par rapport à cette relation de bisimulation. Etant donné un système de transitions étiquetées $S = (Q, A, T, init)$, une partition initiale ρ_{init} de l'ensemble d'états Q et Λ un ensemble de langages disjoints sur A, le principe de l'algorithme est d'itérer le *raffinement* de ρ_{init} en partitions *compatibles* avec T, jusqu'à obtention d'une partition *stable* par rapport à elle-même. On parle alors de *raffinement de partition* par rapport à une équivalence de bisimulation.

Ce calcul est connu sous le nom de *Generalized partitionning* [KS83]. Kanellakis et Smolka ont proposé un algorithme permettant le calcul de ce raffinement de partition avec une complexité en temps de $O(m.n)$ et en mémoire de $O(m+n)$(m est le nombre de transitions et n le nombre d'états du système). Paige et Tarjan [PT87] ont ensuite proposé une solution au même problème, qu'ils appellent *Relational Coarsest Partition Problem* (RCP problem). Ils proposent une amélioration de l'algorithme, obtenant une complexité de $O(m.log(n))$ en temps, gardant la même complexité en mémoire.

Avant de présenter l'algorithme lui-même, nous allons préciser les notions de compatibilité, de stabilité et de raffinement.

Notations
Etant donné un ensemble d'états Q, nous considérerons l'ensemble P de tous les ensembles

d'ensembles non vides et disjoints deux à deux de Q :

$$P = \{\pi \in 2^Q \mid \forall X, Y \in \pi, X, Y \neq \emptyset \wedge (X \cap Y \neq \emptyset \Rightarrow X = Y)\}$$

Soit $\rho \in P, \rho$ est une *partition* ssi ρ définit un recouvrement de $Q, Q = \bigcup \{X \mid X \in \rho\}$.

Nous noterons $[q]_\rho$ la classe de la partition ρ qui contient l'état q.

A partir de la relation de transition T du modèle, nous définissons une relation de transition entre classes d'une partition ρ :

$$\xrightarrow{\lambda}_\rho = \{(X, Y) \mid X, Y \in \rho \wedge \exists \lambda \in \Lambda, \exists (q, q') \in Q, q \in X, q' \in Y, q \xrightarrow{\lambda}_T q'\}$$

Nous définissons les fonctions *pre* et *post* sur cette nouvelle relation de transition par les fonctions surchargées de 2^P dans 2^P :

$$\begin{aligned} pre^\rho_\lambda(X) &= \{[q]_\rho \mid q \in pre_\lambda(X)\} \\ pre^\rho_\lambda(\rho') &= \{pre^\rho_\lambda(X) \mid X \in \rho'\} \\ post^\rho_\lambda(X) &= \{[q]_\rho \mid q \in post_\lambda(X)\} \\ post^\rho_\lambda(\rho') &= \{post^\rho_\lambda(X) \mid X \in \rho'\} \end{aligned}$$

Accessibilité :

Une classe $X \in \rho$ est accessible si et seulement si :

- soit $X = [init]_\rho$
- soit il existe une séquence $[init]_\rho \xrightarrow{\lambda_1}_\rho X_1 \xrightarrow{\lambda_2}_\rho \ldots \xrightarrow{\lambda_n}_\rho X$

Nous noterons $Acc = \mu\pi.([init]_\rho \cup post^\rho_\Lambda(\pi))$ l'ensemble des classes accessibles de la partition ρ.

Remarque 2-1
Cette définition de l'accessibilité ne garantit pas qu'une classe $X \in Acc$ contienne un état q accessible depuis l'état initial *init*. ■

Compatibilité:

Soit un système de transitions étiquetées $S = (Q, A, T, init)$, ρ une partition et Λ un ensemble de langages disjoints sur A. On dit que la partition ρ est *compatible* avec T pour l'ensemble de langages Λ si et seulement si :

$$\forall \lambda \in \Lambda, \forall X, Y \in \rho, \forall p, q \in X, (post_\lambda(\{p\}) \cap Y \neq \emptyset) - (post_\lambda(\{q\}) \cap Y \neq \emptyset)$$

La notion de partition compatible est liée à celle de bisimulation par la proposition suivante :

Proposition 2.2-1
Soit $S = (Q, A, T, init)$, R une relation binaire définie sur $Q \times Q$ et Λ un ensemble de langages définis sur A.

R est une bisimulation pour Λ si et seulement si la partition associée à R est compatible avec Tpour Λ. ■

La preuve de cette proposition est donnée pour des systèmes non étiquetés dans [Fer88]. Elle se généralise sans difficultés au cas des systèmes étiquetés.

Raffinement :
L'opérateur de raffinement utilisé pour la génération de modèle minimal est la fonction *split* définie de façon surchargée comme suit :

- raffinement des classes d'une partition entre elles pour un langage donné :

$$\forall \lambda \in \Lambda, \forall X, Y \in 2^Q, split_\lambda(X,Y) = \{X \cap pre_\lambda(Y)\} \cup \{X \setminus pre_\lambda(Y)\}$$

  Cette version de la fonction *split* est shématisé dans la figure 2.1.

- raffinement des classes d'une partition entre elles :

$$\forall X, Y \in 2^Q, split_\Lambda(X,Y) = \prod_{\lambda \in \Lambda} split_\lambda(X,Y)$$

- raffinement d'une classe par rapport à la partition :

$$\forall X \in 2^Q, \forall \rho \in P, split_\Lambda(X,\rho) = \prod_{Y \in \rho} split_\Lambda(X,Y)$$

- raffinement de la partition par rapport à une classe :

$$\forall Y \in 2^Q, \forall \rho \in P, split_\Lambda(\rho,Y) = \bigcup_{X \in \rho} split_\Lambda(X,Y)$$

- raffinement d'une partition par rapport à une autre :

$$\forall \rho, \rho' \in P, split_\Lambda(\rho,\rho') = \bigcup_{X \in \rho} split_\Lambda(X,\rho')$$

Stabilité :

Soit $X \in \rho$, nous dirons que X est *stable* par rapport à une classe Y pour $\lambda \in \Lambda$ ssi $split_\lambda(X,Y) = \{X\}$. La stabilité de X par rapport à Y correspond au fait que X n'est pas raffinable par Y. Nous noterons cette propriété $Stable_\lambda(X,Y)$. De la même manière que pour l'opérateur *split*, nous surchargeons le prédicat *Stable* :

- $$Stable_\Lambda(X,Y) = \bigwedge_{\lambda \in \Lambda} Stable_\lambda(X,Y)$$

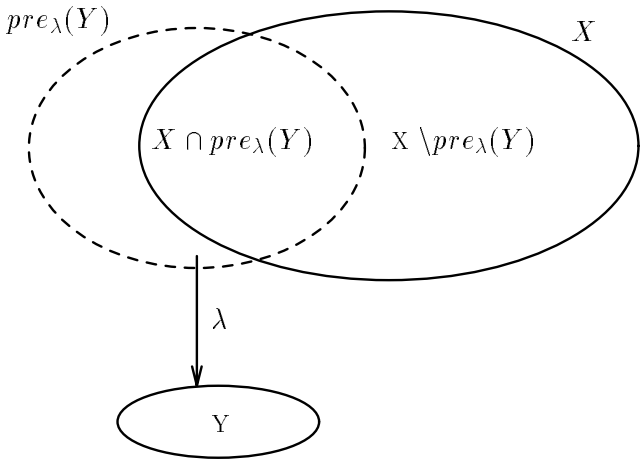


Figure 2.1: Raffinement d'une classe

- $Stable_\Lambda(X, \rho) = \bigwedge_{Y \in \rho} Stable_\Lambda(X, Y)$
- $Stable_\Lambda(\rho, Y) = \bigwedge_{X \in \rho} Stable_\Lambda(X, Y)$
- $Stable_\Lambda(\rho, \rho') = \bigwedge_{X \in \rho} Stable_\Lambda(X, \rho')$

La notion de stabilité est équivalente à la notion de compatibilité : il est facile de voir que si une partition ρ est stable par rapport à elle même ($split_\Lambda(\rho, \rho) = \rho$), alors elle est compatible avec la relation de transition T. On peut alors dire d'après la proposition 2.2-1 que si la partition ρ est stable par rapport à elle même, alors la relation d'équivalence $\sim$ qui lui est associée est une bisimulation.

Propriétés de l'opérateur $split_\Lambda$

1. $split_\Lambda(\rho, \rho) \sqsubseteq \rho$
2. $\rho_1 \sqsubseteq \rho_2 \rightarrow split_\Lambda(\rho_1, \rho) \sqsubseteq split_\Lambda(\rho_2, \rho)$
3. $\rho_1 \sqsubseteq \rho_2 \rightarrow split_\Lambda(\rho_1, \rho_1) \sqsubseteq split_\Lambda(\rho_2, \rho_2)$
4. $split_{\{\lambda_1, \lambda_2\}}(\rho_1, \rho_2) = split_{\lambda_1}(split_{\lambda_2}(\rho_1, \rho_2), \rho_2) = split_{\lambda_1}(split_{\lambda_1}(\rho_2, \rho_2), \rho_2)$
5. $\rho' = split_\Lambda(\rho, \rho) \Rightarrow \rho' = split_\Lambda(\rho', \rho)$ (ρ' est stable par rapport à ρ)
6. $\rho = split_\Lambda(\rho, X) \wedge X \in \rho' \Rightarrow split_\Lambda(\rho, \rho' - \{X\}) = split_\Lambda(\rho, \rho')$

Preuve

La preuve de ces propriétés est donnée dans [Fer90].

■

Si on considère un système de transitions étiquetées $S = (Q, A, T, init)$, et une partition initiale ρ_{init} de Q, le calcul de raffinement de partition pour une relation de bisimulation $\sim_\Lambda$ correspond alors au calcul du plus grand point fixe $\nu\rho.(\rho_{init} \sqcap split_\Lambda(\rho, \rho))$. Dans [Fer90], il est montré que le calcul de ce plus grand point fixe dans le cas des systèmes à branchement fini correspond à la limite de la suite décroissante :

$$\begin{cases} \rho_0 = \rho_{init} \\ \rho_{n+1} = split_\Lambda(\rho_n, \rho_n) \end{cases}$$

A partir de la définition de cette suite, un premier algorithme de raffinement de partition peut être déduit.

Algorithme 2-1

$\rho = \{Q\}$;
do
    $\rho' = \rho$;
    $\rho = split_\Lambda(\rho', \rho')$;
while $(\rho \neq \rho')$;

■

A l'issue de l'application de cet algorithme, la partition ρ contient les classes d'équivalence du modèle pour la relation $\sim_\Lambda$.

2.2.1 Applications "classiques" du raffinement de partition

Les applications classiques du raffinement de partition sont basées sur le prémisse suivant : l'ensemble Q des états du système est fini et la partition initiale ρ_{init} est définie sur l'ensemble des états accessibles de Q. La première ligne $\rho := \{Q\}$ de l'algorithme 2-1 est remplacée par $\rho = Acc(Q)$. Cet algorithme va nous permettre de comparer ou de minimiser des systèmes de transitions étiquetées suivant $\sim_\Lambda$. La comparaison de systèmes de transitions étiquetées va se faire par le calcul du raffinement de partition sur la réunion des deux systèmes alors que la minimisation d'un système de transitions étiquetées se fait par application directe du raffinement de partition. Ces deux points sont détaillés dans les paragraphes suivants.

Comparaison de systèmes de transitions étiquetées

Etant donnés deux systèmes de transitions étiquetées S_1 et S_2, nous voulons déterminer si ces deux systèmes ont un comportement équivalent modulo l'équivalence de bisimulation $\sim_\Lambda$. La comparaison de S_1 et S_2 est basée sur le principe suivant : nous construisons un système de transitions étiquetées S défini à partir de l'union de S_1 et S_2, puis nous calculons la partition de S associée à $\sim_\rho$. Il suffit alors de vérifier que les états initiaux de S_1 et S_2 appartiennent bien à la même classe de la partition obtenue.

Plus formellement, l'union de deux systèmes de transitions étiquetées est définie comme suit : soient $S_i = (Q_i, A_\tau, T_i, q_i)_{(i=1,2)}$ deux systèmes de transitions étiquetées, tels que $Q_1 \cap Q_2 = \emptyset$. $S = S_1 \cup S_2$ est le système de transitions étiquetées $S = (Q, A, T, init)$ tel que :

- $Q = Q_1 \cup Q_2 \cup \{init\}$
- $T = T_1 \cup T_2$
- $A = A_1 \cup A_2$

L'algorithme de raffinement de partition appliqué à ce problème devient :

Algorithme 2-2

$\rho = Q_1 \cup Q_2$;
do
    $\rho' = \rho$;
    $\rho = split_\Lambda(\rho', \rho')$;
while $(\rho \neq \rho') \wedge (\exists X \in \rho, \{q_{01}, q_{02}\} \subseteq X)$;
si $\exists X \in \rho, \{q_{01}, q_{02}\} \subseteq X$ **alors**
    afficher $(S_1 \sim_\Lambda S_2)$
sinon
    afficher $(S_1 \not\sim_\Lambda S_2)$
fsi

■

Minimisation de systèmes de transitions étiquetées
La minimisation d'un système de transitions étiquetées S par rapport à une équivalence de bisimulation $\sim_\Lambda$consiste en un calcul du *quotient* $S/\sim_\Lambda$; ce quotient doit respecter deux contraintes :

- $S/\sim_\Lambda$ est le plus petit système de transitions étiquetées en nombre d'états, tel que $S/\sim_\Lambda$ est équivalent à S modulo Λ.
- $S/\sim_\Lambda$ est une *forme normale* de S, pour tout S' équivalent à S modulo Λ, $S'/\sim_\Lambda$ est identique à $S/\sim_\Lambda$ à un renommage des états près.

La définition plus précise du quotient d'un système de transitions étiquetées est la suivante :

Définition 2.2-1
Soit $S = (Q, A, T, init)$ et $\sim_\Lambda$ une équivalence de bisimulation définie sur $Q \times Q$. Le système de transitions étiquetées $S/\sim_\Lambda$ est le *quotient* de S modulo $\sim_\Lambda$ et est défini par : $S/\sim_\Lambda = (Q_{\sim_\Lambda}, A_{\sim_\Lambda}, T_{\sim_\Lambda}, init)$

- $Q_{\sim_\Lambda} = \{[q]_{\sim_\Lambda} \mid q \in Q\}$, c'est l'ensemble des classes d'équivalence de la partition la moins fine compatible avec T pour Λ.
- $A_{\sim_\Lambda} = \{\alpha(\lambda) \mid \lambda \in \Lambda\}$, où α est une fonction *injective* de Λ dans un ensemble de représentants canoniques.
- $T_{\sim_\Lambda} = \{([q]_{\sim_\Lambda}, \alpha(\lambda), [q']_{\sim_\Lambda}) \mid q \xrightarrow{\lambda}_T q'\}$

■

Pour les équivalences de bisimulation que nous considérons, l'ensemble Λ est tel que les langages $\lambda \in \Lambda$ sont construits sur au plus une action visible :

$$\forall \lambda = a_1 \ldots a_n \in \Lambda, (\exists i \in [1..n], a_i \in A) \Rightarrow (\forall j \in [1..n], j \neq i, a_j = \tau))$$

La fonction α est alors définie telle que :

$$\alpha(\lambda) = \begin{cases} a_i \text{ si } (\exists i \in [1..n], a_i \in A) \\ \tau \text{ sinon} \end{cases}$$

La preuve de la proposition qui justifie que le système $S/\sim_\Lambda$ définie en 2.2-1 est bien le système minimal équivalent à S peut être trouvée dans [Mou92], ainsi que la méthode de construction de $T_{\sim_\Lambda}$.

2.3 Génération de Modèle Minimal

L'application classique de l'algorithme de raffinement de partition à un système de transitions étiquetées est en général basée sur une connaissance préalable de l'ensemble Acc de ses états accessibles. Le calcul de cet ensemble et de la relation de transition qui lui est associée, est en général un calcul *explicite*, par *énumération* de Acc. Les performances de ce calcul sont directement liées à la taille du modèle.

A priori, il semble peu productif d'avoir à générer le système de transitions étiquetées complet, qui peut être énorme, dans le seul but d'obtenir les informations nécessaires à sa réduction. Il serait bien plus intéressant de réduire ce système de transitions étiquetées *pendant* sa génération. Cet objectif est celui de l'algorithme de *génération de modèle minimal* [BFH90] que nous présentons plus loin.

Nous énumérons d'abord les différentes stratégies de réduction d'un modèle $S = (Q, A, T, init)$.

Stratégies de minimisation

Soit un système de transitions étiquetées $S = (Q, A, T, init)$. Nous voulons construire à partir de S un système $S_{\sim_\Lambda} = (Q_{\sim_\Lambda}, A_{\sim_\Lambda}, T_{\sim_\Lambda}, init_{\sim_\Lambda})$ qui soit le système de transitions étiquetées *minimal* par rapport à une équivalence de bisimulation $\sim_\Lambda$contenue dans une partition initiale ρ_{init}. Une approche possible est de combiner un calcul d'*accessibilité* et un calcul de *raffinement*. L'organisation de ces calculs peut suivre les stratégies définies dans les paragraphes suivants.

Accessibilité des états puis raffinement :

L'idée classique des algorithmes de réduction de modèle est d'abord de calculer l'ensemble des états accessibles du modèle, puis de raffiner la partition définie sur cet ensemble. Si on considère les fonctions suivantes :

$$\begin{aligned} F_{Acc}(X) &= init \cup post(X) \\ Refine(\rho, \pi) &= \rho_{init} \sqcap split_\Lambda(\rho \sqcap \pi, \rho \sqcap \pi) \end{aligned}$$

le calcul de l'ensemble des états accessibles de S correspond au calcul du plus petit point fixe $Acc = \mu X.F_{Acc}(X)$ Le raffinement est alors effectué sur Acc, par le calcul du plus grand point fixe $\nu\pi.Refine(\{Acc\}, \pi)$.

Le calcul de l'ensemble des états accessibles par des méthodes énumératives n'est possible que si cet ensemble est fini. Même dans le cas fini, le temps de calcul des états accessibles d'un modèle énorme par des méthodes énumératives peut être prohibitif. Néanmoins, cette approche est celle appliquée dans beaucoup d'outils comme ALDÉBARAN, AUTO, MEC, TAV, ..., soit implicitement, en considérant que le système de transitions étiquetées fourni en entrée est connexe, soit explicitement en effectuant le calcul de l'ensemble des états accessibles.

Raffinement puis accessibilité des classes :

Une autre approche est de commencer par calculer le raffinement de l'ensemble Q, sans considération d'accessibilité. Une fois que le quotient $Q_{\sim_\Lambda}$ est obtenu, l'ensemble des classes accessibles depuis la classe initiale est calculé. L'inconvénient de cette méthode est la prise en considération d'états inaccessibles dans les calculs de raffinement, et notamment le raffinement de classes inaccessibles. Le résultat de calculs sur des classes inaccessibles est rejeté lors du calcul de l'accessibilité des classes et reste donc inutile.

La partie inaccessible des états d'un modèle étant bien souvent beaucoup plus grande que la partie accessible, ces calculs inutiles peuvent peser lourdement sur les performances globales de l'algorithme. D'autre part, il est possible que le raffinement d'un modèle infini soit lui-même un modèle infini, mais avec une partie accessible finie. Dans ce cas, le processus de raffinement ne termine pas, alors qu'il est possible de calculer un modèle minimal fini.

Accessibilité *pendant* le raffinement :

Une idée naturelle est de combiner les deux calculs : calculer l'accessibilité des classes de la partition courante et ne raffiner que les classes accessibles.

Cette stratégie essaie de combiner le meilleur des stratégies précédentes. De plus, le modèle minimal obtenu est nécessairement plus petit que la partition obtenue par simple raffinement et aussi plus petit que l'ensemble des états accessibles. Ce modèle minimal peut même être fini, sans que ni l'ensemble des états accessibles, ni le raffinement de l'ensemble de tous les états le soit.

Plus formellement, on peut décrire ce calcul de la manière suivante : soit la fonction

$$F = \lambda\rho.\rho_{init} \sqcap split_{\Lambda}(\rho, \rho)$$

Cette fonction est monotone d'après la propriété 2 de l'opérateur *split*. Le calcul fait par l'algorithme 2-3 correspond alors au calcul du plus grand point fixe $\nu\rho.F(Acc(\rho))$. La démonstration de la validité de cette stratégie est donnée dans [BFH+92].

L'implémentation de cette idée correspond à l'algorithme 2-3 :

Algorithme 2-3

$\rho = \rho_{init}$;
do
    $\rho' = \rho$;
    $\pi = Acc(\rho)$;
    $\rho = split_{\Lambda}(\pi, \pi)$;
while $(\rho \neq \rho')$;

■

2.3.1 Premières optimisations

Cette première version de l'algorithme n'est évidemment pas optimale et nécessite un certain nombre d'optimisations. Ces points faibles les plus évidents sont les suivants :

- l'ensemble des classes accessibles de ρ est recalculé à chaque étape. Une possibilité est de recalculer l'accessibilité seulement sur l'ensemble des classes créées par le dernier *split*, c'est à dire les classes de $\rho \setminus \rho'$.
- il n'est pas nécessaire d'essayer de raffiner *toutes* les classes de ρ à chaque étape. En effet, d'une étape à l'autre, un certain nombre de classes de ρ resteront stables; la tentative de raffinement de ces classes est donc inutile. Une première idée permet d'isoler des classes "candidates" au raffinement : si une classe X a été raffinée à l'étape n, ces prédécesseurs ont de bonnes chances d'être effectivement raffinés à l'étape $n+1$.

- la définition de la fonction Acc ne garantit pas que si une classe appartient à $Acc(\rho)$ alors elle contient au moins un état accessible depuis l'état initial.

Tous ces points faibles ont la même origine : l'algorithme 2-3 retraite à chaque étape la partition ρ intégralement. Il est nécessaire de restreindre les traitements de chaque étape aux classes nouvellement créées et d'isoler parmi celles-ci les classes contenant un état accessible depuis l'état initial.

Pour affiner la notion d'accessibilité des classes et limiter le raffinement aux classes susceptibles d'être vraiment raffinées, nous devons descendre au niveau du raffinement d'une classe et définir le comportement de l'algorithme en fonction du résultat de ce raffinement. Considèrons une étape n de l'algorithme de raffinement de la partition ρ . L'opération $\rho' = split_\Lambda(\rho, \rho)$ revient à remplacer tout $X \in \rho$ par $split_\Lambda(X, \rho)$. Deux résultats sont possibles :

$split_\Lambda(X, \rho) = \{X\}$:

la classe X est stable par rapport à ρ. La stabilité de X implique que tous les états de X mènent exactement aux mêmes classes dans ρ :

$$\forall q_1, q_2 \in X, \forall Y \in \rho, \forall \lambda \in \Lambda, (\exists q_1' \in Y, q_1 \xrightarrow{\lambda} q_1') - (\exists q_2' \in Y, q_2 \xrightarrow{\lambda} q_2')$$

Si la classe X contient un état q accessible, alors toutes les classes accessibles depuis X contiendront au moins un état accessible depuis q. Ceci nous permet de dire que si X contient un état accessible et si $\{X\} = split_\Lambda(X, \rho)$, alors nous pouvons calculer la prochaine étape de raffinement sur toutes les classes successeurs de X, car elles sont accessibles aussi.

$split_\Lambda(X, \rho) = \{X_1, X_2, \ldots, X_k\}$:

la classe X est raffinée en un ensemble de sous classes. L'accessibilité des sous classes engendrées n'est plus connue, il faut reconsidérer les prédécesseurs de X pour recalculer cette information. Par contre, les classes prédécesseurs de X font maintenant partie des "bons" candidats au prochain raffinement, puisque leur stabilité vis-à-vis de X n'a probablement pas été conservée vis-à-vis des sous classes de X.

Pour tenir compte de ces remarques, nous garderons à jour deux ensembles pendant les différentes étapes du raffinement :

- un ensemble π des classes dont on sait qu'elles contiennent *au moins* un état *accessible*. Cet ensemble contient à l'origine uniquement la classe $[init]_\rho$. Lorsqu'à une étape donnée du processus de raffinement, une classe X de π est raffinée en plusieurs sous classes, on enlèvera X de π. Si par contre X est stable, on ajoutera les classes successeurs de X dans π.

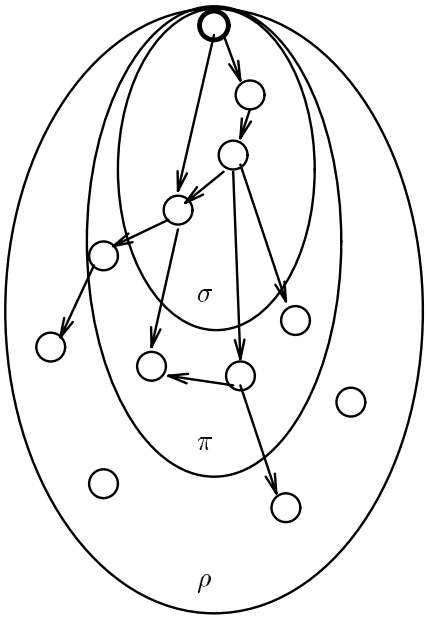


Figure 2.2: Ensembles σ et π dans ρ

- un ensemble σ des classes *stables* pour la partition ρ courante. Lors d'une étape de raffinement, une classe X trouvée stable sera incluse dans σ. Si par contre X est raffinée en plusieurs sous classes, les classes prédécesseurs de X seront ôtées de σ.

La connaissance de ces deux ensembles offre des intérêts supplémentaires : les classes à raffiner sont les classes *accessibles* et *non stables*, c'est-à-dire les classes de $\pi \setminus \sigma$. Enfin, on peut prévoir que quand toutes les classes accessibles sont stables ($\sigma = \pi$), la réduction du modèle est terminée.

Les ensembles σ et π d'une partition courante ρ sont représentés par la figure 2.2

L'introduction des optimisations liées à la connaissance de ces deux ensembles nous donne l'algorithme 2-4 , qui est l'algorithme proposé dans [BFH90] :

Algorithme 2-4

```
begin
    ρ = ρ_init
    π = {[init]_ρ}
    σ = ∅
    while π ≠ σ do
        let X in π \ σ (1)
        let π' = split_Λ(X, ρ) (2)
        if π' = {X} then
            -- la Classe X est stable --
            σ := σ ∪ {X}
            π := π ∪ {post^ρ_λ(X) | λ ∈ Λ}
        else
            π := π \ {X}
            if ∃Y ∈ π' such that init ∈ Y then
                π := π ∪ {Y}
            fi
            σ := σ \ pre^ρ_Λ(X)
            ρ := (ρ \ {X}) ∪ π'
        fi
    od
end
```

■

Maintenant que l'algorithme est déterminé, nous allons nous intéresser plus particulièrement à deux points précis :

Le choix de la prochaine classe à raffiner :
Ce choix correspond à la ligne (1) de l'algorithme. Ce choix va déterminer en grande partie le déroulement de l'algorithme.

Le raffinement d'une classe par rapport à la partition :
Ce point correspond à la ligne (2) de l'algorithme. Une définition précise et optimisée de la fonction $split_\Lambda$ est importante; cette fonction est l'élément central de l'algorithme. En particulier, nous nous intéresserons à la définition d'un sous ensemble *suffisant* de classes de la partition courante pour le raffinement d'une classe X donnée. Cet ensemble sera appelé l'ensemble des *partitionneurs effectifs* de X. Nous présenterons d'autre part quelques optimisations liées à l'enchaînement des raffinements et à l'utilisation de méthodes de représentation symbolique.

2.4 Choix du candidat au raffinement

Le bon choix de la prochaine classe à raffiner détermine en partie le bon comportement de l'algorithme. Ce choix s'effectue parmi les classes de $\pi \setminus \sigma$. Pour pouvoir définir une stratégie,

nous allons définir une relation d'ordre $<_c$ sur les classes d'équivalence telle que :

$\forall X, Y \in \rho, X <_c Y$ – X a été créée *après* Y

Comme $<_c$ est une relation d'ordre strict, nous choisirons arbitrairement un ordre parmi les éléments de ρ_{init}. De même, lors de la création "simultanée" de plusieurs classes (lors d'un raffinement), un ordre arbitraire sera choisi entre ces nouvelles classes.

Cet ordre va nous permettre de déterminer plusieurs stratégies de choix de la prochaine classe à raffiner; parmi celles-ci nous retenons les stratégies suivantes :

- choix de la classe la plus récente dans π. Ceci correspond à un parcours en profondeur d'abord par rapport à l'accessibilité des classes.

  Le choix d'une classe à raffiner X se fait alors en prenant la *plus petite* classe pour l'ordre $<_c$ qui appartiennent à $\pi \setminus \sigma$.

  Cette stratégie privilégie le raffinement des nouvelles classes par rapport au raffinement de classes étant déjà passées par un cycle raffinement-stabilisation-déstabilisation par une classe successeur.

- choix de la classe la plus ancienne. Ce choix correspond alors à un parcours en largeur d'abord par rapport à l'accessibilité des classes. Le choix de la classe à raffiner se fait alors en prenant la *plus grande* classe pour l'ordre $<_c$ qui appartiennent à $\pi \setminus \sigma$.

2.4.1 Comparaison des deux stratégies

L'intérêt d'une stratégie par rapport à l'autre dépend de l'exemple considéré. Les différences de résultat pour chaque stratégie peuvent être shématisées par le dessin 2.3.

On considère trois classes X_1, X_2 et X_3 telles que $X_1 \subseteq pre(X_2)$ et $X_1 \subseteq pre(X_3)$ (X_1 est stable par rapport à X_2 et X_3) et $X_3 <_c X_1$. $X_1 \in \sigma$ et nous supposerons que X_2 et X_3 sont dans $\pi \setminus \sigma$. Un raffinement de la classe X_2 a produit les classes X_2' et X_2'', déstabilisant la classe X_1. La stratégie de choix de la classe la plus récente donnerait préférence au raffinement de X_3 (les classes X_2' et X_2'' ne sont pas considérées, puisqu'elles ne sont pas dans π) alors que la deuxième stratégie provoquerait le raffinement de X_1 (et pourrait éventuellement propager l'instabilité jusqu'à X_0).

A priori, les classes les plus récentes sont celles dont le raffinement sera le moins productif; en effet, nous verrons plus loin (proposition 2.5-3) que ces classes sont stables (mais pas forcément accessibles), sauf si elles sont issues d'un raffinement de la forme $split_\Lambda(X, X)$. Il semble alors que la stratégie de choix de la classe la plus ancienne soit préférable.

En suivant de plus près les enchaînements de calcul de l'algorithme, il est facile de se rendre compte qu'une grande part des calculs consiste à recalculer l'accessibilité des nouvelles classes. Considérons le résultat $\{X', X''\}$ du raffinement d'une classe X. Quelque soit la stratégie utilisée il va falloir stabiliser de nouveau au moins une classe Y qui menait à X pour décider de l'accessibilité de X' et X''. Alors seulement nous pourrons continuer à raffiner les successeurs de X' et X''. Il est parfois plus intéressant de continuer le raffine-

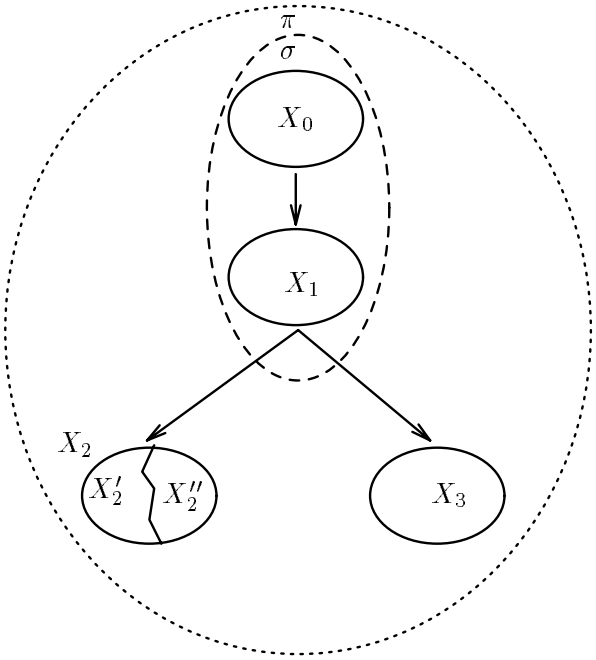


Figure 2.3: Stratégies de choix de la classe à raffiner

ment en profondeur d'abord, pour obtenir un raffinement plus fin de X avant de chercher à restabiliser les classes menant à X.

Comme la classe X est accessible depuis une classe Y, nous sommes sûr qu'au moins une classe parmi X' et X'' est accessible depuis Y. Une idée serait alors d'avoir un critère permettant de déterminer cette accessibilité, sans passer par un nouveau calcul de raffinement au niveau de Y.

Dans le cas de la classe contenant l'état initial, ce critère est facile a trouver : après chaque raffinement productif, il suffit de continuer le raffinement pour la classe générée qui contient l'état initial. Le coût de détermination de cette classe revient au coût de recherche de l'appartenance d'un état à un ensemble d'états. L'extension de ce critère serait alors d'être capable d'associer à toute classe au moins un état (s'il y en a un) accessible *depuis l'etat initial* ; nous appelerons cet état le *représentant* de la classe. C'est l'idée qui est exploitée par Lee et Yannakakis [LY92], dans un algorithme d'autre part similaire à celui présenté ici.

2.4.2 Algorithme de Lee et Yannakakis

Un algorithme basé sur le même principe que l'algorithme de génération de modèle minimal a été défini dans [LY92]. Sa principale caractéristique est l'utilisation systématique de représentants d'accessibilité des classes. En particulier, l'algorithme contient une phase de recherche en profondeur d'abord de toutes les classes accessibles par ce nouveau critère. Cette recherche en profondeur d'abord est basée sur l'étape de calcul suivante : soit une classe X

et son représentant d'accessibilité q, soit $Succ = post_\lambda(\{q\})$ l'ensemble des états successeurs de q par λ, pour toute classe Y telle que Y n'a pas de représentants et $Y \cap Succ \neq \emptyset$, on associe à Y un état (choisi arbitrairement) de $Y \cap Succ$, qui sera son représentant.

Cet algorithme nécessite d'être capable de calculer la fonction $post_\lambda$, ce qui n'est pas nécessaire dans l'algorithme de génération de modèle minimal. Cette optimisation du calcul d'accessibilité semble attractive; néanmoins, il faut considérer le coût du calcul des représentants, en particulier quand nous considérons des équivalences de bisimulations pour lesquelles le coût du calcul de $post_\lambda$ peut être assez élevé. De plus, l'algorithme présenté par [LY92] peut imposer de calculer plusieurs fois l'ensemble des successeurs d'un représentant donné (ou de conserver cet ensemble en mémoire), puisqu'il est utilisé lors de la recherche en profondeur d'abord de l'accessibilité, mais aussi lors de la phase de raffinement, pour le calcul de représentant des nouvelles classes.

Par conséquent, le calcul des représentants peut revenir *plus cher* que calculer l'ensemble des états accessibles. Un exemple est un modèle déjà minimal, sur lequel on applique l'algorithme.

Toutes ces considérations nous amènent au point suivant : le calcul de l'accessibilité des classes à l'aide de représentants revient souvent à un calcul d'accessibilité des états de tout le modèle, *en profondeur d'abord*. Or, les méthodes de représentation symbolique sont plutôt adaptées à des calculs *en largeur d'abord*. Par conséquent, plutôt que de mettre en œuvre un mécanisme de calcul systématique de représentants d'accessibilité, il parait plus intéressant de calculer à priori l'ensemble des états accessibles. Le calcul des représentants d'accessibilité offrent surtout un intérêt dans le cas d'un modèle infini, dont le modèle minimal est fini.

2.5 Définition de la fonction $split_\Lambda$

Nous allons maintenant définir plus précisément la fonction $split_\Lambda$. La version courante de l'algorithme 2-4 indique que le raffinement d'une classe se fait par rapport à l'ensemble de la partition. Néanmoins, en fonction des raffinements précédents, il est possible de déterminer pour une classe X un ensemble *suffisant* de *partitionneurs* qui soit inclus dans la partition courante. D'autre part, nous verrons que le raffinement d'une classe X par rapport à une classe Y qui elle-même vient d'être raffinée peut donner lieu à certaines optimisations.

Pour pouvoir déterminer cet ensemble de partitionneurs, nous allons construire et mettre à jour des structures de données permettant de conserver des informations sur le raffinement en cours; ce seront respectivement un *arbre de raffinement* et une *relation de transition* définie sur la partition courante.

2.5.1 Ensemble de partitionneurs

Le calcul d'un ensemble suffisant de partitionneurs est donné dans les deux propositions suivantes :

Définition 2.5-1 (Ensemble de partitionneurs *effectifs*)
Soit une classe X, l'ensemble des classes à considérer pour le raffinement de X par rapport à $\lambda \in \Lambda$ est l'ensemble $Eff_\lambda(X) = \{Y \in \rho \mid \exists \lambda \in \Lambda, X \cap pre_\lambda(Y) \neq \emptyset \wedge X \not\subseteq pre_\lambda(Y)\}$ ■

Nous noterons $Eff_\Lambda(X) = \bigcup_{\lambda \in \Lambda} Eff_\lambda(X)$.

Proposition 2.5-1
Soit ρ une partition et $X \in \rho$, nous avons alors la propriété suivante :

$$split_\Lambda(X, \rho) = split_\Lambda(X, Eff_\Lambda(X))$$

■

Preuve D'après la propriété 6 de l'opérateur $split_\Lambda$,

$$\forall\, Y \notin Eff_\lambda(X), split_\Lambda(X, Y) = \{X\} - \; split_\Lambda(X, \rho) = split_\Lambda(X, \rho \setminus \{Y\})$$

■

Le lien entre l'ensemble Eff_Λ et $Stable_\Lambda$ est facile à trouver :

$$Eff_\Lambda(X) = \{Y \in \rho \mid \neg Stable_\Lambda(X, Y)\}$$

Le nombre de raffinements à effectuer peut encore être diminué, si on sépare les raffinements d'une classe X en fonction du langage $\lambda \in \Lambda$ pour lequel on applique le raffinement.

On obtient en définitive la proposition suivante :

Proposition 2.5-2
Soit ρ une partition, $X \in \rho$ et $\lambda \in \Lambda$, on a la propriété suivante :

$$split_\Lambda(X, \rho) = \prod_{\lambda \in \Lambda} split_\lambda(X, Eff_\lambda(X))$$

■

Cette dernière proposition va nous permettre d'écrire une version de la fonction $split_\Lambda$ qui effectue le raffinement seulement en fonction des classes pour lesquelles le raffinement est effectif.

Algorithme 2-5

```
function split_Λ(X, ρ)
        var  P, Z : class
        Ñ : set of class
    Ñ := {X}
    for λ ∈ Λ do (1)
        for  Y ∈ Eff_λ(X) do
            for  Z ∈ Ñ do
                Ñ := Ñ − {Z} ∪ split_λ(Z, Y)
    split_Λ := Ñ
    end
```

■

Nous avons proposé un ensemble suffisant de partitionneurs. Nous devons maintenant mon-

trer comment obtenir cet ensemble d'un raffinement à l'autre. Il n'est en effet pas question d'"essayer" chaque classe de ρ pour voir si c'est un partitionneur effectif d'une classe X. Nous allons donc conserver des informations d'un raffinement à l'autre, et tirer parti des propositions suivantes :

Proposition 2.5-3 (Stabilité de classes raffinées)

$$\forall X, Y \in \rho, \forall \lambda \in \Lambda$$
$$(X \neq Y \wedge \{X_1, X_2\} = split_\lambda(X, Y)) \Rightarrow Stable_\lambda(X_1, Y) \wedge Stable_\lambda(X_2, Y)$$

■

Preuve

Le résultat de l'opérateur $split_\lambda(X, Y)$ est un couple $\{X \cap pre_\lambda(Y), X \setminus pre_\lambda(Y)\}$.
Supposons que $X_1 = X \cap pre_\lambda(Y)$ et $X_2 = X \setminus pre_\lambda(Y)$.
Alors $X_1 \subseteq pre_\lambda(Y)$, donc $Stable_\lambda(X_1, Y)$ et $X_2 \cap pre_\lambda(Y) = \emptyset$
donc nous avons $Stable_\lambda(X_2, Y)$.

■

Proposition 2.5-4 (Héritage de la stabilité par raffinement)

$$\forall X, Y \in \rho, \forall \lambda \in \Lambda$$
$$Stable_\lambda(X, Y) \Rightarrow \forall X_i \in split_\Lambda(X, \rho), Stable_\lambda(X_i, Y)$$

■

Preuve

$$\begin{array}{lll} \forall \lambda \in \Lambda, Stable_\lambda(X, Y) & - & split_\lambda(X, Y) = \{X\} \\ & - & X \subseteq pre_\lambda(Y) \vee (X \cap pre_\lambda(Y) = \emptyset) \\ & \Rightarrow & \forall X_i \subseteq X, X_i \subseteq pre_\lambda(Y) \vee (X_i \cap pre_\lambda(Y) = \emptyset) \\ & - & split_\lambda(X_i, Y) = \{X_i\} \\ & - & Stable_\lambda(X_i, Y) \end{array}$$

■

En particulier, si nous avons $Stable_\lambda(X, Y)$ parce que $pre_\lambda(Y) \cap X = \emptyset$, il n'est plus nécessaire à l'avenir de considérer Y ou des sous classes de Y pour le raffinement de X.

Nous pouvons essayer de faire encore mieux, en utilisant l'idée de Paige et Tarjan.

2.5.2 Optimisation de Paige et Tarjan

L'idée originale de Paige et Tarjan a pour but une diminution du nombre de partitionneurs utilisés pour le raffinement d'une classe. Elle est basée sur la proposition suivante :

Proposition 2.5-5

Soit ρ une partition et deux classes X et $Y \in \rho$ telles que $X \subseteq pre_\lambda(Y)$ (donc $Stable_\lambda(X,Y)$) pour $\lambda \in \Lambda$. Soit l'ensemble $\{Y_1, Y_2\} \subseteq \rho$ obtenu par raffinement de $Y = Y_1 \cup Y_2$. On a alors :

$$split_\lambda(X, Y_1) \sqcap split_\lambda(X, Y_2) = \{X_1, X_2, X_3\}$$

avec

$$\begin{aligned} X_1 &= (X \cap pre_\lambda(Y_1)) - pre_\lambda(Y_2) = X - pre_\lambda(Y_2) \\ X_2 &= (X \cap pre_\lambda(Y_2)) - pre_\lambda(Y_1) = X - pre_\lambda(Y_1) \\ X_3 &= (X \cap pre_\lambda(Y_1)) \cap pre_\lambda(Y_2) \\ et & \\ X_4 &= X - pre_\lambda(Y_1) - pre_\lambda(Y_2) = \emptyset \end{aligned}$$

■

Preuve

On a $pre_\lambda(Y) = pre_\lambda(Y_1 \cup Y_2) = pre_\lambda(Y_1) \cup pre_\lambda(Y_2)$

donc $X \subseteq pre_\lambda(Y) - X \subseteq pre_\lambda(Y_1) \cup pre_\lambda(Y_2)$.

donc $X - pre_\lambda(Y_1) - pre_\lambda(Y_2) = \emptyset$.

■

De cette proposition nous déduisons la proposition suivante :

Proposition 2.5-6
Soient ρ une partition et deux classes X et $Y \in \rho$ telles que X soit stable par rapport à Y pour $\lambda \in \Lambda$ et $Y = Y_1 \cup Y_2$. On a alors :

$$split_\lambda(X, \{Y_1, Y_2\}) = \{X_1, X_2, X_3\}$$

où X_1, X_2 et X_3 sont calculés en deux étapes :

1.
$$\begin{aligned} X' &= X \cap pre_\lambda(Y_2) \\ X_1 &= X - pre_\lambda(Y_2) \end{aligned}$$

2.
$$\begin{aligned} X_2 &= X' \cap pre_\lambda(Y_1) \\ X_3 &= X' - pre_\lambda(Y_1) \end{aligned}$$

■

Le premier résultat est qu'il est inutile de chercher à raffiner la classe X_1 par rapport à Y_1, faisant ainsi l'économie d'un calcul de l'opération $-$.

Paige et Tarjan utilisent la proposition 2.5-5 pour construire une fonction de raffinement *split* qui ne considère que la *plus petite* classe (en nombre d'états) parmi Y_1 et Y_2 pour

raffiner X. Leur algorithme nécessite d'être capable d'associer à chaque état de l'ensemble Q le nombre de ses successeurs dans une classe $X \in \rho$, c'est-à-dire de disposer de la fonction $nb\_succ_\lambda : Q \times \rho \rightarrow I\!N$ qui est telle que :

$$\forall q \in Q, \forall X \in \rho, nb\_succ_\lambda(q, X) = \mid post_\lambda(q) \cap X \mid$$

Le calcul du raffinement de X en $\{X_1, X_2, X_3\}$ se fait alors suivant les règles :

$$\frac{nb\_succ_\lambda(p, Y_1) = nb\_succ_\lambda(p, Y)}{X_1 = X_1 \cup \{p\}} \quad [R1]$$

$$\frac{nb\_succ_\lambda(p, Y_1) = 0}{X_2 = X_2 \cup \{p\}} \quad [R2]$$

$$\frac{0 < nb\_succ_\lambda(p, Y_1) < nb\_succ_\lambda(p, Y)}{X_3 = X_3 \cup \{p\}} \quad [R3]$$

Ces règles dépendent uniquement des valeurs de la fonction nb_succ par rapport aux classes Y et Y_1. Il suffit donc de choisir Y_1 comme étant la plus "petite" des deux classes Y_1 et Y_2.

Dans le cas de représentations symboliques, la fonction nb_succ peut être coûteuse. D'autre part, l'algorithme de Paige et Tarjan est habituellement utilisé pour le calcul de raffinement de partition par rapport à la bisimulation forte; la fonction nb_succ est très coûteuse à calculer, en particulier pour des relations comme la bisimulation $\tau^* a$.

Par conséquent, nous n'utiliserons pas cette optimisation pour notre fonction de raffinement, puisque nous voulons travailler avec des représentations symboliques.

Nous avons défini quelques propositions permettant de construire d'un raffinement à l'autre un ensemble suffisant de partitionneurs. Nous allons maintenant décrire les structures de données qui vont nous permettre de garder et mettre à jour ces informations.

2.5.3 Structure de données

Pour pouvoir calculer à chaque étape l'ensemble minimal de partitionneurs nécessaires, tel qu'il est défini par la proposition 2.5-2, il faut conserver des informations supplémentaires d'un raffinement à l'autre.

Pour cela, nous allons construire et mettre à jour tout au long du processus de raffinement une relation entre chaque classe, qui s'apparente à une *relation de transition*. De plus, nous garderons un *arbre de raffinement* qui conservera des informations sur les raffinements successifs des classes de ρ.

Arbre de raffinement $\mathcal{D}$

Cet arbre va permettre de conserver des informations sur les raffinements déjà effectués. Nous représentons cette arbre sous la forme d'un arbre *binaire* défini comme suit :

Définition 2.5-2 (Arbre de raffinement)
Un arbre de raffinement $\mathcal{D}$ est un arbre binaire tel que :

- *chaque nœud de $\mathcal{D}$ correspond à une classe $X \in 2^Q$. Nous noterons X ce nœud.*
- *un nœud X de fils X_1 et X_2 est tel que $\exists Y \in 2^Q, \exists \lambda \in \Lambda, \{X_1, X_2\} = split_\lambda(X, Y)$*
- *l'ensemble Root des nœuds racines de $\mathcal{D}$ est égal à ρ_{init}.*
- *l'ensemble des feuilles de $\mathcal{D}$ correspond à la partition courante ρ.*

■

Nous noterons $\mathcal{N}$ l'ensemble des nœuds de $\mathcal{D}$, que nous appellerons *blocs*. Parmi les éléments de $\mathcal{N}$ nous distinguons les *blocs simples* de $\mathcal{N}$, qui sont les feuilles de $\mathcal{D}$ et dont l'ensemble forme la partition courante ρ et les *blocs arbres* qui sont les blocs autres que les feuilles de $\mathcal{D}$. D'autre part, nous utiliserons la fonction $Fils : \mathcal{N} \rightarrow \mathcal{N} \times \mathcal{N}$ qui pour un nœud de $\mathcal{D}$ donne ces deux fils. Nous noterons $Fils^* : \mathcal{N} \rightarrow \rho$ la fonction qui pour un bloc arbre donné rend l'ensemble de ses feuilles.

Système de transition $\mathcal{R}af$

Dans l'algorithme 2-4, il est nécessaire de construire et mettre à jour la relation de transition induite par le processus de raffinement et le calcul d'accessibilité. Cette relation de transition permet de conserver les informations utiles pour le calcul des ensembles de partitionneurs, ainsi que des fonctions pre^ρ_λ et $post^\rho_\lambda$ nécessaires pour l'algorithme. De plus, nous verrons que cette relation de transition coïncide avec celle du modèle minimal, lorsque l'algorithme termine.

On considère à chaque étape le système de transition suivant, construit sur $\mathcal{N}$:

$$\mathcal{R}af = (\mathcal{N}, \Lambda, \mathcal{T}_{\mathcal{R}af}, Init_{\mathcal{R}af})$$

- $\mathcal{N}$ est l'ensemble des nœuds de $\mathcal{D}$
- Λ est l'ensemble de langages utilisé pour ce raffinement
- $\mathcal{T}_{\mathcal{R}af}$ est une relation définie sur $\mathcal{N} \times \Lambda \times \mathcal{N} \times \{0, 1\}$. Chaque transition a pour source une classe de la partition courante et pour destination soit un bloc simple, soit un bloc arbre de la partition courante.
- $Init_{\mathcal{R}af}$ est la classe de l'état initial du modèle.

Remarque 2-2
La valeur booléenne associée à chaque transition est un marquage qui indique si une transition est *stable*. On a l'implication suivante :
$\forall \lambda \in \Lambda, (X, \lambda, Y, 1) \in \mathcal{T}_{\mathcal{R}af} \Rightarrow Stable_\lambda(X, Y)$ ■

2.5.4 Initialisation de $\mathcal{R}af$ et $\mathcal{D}$

L'arbre de raffinement $\mathcal{D}$ est construit avec l'ensemble de ces nœuds réduit à $Root = \rho_{init}$.

La relation de transition de $\mathcal{R}af$ est initialisée comme suit :

- $\mathcal{N} = \rho_{init}$
- $\mathcal{T}_{\mathcal{R}af} = \{(X, \lambda, Y, 0) \mid \lambda \in \Lambda, X, Y \in \rho_{init}\}$

2.5.5 Mise à jour de $\mathcal{R}af$ et $\mathcal{D}$

La mise à jour de $\mathcal{R}af$ et $\mathcal{D}$ se fait lors de chaque $split_\lambda$, et en particulier lors de chaque raffinement par rapport à une transition non stable. Les méthodes de mise à jour de ces structures seront détaillées en section 2.5.6 .

Ensemble de partitionneurs

A partir de la définition de la relation de transition $\mathcal{R}af$, nous pouvons définir l'ensemble des partitionneurs d'une classe X :

Définition 2.5-3 (Ensemble de partitionneurs)
Nous appellerons ensemble de partitionneurs *de X et noterons $Part_\Lambda(X)$ l'ensemble :*

$$Part_\Lambda(X) = \bigcup_{\lambda \in \Lambda} Part_\lambda(X)$$

avec

$$Part_\lambda(X) = \{(X, \lambda, Y, S) \mid (X, \lambda, Y, S) \in \mathcal{T}_{\mathcal{R}af}\}$$

■

Les éléments de $Part_\Lambda(X)$ peuvent être de 2 natures différentes : Soit $(X, \lambda, Y, S) \in Part_\Lambda(X)$,

Y est un bloc simple et $S = 1$
Dans ce cas, X est stable par rapport à Y pour λ, il n'est pas nécessaire de raffiner X pour cette transition.

Y est un bloc arbre ou $S = 0$
Y a été raffinée, ou la stabilité de X par rapport à Y n'est pas connue. Dans ce cas, il faut raffiner X par rapport à Y.

Nous allons maintenant nos intéresser plus particulièrement au raffinement par rapport à un bloc arbre.

2.5.6 Raffinement par rapport à un bloc arbre

Soit X une classe de ρ et Y une classe correspond à un bloc arbre de $\mathcal{D}$. Le raffinement de X par rapport à Y est alors défini par :

Définition 2.5-4 (Raffinement par rapport à un bloc arbre)

$$split_\Lambda(X, Y) = \prod_{B \in Fils^*(Y)} split_\Lambda(X, B)$$

■

Dans le cas de l'utilisation de techniques symboliques de représentation du modèle, le coût de calcul de la fonction pre_λ est du même ordre pour toute classe X. En particulier, ce coût ne dépend pas du cardinal de l'ensemble représenté. Il peut donc être intéressant de calculer le raffinement de X par rapport à des nœuds intermédiaires (i.e. autre que des feuilles) du bloc arbre Y. En effet, considérons un nœud y du bloc arbre Y, tel que $Fils(y) \neq \emptyset$. Si la tentative de raffinement de X par rapport à y nous montre que $X \cap pre_\lambda(y) = \emptyset$, alors il est inutile de chercher à raffiner X par rapport aux éléments de $Fils^*(y)$. On évite alors autant de calculs de pre_λ qu'il y a d'éléments dans $Fils^*(y)$.

Remarque 2-3
La considération précédente est liée à une notion d'accessibilité locale (est ce que y contient au moins un état accessible depuis la classe X?). Nous discutons plus loin des considérations sur l'accessibilité globale d'une classe (contient-elle au moins un état accessible depuis l'état initial?). En particulier, les avantages et inconvénients des deux techniques suivantes sont discutées en dehors de considérations d'accessibilité globale. ■

Nous présentons deux techniques différentes du calcul du raffinement de X par rapport au bloc arbre Y. La première va effectuer le calcul uniquement à partir des *feuilles* du bloc arbre Y. La deuxième va parcourir le bloc arbre Y et effectuer un raffinement pour chaque nœud.

Remarque 2-4
Suivant le déroulement de l'algorithme de raffinement de partition, il est possible que $X \subseteq Y$. En particulier, si Y est un bloc arbre, il est possible que $X \in Fils^*(Y)$. Si l'arbre de raffinement $\mathcal{D}$ qui permet le calcul de la fonction $Fils$ est mise à jour au fur et à mesure du parcours du bloc arbre Y, alors nous risquons d'effectuer le raffinement de X par rapport au résultat courant du raffinement de X. Dans ce cas, la propriété d'héritage de la stabilité par raffinement n'est plus vérifiée.

Pour éviter ces problèmes, nous ajoutons en paramètre de la fonction $split_\lambda$ la classe X originale, pour permettre sa détection dans le bloc arbre Y. Si une feuille de Y est égale à X, nous effectuons un raffinement par rapport à X, mais pas par rapport à $Fils^*(X)$. ■

Raffinement pour chaque feuille
L'algorithme suivant est dérivé de la définition 2.5-4. Nous effectuons directement le calcul de $split_\lambda(X, B)$ pour chaque $B \in Fils^*(Y)$.

function *$split_\lambda$ (X : class, Y : bloc) : set of class*

$\widetilde{Y} := Sons^*(Y)$
-- l'ensemble des feuilles de Y est déterminé une fois pour toute
-- il n'est pas nécessaire (dans cette version) de tester si une feuille est égale à X
$\widetilde{X} := \{X\}$

for $Z \in \widetilde{Y}$ **do**
  for $X \in \widetilde{X}$ **do**
    $\widetilde{X} = \widetilde{X} \cup \{X \cap pre_\lambda(Y)\} \cup \{X \setminus pre_\lambda(Y)\}$ *(1)*
return $\widetilde{X}$

Raffinement pour chaque nœud

Le raffinement par nœud est calculé par l'appel $split_\lambda(X, Y)$ de la fonction suivante :

function *$split_\lambda$ (X : class, Y : bloc) :* **set of** *class*
  return *$split_\lambda$ -rec(X,{X}, Y)*

function *$split_\lambda$ -rec (X : class$\widetilde{X}$: set of class, Y : bloc) :* **set of** *class*
  pY : class
  $\widetilde{X_1}, \widetilde{X_2}$ *:* **set of** *class*

  $\widetilde{X_1} := \widetilde{X_2} := \emptyset$
  -- $\widetilde{X_1}$ est l'ensemble des classes ayant un successeur dans Y --
  -- $\widetilde{X_2}$ est l'ensemble de celles n'en ayant pas --
  $pY := pre_\lambda(Y)$
  for $X \in \widetilde{X}$ **do**
    $\widetilde{X_1} := \widetilde{X_1} \cup (X \cap pY)$
    $\widetilde{X_2} := \widetilde{X_2} \cup (X \setminus pY)$

  -- Seul les éléments de X_1 sont à raffiner pour les descendants de Y
  if $(\widetilde{X_1} \neq \emptyset$ **and** $Fils(Y) \neq \emptyset$ **and** $X \neq Y$ $)$ **then**
    $\{Y_1, Y_2\} = Fils(Y)$
    $\widetilde{X_1} := split_\lambda - rec(\widetilde{X_1}, Y_1)$
    $\widetilde{X_1} := split_\lambda - rec(\widetilde{X_1}, Y_2)$

  return $\widetilde{X_1} \cup \widetilde{X_2}$

Remarque 2-5
Dans les deux cas, le parcours de l'arbre est nécessaire. Ce parcours peut être partiel dans la solution par nœuds. Dans la solution par feuilles, ce parcours est complet, puisqu'il faut déterminer l'ensemble des feuilles. ■

Comparaison des deux solutions

La solution par feuilles provoque autant de calculs de raffinement que de feuilles.

La solution par nœuds peut nous permettre d'éviter des parties entières de l'arbre et donc de diminuer le nombre de raffinements à effectuer. Mais dans le pire des cas, qui correspond au cas où toutes les feuilles de Y permettent un raffinement productif, nous effectuons alors autant de raffinements que de nœuds dans l'arbre.

Si nous considérons un arbre binaire complet de profondeur l, le nombre de raffinement pour la solution par feuilles sera alors égal à 2^{l-1}. Dans le cas de la solution par nœud, le nombre de calcul de raffinement est compris entre $2(l-1)$ et $2^l - 2$. La borne inférieure $2(l-1)$ correspond au cas où une seule feuille de l'arbre produit un raffinement effectif. La borne supérieure $2^l - 2$ correspond au nombre maximum de nœuds dans le bloc arbre, sans compter la racine. Dans le pire des cas, la solution par nœuds peut provoquer environ 2 fois plus de calculs de raffinements que la solution par feuilles.

Solution retenue

Une alternative est de moduler la solution par nœuds en choisissant une valeur r telle que pour tout nœud intermédiaire (qui n'est pas une feuille) le calcul de raffinement n'est effectué que si sa profondeur l' est telle que $l' \equiv 0 \bmod r$ (la profondeur l' est un multiple de r). Le raffinement pour chaque feuille continue à se faire normalement. Cette solution permet de conserver la possibilité d'éviter le parcours d'une partie du bloc arbre, mais permet aussi d'éviter le calcul d'un raffinement pour chaque nœud. Nous retrouvons les deux solutions précédentes, en prenant $r = 1$ pour la solution par nœuds et $r > l$ pour la solution par feuilles. Le choix de la valeur de r peut se faire en fonction de la profondeur de l'arbre l (si cette valeur est connue à priori) ou arbitrairement.

Cette solution est à la base de l'algorithme suivant, qui est complété avec la mise à jour de $\mathcal{D}$ et de $\mathcal{R}af$.

Soit $r \geq 1$ le nombre de niveaux déterminant un raffinement obligatoire. L'algorithme de $split_\lambda$ est alors :

function $split_\lambda$ (X_0 : *class*,$\widetilde{X}$: *set of class*, Y : *bloc*, P : *entier*) : **set of** *class*

if ($P \equiv 0\ mod r$) *ou* $Fils(Y) = \emptyset$ **then**
$\widetilde{X_1} := \widetilde{X_2} := \emptyset$
-- $\widetilde{X_1}$ *est l'ensemble des classes ayant un successeur dans* Y --
-- $\widetilde{X_2}$ *est l'ensemble de celles n'en ayant pas* --
$pY := pre_\lambda(Y)$
for $X \in \widetilde{X}$ **do**
$\widetilde{X_1} := \widetilde{X_1} \cup (X \cap pY)$
$\widetilde{X_2} := \widetilde{X_2} \cup (X \setminus pY)$
if $X_1 \neq \emptyset$ **then**
$\mathcal{T}_{\mathcal{R}af} := \mathcal{T}_{\mathcal{R}af} \cup \{(X, \lambda, Y, 1)\}$
if $X_2 \neq \emptyset$ **then**
$Fils(X) = \{X_1, X_2\}$
fi
fi
endfor
else
$\widetilde{X_1} := \widetilde{X}$
fi

-- *Seul les éléments de* X_1 *sont à raffiner pour les descendants de* Y
if ($\widetilde{X_1} \neq \emptyset$ **and** $Fils(Y) \neq \emptyset$ **and** $(Y \neq X_0)$) **then**
$\{Y_1, Y_2\} = Fils(Y)$
$\widetilde{X_1} := split_\lambda(X_0, \widetilde{X_1}, Y_1, P+1)$
$\widetilde{X_1} := split_\lambda(X_0, \widetilde{X_1}, Y_2, P+1)$
fi
return $\widetilde{X_1} \cup \widetilde{X_2}$

Cette dernière version de la fonction $split_\lambda$ est celle que nous utiliserons dans notre mise en œuvre. C'est la solution la plus générale (puisqu'elle contient les deux autres) et elle permet de moduler facilement le raffinement d'une classe par rapport à un autre, par simple variation du paramètre r.

Algorithme de la fonction de raffinement

Pour compléter cette présentation, nous donnons l'algorithme de la fonction $split_\Lambda$ de raffinement d'une classe X par rapport à un bloc Y.

Algorithme 2-6

```
function split_Λ (X : Class)
    Ñ : set of class;

    Ñ := {X}
     -- Raffinement pour toutes les transitions non stables
    for (X, λ, Y, S) ∈ Raf do
        if Sons(Y) ≠ ∅ ou S = 0 then
            Ñ := split_λ (X, Ñ, Y, 0)
        fi
    od
     -- Mise à jour de Raf pour toutes les transition stables
    for (X, λ, Y, S) ∈ Raf do
        T_Raf := T_Raf \ {(X, λ, Y, S)}
        if Sons(Y) = ∅ et S = 1 then
            for Z ∈ Ñ do
                T_Raf := T_Raf ∪ {(Z, λ, Y, 1)}
            od
        fi
```

■

Ce dernier algorithme termine la présentation générale de l'algorithme de génération de modèle minimal et des aménagements que nous y avons apportés pour son optimisation et sa mise en œuvre. Nous allons maintenant étudier l'adaptation de cet algorithme à différentes équivalences de bisimulation.

2.6 Application à différentes bisimulations

L'algorithme que nous avons présenté est défini par rapport à une notion générale de bisimulation. Néanmoins, une utilisation intéressante de ce genre d'algorithmes se fait en faisant varier la notion d'observabilité des actions du système; d'où la définition de différentes relations d'équivalence, utilisées pour la vérification de différentes classes de propriété.

Dans cette section, nous allons nous intéresser à l'adaptation de l'algorithme de génération de modèle minimal à un nombre limité de ces équivalences de bisimulation : la bisimulation forte, la bisimulation $\tau^* a$ et la bisimulation de branchement. Ces trois équivalences permettent une couverture assez large de l'ensemble des équivalences de bisimulation couramment utilisées dans les outils de vérification.

2.6.1 Bisimulation forte

Cette relation de bisimulation $\sim$ est obtenue en considérant comme observables toutes les actions de A_τ, elle est définie avec l'ensemble :

$$\Lambda = \{\{a\} \mid a \in A_\tau\}$$

Cette relation ne permet aucune abstraction sur les comportements observés. Elle n'identifiera donc que des systèmes présentant exactement le même comportement. Elle est plus forte que les autres relations de bisimulation que nous considérons, et préserve en particulier toutes les propriétés exprimables dans des logiques temporelles telles que HML [HM85] , LTAC[QS83], CTL [CES83] et ses variantes. Du point de vue minimisation, la fonction α de choix de représentants canoniques est l'identité.

Comme l'algorithme de génération de modèle minimal est conçu à l'origine pour la bisimulation forte, aucune modification des algorithmes précédents n'est nécessaire.

2.6.2 Les bisimulations "faibles"

Ces relations introduisent une notion d'abstraction, par la distinction d'actions *visibles*, qui sont celles apparaissant dans les propriétés à vérifier, et d'actions *internes*, dont l'observation n'est pas considérée pertinente pour la vérification des propriétés. Les actions internes sont alors renommées en τ. Du point de vue vérification, cette notion d'abstraction présente un intérêt particulier par rapport à la bisimulation forte, puisqu'il va être possible d'exprimer les spécifications uniquement en terme d'actions pertinentes pour les propriétés à vérifier.

Les bisimulations "faibles" que nous considérons, introduisent dans leur ensemble de langages Λ le langage τ^* en préfixe et/ou en suffixe d'actions visibles. Le calcul de la fonction de raffinement $split_\lambda(X,Y)$ ne repose plus simplement sur le calcul de la fonction pre^a, mais doit être étendu pour tenir compte de langages de la forme $\{\tau^* a\}, \{\tau^* a \tau^*\}, \ldots$ En pratique, le calcul d'une fonction pre^{τ^*}, calcul rendu nécessaire pour ces bisimulations, peut être très coûteux.

Pour pallier le coût important du calcul de cette fonction, une solution adoptée dans les outils de vérification classiques est de pré-calculer la relation de transition correspondant à la fonction pre_λ. Ce calcul consiste à construire un nouveau système de transitions étiquetées en remplaçant dans le modèle chaque séquence correspondant à un élément λ de Λ par une transition étiquetée par le représentant canonique $\alpha(\lambda)$. Ce calcul est introduit dans [Fer88] sous le nom de *forme pré-normale*, que nous noterons $PNF_\Lambda(S)$.

Grâce au calcul de ces formes pré-normales, le quotient du système de transitions étiquetées S pour une relation $\sim_\Lambda$ revient à calculer le quotient de $PNF_\Lambda(S)$ pour la bisimulation forte. De même, la comparaison de deux systèmes de transitions étiquetées S_1 et S_2 se réduit à la comparaison pour la bisimulation forte de leur forme pré-normale. La justification de cette démarche est donnée dans [Fer88].

Le calcul de ces formes pré-normales peut être coûteux, puisqu'il correspond au calcul de la fermeture transitive de la relation de transition de S par rapport aux transitions τ.

L'algorithme classique pour ce genre de calcul est en $O({n_\tau}^3)$ [AHU74]. Certains algorithmes plus élaborés peuvent améliorer cette complexité à $O({n_\tau}^{2.376})$.

Le coût de ce calcul est un problème important, étant donné que la plupart des systèmes de transitions étiquetées rencontrés en pratique contiennent une proportion importante de transitions étiquetées τ et donc un n_τ d'autant plus grand. De plus, cette transformation diminue sensiblement le nombre des états du système entre S et $PNF_\Lambda(S)$ mais tend à augmenter fortement le nombre de transitions.

2.6.3 Equivalence observationnelle

L'équivalence observationnelle $\sim_o$ est l'équivalence de bisimulation obtenue pour l'ensemble $\Lambda = \{\tau^*\} \cup \{\tau^* a \tau^* \mid a \in A\}$.

La fonction de choix de représentants canoniques α est définie comme suit :

$$\alpha(\lambda) = \begin{cases} a_i \text{ si } (\exists i \in [1..n], a_i \in A) \\ \tau \text{ sinon} \end{cases}$$

Le calcul du raffinement de partition pour l'équivalence observationnelle passe par le calcul de la forme prénormale $PNF_o(S)$.

2.6.4 Bisimulation $\tau^* a$

La bisimulation $\tau^* a$ $\sim_{\tau^* a}$ est l'équivalence de bisimulation obtenue pour l'ensemble $\Lambda = \{\tau^* a \mid a \in A\}$. La fonction de choix de représentants canoniques α est telle que $\forall \lambda \in \Lambda, \lambda = \tau^* a \Rightarrow \alpha(\lambda) = \{a\}$.

Deux stratégies d'application du raffinement de partition sont possibles pour cette bisimulation.

1. La première stratégie passe par le calcul de la forme pré-normale $PNF_{\tau^* a}(S) = (Q', A, T', q_0)$ où :

   - $Q' = \{q \in Q \mid \exists a \in A, pre_a(q) \neq \emptyset\} \cup \{q_0\}$
   - $T' = \{(p, a, q) \mid p \in Q' \wedge p \xrightarrow{\tau^* a}_T q\}$

   Le système quotient $S/\sim_{\tau^* a}$ est alors égal à $PNF_{\tau^* a}(S)/\sim$.

2. La deuxième stratégie est basée sur une modification de la fonction de raffinement entre classes :

   $$\forall a \in A, \forall X, Y \in 2^Q, split_\lambda(X, Y) = \{X \cap pre_{\tau^* a}(Y)\} \cup \{X \setminus pre_{\tau^* a}(Y)\}$$

   Les autres opérateurs de raffinement sont définis à partir de cette version modifiée de l'opérateur *split* de "base".

   Le quotient $S/\sim_{\tau^* a}$ est alors obtenu par le calcul de la partition la moins fine, avec ces opérateurs de raffinement modifiés.

2.6.5 Bisimulation de branchement

La bisimulation de branchement [GV90] n'est pas directement exprimable en terme de relation de bisimulation; en effet, sa définition ne coïncide pas directement avec celle donnée en 2.1-1. Une définition de cette relation peut être donnée sous la forme suivante [Mou92] :

Définition 2.6-1 (Bisimulation de branchement)
Soit $\mathcal{B}^{br} : 2^{Q_1 \times Q_2} \longrightarrow 2^{Q_1 \times Q_2}$ *défini par :*

$$\begin{aligned}
\mathcal{B}^{br}(R) \;=\; & \{(p_1,p_2) \mid \forall a \in \mathcal{A} \cup \{\tau\} \,. \\
& \forall q_1 \,.\, (p_1 \xrightarrow{a}_{T_1} q_1 \Rightarrow (a = \tau \wedge (q_1,p_2) \in R) \vee \\
& \qquad (\exists q_2 q_2' \,.\, (p_2 \xrightarrow{\tau^*}_{T_2} q_2' \wedge q_2' \xrightarrow{a}_{T_2} q_2 \wedge (p_1,q_2') \in R \wedge (q_1,q_2) \in R))) \\
& \forall q_2 \,.\, (p_2 \xrightarrow{a}_{T_2} q_2 \Rightarrow (a = \tau \wedge (p_1,q_2) \in R) \vee \\
& \qquad (\exists q_1 q_1' \,.\, (p_1 \xrightarrow{\tau^*}_{T_1} q_1' \wedge q_1' \xrightarrow{a}_{T_1} q_1 \wedge (q_1',p_2) \in R \wedge (q_1,q_2) \in R)))\}
\end{aligned}$$

Une relation R *sur* $Q_1 \times Q_2$. R *est une bisimulation de branchement si et seulement si* $R \subseteq \mathcal{B}^{br}(R)$. ■

Même si la définition de la bisimulation de branchement ne coïncide pas tout a fait avec la notion classique de bisimulation, certains travaux [GV90] ont introduit une variante du *RCP problem* qui donne la partition associée à la bisimulation de branchement. Cette variante est connue sous le nom de *Relational Coarsest Partition with Stuttering problem*(RCPS). Groote et Vaandrager proposent un algorithme résolvant ce problème, suivant une méthode similaire à celle utilisée dans le cas du *RCP problem*.

L'avantage de cette approche sur celle appliquée dans le cas de relations comme la bisimulation observationnelle est qu'il n'est pas nécessaire de calculer une forme pré-normale coûteuse en temps de calcul.

De plus, cette bisimulation offre un certain nombre d'avantages du point de vue de la vérification :

- elle est plus forte que la majorité des équivalences de bisimulation "usuelles"(hormis la bisimulation forte), et conserve une procédure de décision efficace.
- elle préserve la structure de branchement des modèles.
- elle préserve une extension de la logique de Hennessy-Milner [NV90]
- elle préserve une restriction de la logique CTL (CTL-X)

Nous présentons la notion de compatibilité et les opérateurs de raffinement associés au *RCPS problem*.

$$\begin{aligned}
\forall a \in A_\tau, \mathcal{F}_a(X,Y) \;&=\; \mu Z.(X \cap pre^\tau(Z) \cup X \cap pre^a(Y)) \\
split_a(X,Y) \;&=\; \{\mathcal{F}_a(X,Y), X \setminus \mathcal{F}_a(X,Y)\} \wedge (a \neq \tau) \\
split_\tau(X,Y) \;&=\; \{\mathcal{F}_\tau(X,Y), X \setminus \mathcal{F}_\tau(X,Y)\} \wedge (X \neq Y) \\
split_\tau(X,X) \;&=\; \{X\}
\end{aligned}$$

$$split(X,Y) = \prod_{a \in A_\tau} split_a(X,Y)$$
$$split(X,\rho) = \prod_{Y \in \rho} split(X,Y)$$

La fonction de choix de représentants canoniques α est l'identité.

Les propriétés vérifiées par $\mathcal{F}_a$ sont légèrement différentes de celles de l'opérateur pre_λ :

$\mathcal{F}_a(X_1 \cup X_2, Y) \supseteq \mathcal{F}_a(X_1,Y) \cup \mathcal{F}_a(X_2,Y)$; mais l'égalité n'est pas vérifiée.

Cette propriété implique que la stabilité n'est pas héritée par raffinement [GV90]. La proposition 2.5-6 n'est pas vérifiée pour la bisimulation de branchement. La mise en œuvre de l'algorithme pour la bisimulation de branchement implique donc certaines modifications.

Boucles de τ

La définition de la fonction *split* indique qu'il ne faut pas raffiner une classe X par rapport à elle-même si l'étiquette considérée est τ. Par contre, nous voulons pouvoir raffiner deux classes X et Y par rapport à τ, si $X \neq Y$. Pour pouvoir traiter ce cas particulier, sans changer profondément le déroulement de l'algorithme du raffinement, nous conservons une boucle τ sur chaque classe, par l'introduction dans T_ρ de la transition $(X,\tau,X,0)$ pour tout $X \in \rho_{init}$. Ainsi, nous permettons la création des transitions (X_1,τ,X_2) et (X_2,τ,X_1) lors du raffinement de X en $\{X_1,X_2\}$. L'interdiction du raffinement de la classe X par rapport à elle-même pour τ se fait alors par un simple test au niveau de la fonction $split_\Lambda$. Lorsque l'algorithme termine, nous éliminons alors les transitions $(X,\tau,X,s)\ \forall X \in \rho$ de T_ρ, pour obtenir le modèle minimal pour la bisimulation de branchement.

Non héritage de la stabilité par raffinement

La propriété d'héritage de la stabilité par raffinement n'est plus vérifiée. Par conséquent, si nous effectuons un raffinement de X par rapport à Y avec comme résultat $\{X_1,X_2\}$, il faut alors considérer la suite du raffinement de X par les transitions $t' = (X,a',Y',1)$ telle que Y' est un bloc simple, ce qui n'était pas le cas précédemment. En effet, la stabilité de X par rapport à Y' qu'indique cette transition n'est plus du tout garantie pour $\{X_1,X_2\}$. Le raffinement complet de X passe alors par le raffinement pour les transitions considérées comme stables jusque là.

```
function split_Λ(X : Class)
    Ñ : set of class;

    Ñ := {X}
     -- Raffinement pour toutes les transitions non stables
    for (X, a, Y, S) ∈ Raf do
        if Sons(Y) ≠ ∅ ou S = 0 then
            Ñ := split_λ(Ñ, a, Y)
        fi
    od
     -- Si le raffinement a été productif, on continue le raffinement
     -- pour toutes les transitions qui étaient stables
    if Ñ ≠ {X} then
        for (X, a, Y, S) ∈ Raf do
            if (X ≠ Y) et Sons(Y) = ∅ et S = 1 then
                Ñ := split_λ(Ñ, a, Y)
            if
        od
    else
         -- Mise à jour de Raf pour toutes les transition stables
        for (X, a, Y, S) ∈ Raf do
            T_Raf := T_Raf \ {(X, a, Y, S)}
            if Sons(Y) = ∅ et S = 1 then
                for Z ∈ Ñ do
                    T_Raf := T_Raf ∪ {(Z, a, Y, 1)}
                od
            fi
        od
    if
```

La fonction $split_\lambda$ est définie en substituant à la fonction pre_λ la fonction $\mathcal{F}_a$.

Nous avons défini l'adaptation de notre mise en œuvre de l'algorithme de génération de modèle minimal à différentes bisimulations. Nous allons maintenant illustrer le fonctionnement de l'algorithme sur petit exemple de protocole.

2.7 Exemple de génération de modèle minimal

Nous allons illustrer le fonctionnement de l'algorithme, ainsi que les mises à jour des différentes structures. L'exemple suivant est celui d'un protocole réunissant un processus Emetteur et un processus Récepteur. Le processus Emetteur émet un message par l'intermédiaire d'un processus Emet qui va répéter la transmission jusqu'à récupération d'un accusé de réception. Le récepteur peut simuler la perte d'un message en invalidant l'accusé de réception.

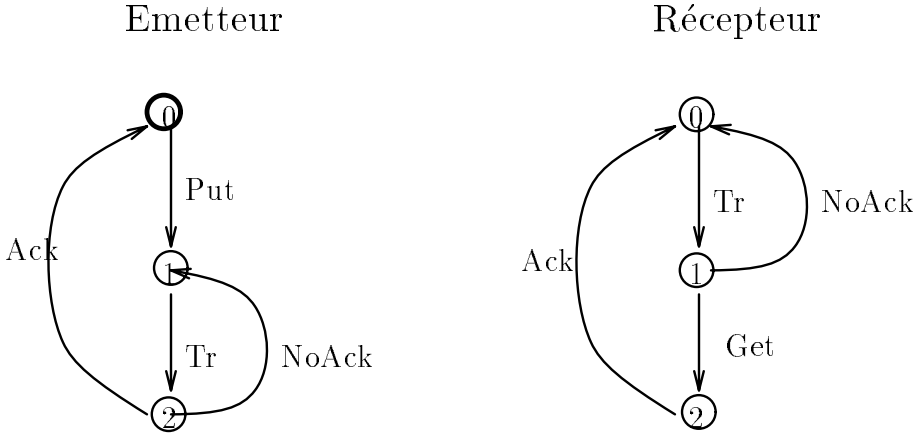


Figure 2.4: Exemple de génération de modèle minimal

Ces processus sont modélisés par les systèmes de transitions étiquetées de la figure 2.4. Ils sont composés parallèlement d'après l'expression suivante (syntaxe LOTOS) :

```
hide Tr,Ack,NoAck in
     Emetteur[Put,Ack,Tr,NoAck]
|[Ack,Tr,NoAck]|
     recepteur[Get,Ack,Tr,NoAck]
```

Nous notons respectivement $S_1 = (Q_1, A_1, T_1, init_1)$ le système de transitions étiquetées de l'émetteur et $S_2 = (Q_2, A_2, T_2, init_2)$ le système de transitions étiquetées du récepteur; nous avons alors les ensembles suivants :

- $Q_1 = Q_2 = \{0, 1, 2\}$
- $A_1 = \{Put, Tr, Ack, NoAck\}$, $A_2 = \{Get, Tr, Ack, NoAck\}$
- $init_1 = init_2 = \{0\}$

Le système $S = (Q, A, T, init)$ qui correspond à la composition parallèle de Emetteur et Récepteur est défini comme suit :

- $Q = Q_1 \times Q_2$
- $A = A_1 \cup A_2$
- $init = \{(0,0)\}$

Dans ce système, les actions Put et Get sont des actions *asynchrones*, alors que les actions Tr, Ack et $NoAck$ sont des actions *synchrones*. De plus, les actions Tr, Ack et $NoAck$ sont

considérées comme internes : dans le modèle global de ce système, elles seront remplacées par τ.

La relation de transition de S est la suivante :

- $\xrightarrow{Put}_T = \{((0,q),(1,q)) \mid q \in Q_2\}$
- $\xrightarrow{Get}_T = \{((q,1),(q,2)) \mid q \in Q_1\}$
- $\xrightarrow{\tau}_T = \xrightarrow{Tr}_T \cup \xrightarrow{Ack}_T \cup \xrightarrow{NoAck}_T$
  $= \{((1,0),(2,1)),((2,1),(1,0)),((2,2),(0,0))\}$

2.7.1 Application de la génération de modèle minimal: bisimulation forte

La partition initiale ρ_{init} est égale à $\{Q\}$.

Le système de transition $\mathcal{R}af$ est initialisé comme suit :

- $\mathcal{N} = \rho_{init}$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_0$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_0, Put, P_0, 0), (P_0, Get, P_0, 0), (P_0, \tau, P_0, 0)\}$

Enfin la fonction de décomposition est telle que $D(root) = P_0$.

Dans la suite, le résultat de chaque raffinement est représenté par une figure. Ces figures sont composées en partie *gauche* d'un dessin représentant la *partition courante* ρ. Sur ce dessin seront portées les transitions *stables* menant d'une classe de ρ à une autre classe de ρ.

En partie droite du dessin, nous représentons l'*arbre de raffinement* $\mathcal{D}$(traits épais) décoré par les transitions instables ou menant à des classes raffinées n'appartenant plus à la partition courante (trait fins). Cet arbre de raffinement est donné sous la forme d'un arbre n-aire pour simplifier la représentation.

Ces conditions initiales d'application de la génération de modèle minimal sont représentés par la figure 2.5.

Premier raffinement

Nous allons effectuer un premier raffinement correspondant par rapport aux transitions $(P_0, Put, P_0, 0), (P_0, Get, P_0, 0)$ et $(P_0, \tau, P_0, 0)$. Ces transitions sont ôtées de $\mathcal{D}$. Nous avons alors :

$\boxed{P_0, Put, P_0}$

$pre_{Put}(P_0) \quad = \quad \{(0,q) \mid q \in Q_2\}$

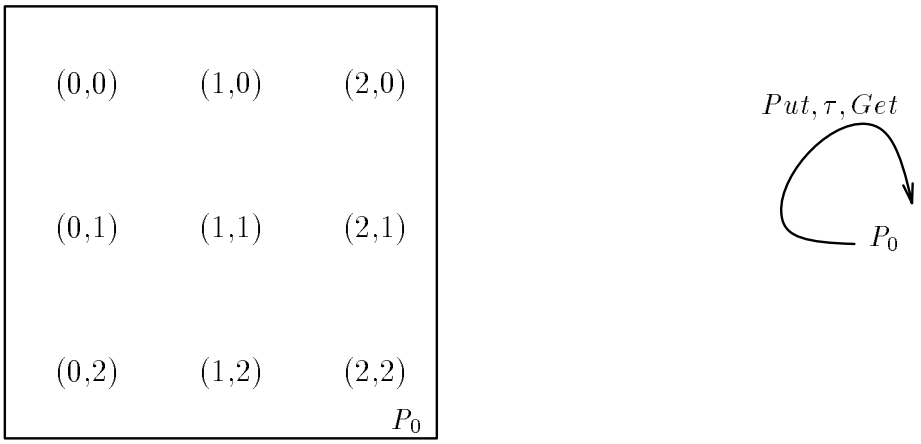


Figure 2.5: Conditions initiales

$$\begin{aligned} split_{Put}(P_0, P_0) &= \{P_0', P_1\} \text{ avec} \\ P_0' &= \{(0,q) \mid q \in Q_2\} = \{(0,0),(0,1),(0,2)\} \\ P_1 &= \{(q_1,q_2) \mid (q_1,q_2) \in Q, q_1 \neq 0\} \end{aligned}$$

Comme la classe P_0 contient $init$, nous continuons le raffinement de la classe ayant gardé $init$, c'est-à-dire P_0'.

Nous mettons à jour $\mathcal{D}$ pour la classe P_1, pour des raffinements ultérieurs :

$$\mathcal{D} = \mathcal{D} \cup \{(P_1, Get, P_0, 0), (P_1, \tau, P_0, 0)\}$$

Le raffinement de P_0' continue pour la transition suivante $(P_0, Get, P_0, 0)$:

$\boxed{P_0', Get, P_0}$

$$pre_{Get}(P_0) = \{(q,1) \mid q \in Q_1\}$$

$$\begin{aligned} split_{Get}(P_0', P_0) &= \{P_0'', P_2\} \text{ avec} \\ P_2 &= \{(0,1)\} \\ P_0'' &= \{(0,0),(0,2)\} \end{aligned}$$

De la même manière, nous isolons P_0'', puisqu'elle contient l'état initial et nous mettons à jour $\mathcal{D}$.

$$\mathcal{D} = \mathcal{D} \cup \{(P_2, Put, P_0, 1), (P_2, Get, P_0, 1), (P_1, \tau, P_0, 0)\}$$

Ce premier raffinement se termine pour la transition $(P_0, \tau, P_0, 0)$.

$\boxed{P_0'', \tau, P_0}$

$$pre_\tau(P_0) = \{(1,0),(2,1),(2,2)\}$$

$$split_\tau(P_0'', P_0) = P_0'' \text{ car } P_0'' \cup pre_\tau(P_0) = \emptyset$$

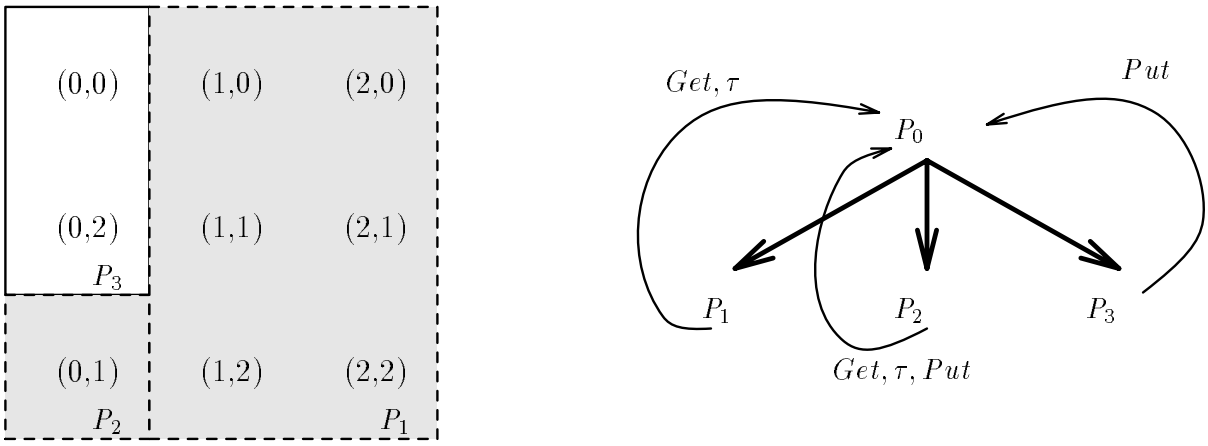


Figure 2.6: Premier raffinement

Nous noterons $P_3 = P_0''$. Les transitions pour P_3 sont maintenant insérées dans $\mathcal{D}$:

$$\mathcal{D} = \mathcal{D} \cup \{(P_3, Put, P_0, 1)\}$$

Le résultat final de cette première phase de raffinement est donné en figure 2.6.

A l'issue de ce premier raffinement, nous avons

- $\rho = \{P_1, P_2, P_3\}$
- $\pi = \{P_3\}, \sigma = \emptyset$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_3$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_3, Put, P_0, 1),$
  $(P_2, Put, P_0, 1), (P_2, \tau, P_0, 0), (P_2, Get, P_0, 1),$
  $(P_1, \tau, P_0, 0), (P_1, Get, P_0, 0)\}$

Deuxième raffinement

La classe contenant l'état initial est P_3, nous voulons la raffiner par rapport à la transition $(P_3, Put, P_0, 1)$. Comme P_0 est maintenant décomposée, cette transition n'est pas stable. Nous allons donc calculer le raffinement de P_3 par rapport à l'ensemble $\{P_1, P_2, P_3\}$ pour l'action Put.

$$\boxed{P_3, Put, \{P_1, P_2, P_3\}}$$

$$\begin{aligned} pre_{Put}(P_2) &= pre_{Put}(P_3) = \emptyset \\ pre_{Put}(P_1) &= P_3 \end{aligned}$$

donc P_2 est stable. Nous avons maintenant

- $\pi = \{P_3, P_1\}, \sigma = \{P_3\}$
- $\mathcal{T}_{\mathcal{R}af} = \mathcal{T}_{\mathcal{R}af} \setminus \{(P_3, Put, P_1, 1)\} \cup \{(P_3, Put, P_1, 1)\}$

Troisième raffinement

Le seul élément de $\pi \setminus \sigma$ est la classe P_1. Le raffinement de P_1 se fait pour les transitions $(P_1, \tau, P_0, 0), (P_1, Get, P_0, 0)$.

$\boxed{P_1, \tau, \{P_1, P_2, P_3\}}$

$$\begin{aligned} pre_\tau(P_2) &= \emptyset \\ pre_\tau(P_3) &= \{(2,2)\} \\ pre_\tau(P_1) &= \{(1,0),(2,1)\} \end{aligned}$$

$$\begin{aligned} split_\tau(P_1, \{P_1, P_2, P_3) &= \{P_4, P_5, P_6\} \text{ avec} \\ P_4 &= \{(1,0),(2,1)\} \\ P_5 &= \{(2,2)\} \\ P_6 &= \{(2,0),(1,1),(1,2)\} \end{aligned}$$

Comme le raffinement par rapport à cette transition a été productif, nous arrêtons le raffinement de P_1. La classe P_3 est maintenant ôtée de σ.

L'ensemble des structures mises à jour devient :

- $\rho = \{P_2, P_3, P_4, P_5, P_6\}$
- $\pi = \{P_3\}, \sigma = \emptyset$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_3$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_3, Put, P_1, 1),$
  $(P_4, \tau, P_1, 1), (P_4, Get, P_0, 0),$
  $(P_5, \tau, P_3, 1), (P_5, Get, P_0, 0),$
  $(P_6, Get, P_0, 0),$
  $(P_2, Put, P_0, 0), (P_2, \tau, P_0, 0), (P_2, Get, P_0, 1)\}$

Quatrième raffinement

La classe P_3 devient de nouveau candidate au raffinement, par rapport à la transition $(P_3, Put, P_1, 1)$. La classe P_1 est décomposée en $\{P_4, P_5, P_6\}$.

$\boxed{P_3, Put, \{P_4, P_5, P_6\}}$

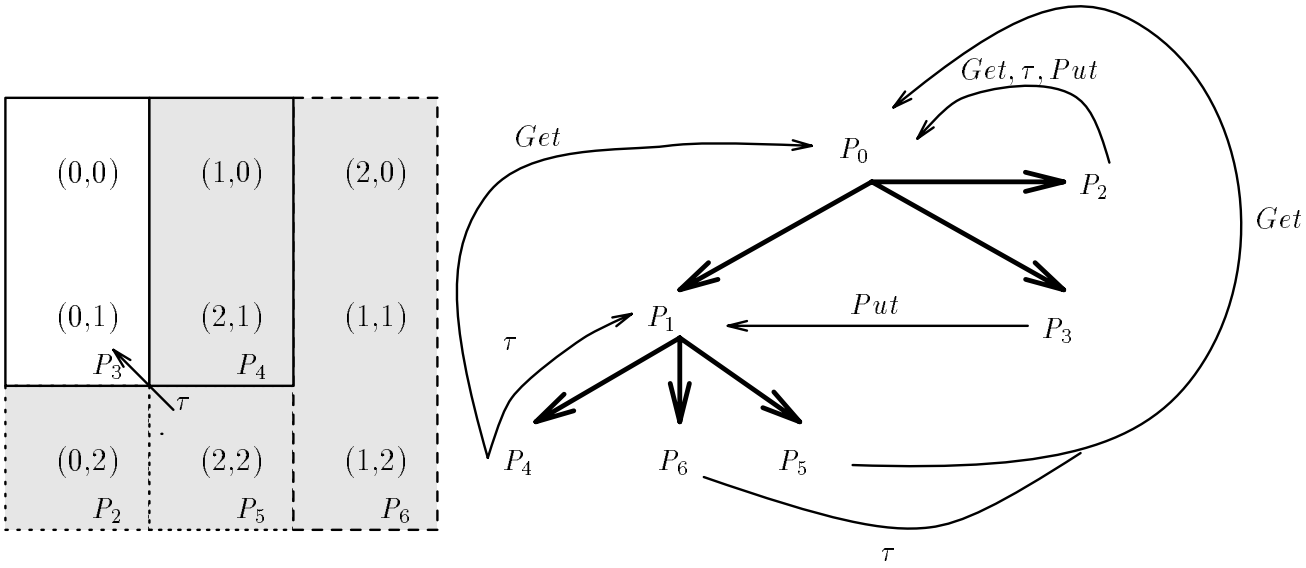


Figure 2.7: Troisième raffinement

$$
\begin{aligned}
pre_{Put}(P_5) &= \emptyset \\
pre_{Put}(P_6) &= \{(0,1),(0,2)\} \\
pre_{Put}(P_4) &= \{(0,0)\}
\end{aligned}
$$

$$
\begin{aligned}
split_{Put}(P_3, \{P_4, P_5, P_6) &= \{P_7, P_8\} \text{ avec} \\
P_7 &= \{(0,0)\} \\
P_8 &= \{(0,1)\}
\end{aligned}
$$

A l'issue de ce raffinement, la classe P_7 qui contient l'état initial est stable (car toutes ses transitions le sont). Nous pouvons donc insérer P_4 dans π.

- $\rho = \{P_2, P_4, P_5, P_6, P_7, P_8\}$
- $\pi = \{P_7, P_4\}, \sigma = \{P_7\}$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_7$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_7, Put, P_4, 1),$
  $(P_8, Put, P_6, 1),$
  $(P_4, \tau, P_1, 1), (P_4, Get, P_0, 0),$
  $(P_5, \tau, P_3, 1), (P_5, Get, P_0, 0),$
  $(P_6, Get, P_0, 0),$
  $(P_2, Put, P_0, 0), (P_2, \tau, P_0, 0), (P_2, Get, P_0, 1)\}$

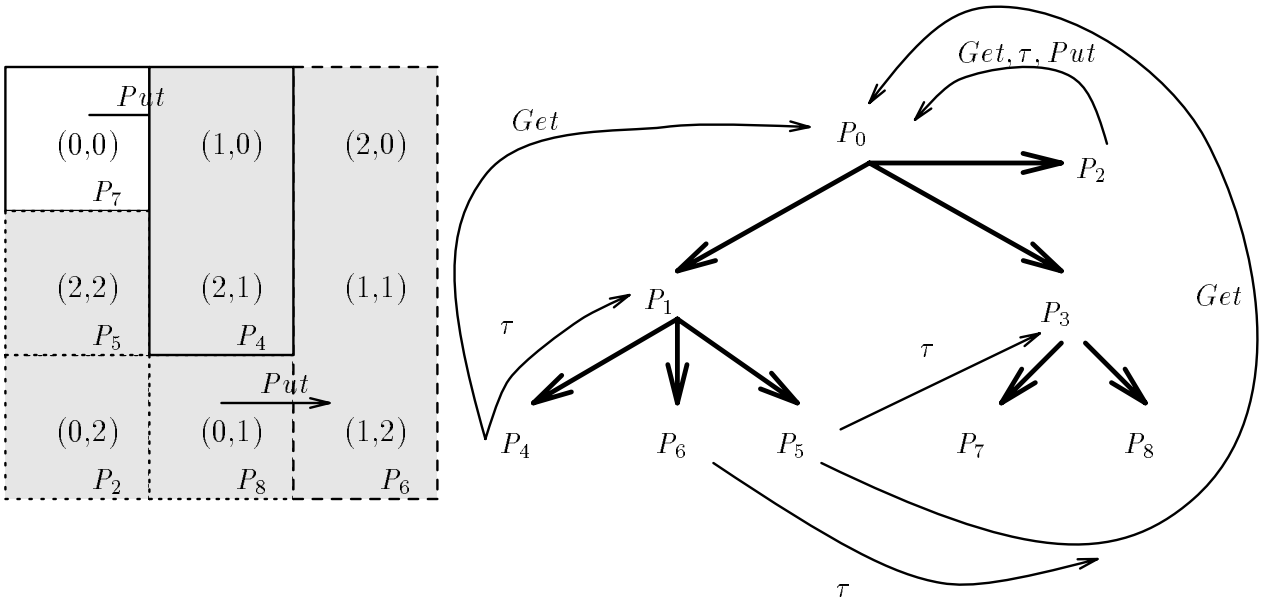


Figure 2.8: Quatrième raffinement

Cinquième raffinement

Le raffinement de P_4 pour la transition $(P_4, \tau, P_1, 1)$ ne va pas provoquer de raffinement supplémentaire. Par contre, le raffinement pour $(P_4, Get, P_0, 0)$ va être productif pour la classe P_5 :

$$\boxed{P_4, Get, \{P_2, P_4, P_5, P_6, P_7, P_8\}}$$

$$pre_{Get}(P_5) = \{(2,1)\}$$

$$split_{Get}(P_4, P_5) = \{P_9, P_{10}\} \text{ avec}$$
$$P_9 = \{(1,0)\}$$
$$P_{10} = \{(2,1)\}$$

La relation de transition est mise à jour (figure 2.9) :

- $\rho = \{P_2, P_5, P_6, P_7, P_8, P_9, P_{10}\}$
- $\pi = \{P_7\}, \sigma = \emptyset$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_7$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_7, Put, P_4, 1),$
  $(P_9, \tau, P_4, 1),$

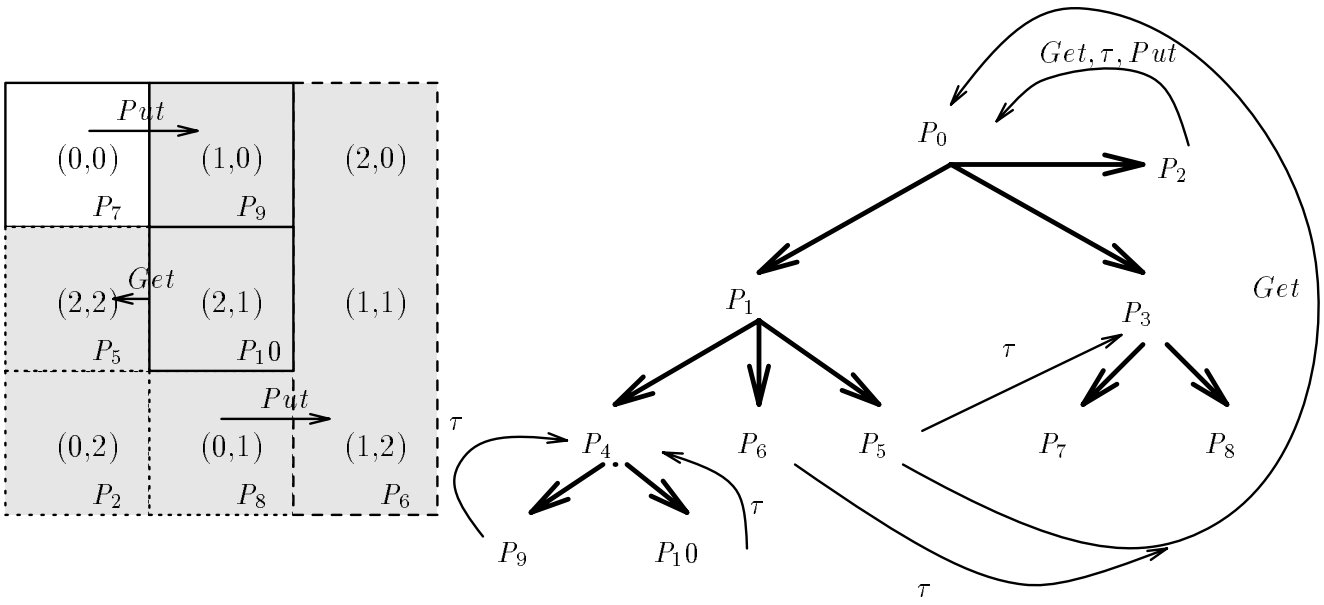


Figure 2.9: Cinquième raffinement

$(P_{10}, \tau, P_4, 1), (P_{10}, Get, P_5, 1),$
$(P_5, \tau, P_3, 1), (P_5, Get, P_0, 0),$
$(P_8, Put, P_6, 1),$
$(P_6, Get, P_0, 0),$
$(P_2, Put, P_0, 0), (P_2, \tau, P_0, 0), (P_2, Get, P_0, 1)\}$

Fin du raffinement

Les étapes suivantes ne provoqueront plus aucun raffinement, mais permettront de stabiliser les transitions entre les classes accessibles et donc de stabiliser les classes de π .

Le résultat du raffinement est donné en figure 2.10

- $\rho = \{P_2, P_5, P_6, P_7, P_8, P_9, P_{10}\}$
- $\pi = \{P_7\}, \sigma = \emptyset$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_7.$
- $\mathcal{T}_{\mathcal{R}af} = \{(P_7, Put, P_9, 1),$
  $(P_9, \tau, P_{10}, 1),$
  $(P_{10}, \tau, P_9, 1), (P_{10}, Get, P_5, 1),$
  $(P_5, \tau, P_7, 1)(P_8, Put, P_6, 1),$

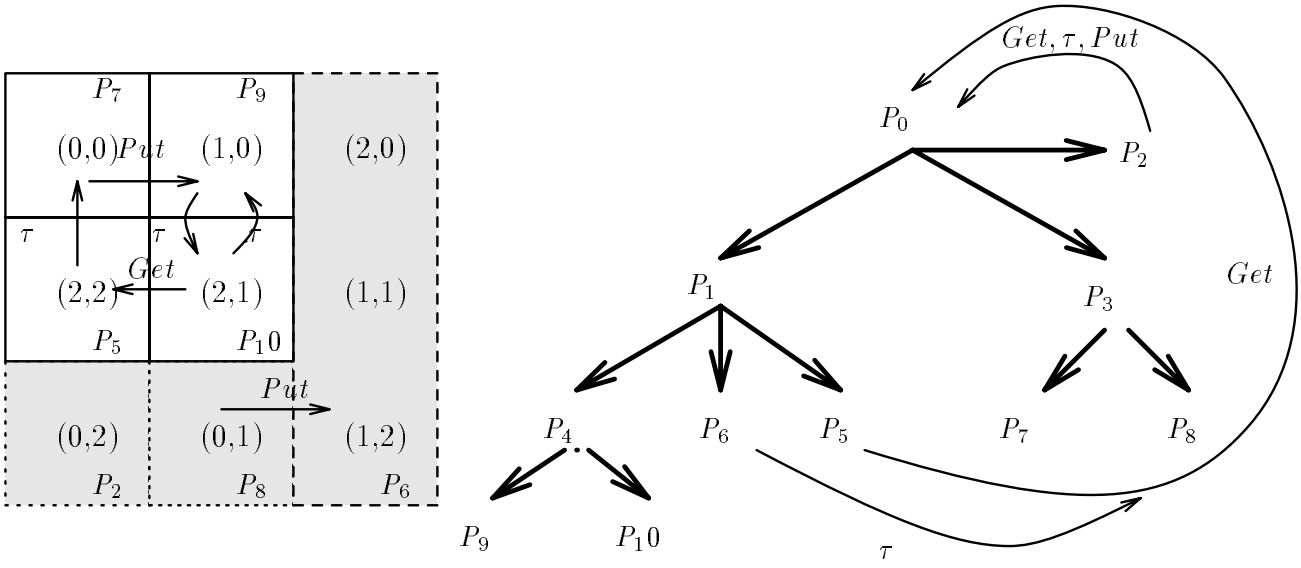


Figure 2.10: Résultat final

$(P_6, Get, P_0, 0)$,
$(P_2, Put, P_0, 0), (P_2, \tau, P_0, 0), (P_2, Get, P_0, 1)\}$

2.7.2 Application de la génération de modèle minimal : bisimulation $\tau^* a$

Le traitement du même exemple dans le cas de la bisimulation $\tau^* a$ permet de mettre plus en valeur l'intérêt de l'algorithme. Nous partons des mêmes conditions initiales que dans l'exemple précédent.

Nous calculons les fonctions $pre_{\tau^* Put}$ et $pre_{\tau^* Get}$ par composition de la pre_{Get} ou pre_{Put} et de la fermeture transitive de la fonction pre_τ.

premier raffinement

- $pre_{\tau^* Put}(P_0) = \{(0,0), (2,2)\}$
- $pre_{\tau^* Get}(P_0) = \{(1,0), (2,1)\}$

$$\begin{aligned} split_{\tau^* Put}(P_0, P_0) &= \{P_1, P_0'\} \text{ avec} \\ P_1 &= \{(0,0), (2,2)\} \\ P_0' &= \{(0,1), (0,2), (1,0), (1,1), (1,2), (2,0), (2,1)\} \end{aligned}$$

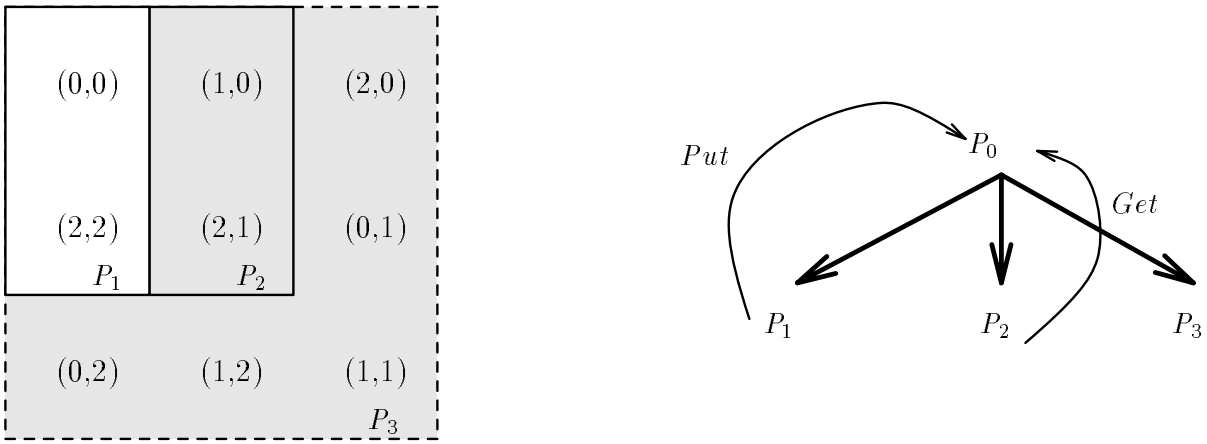


Figure 2.11: Bisimulation τ^*a : premier raffinement

$$
\begin{aligned}
pre_{\tau^*Get}(P_0) \cap P_1 &= \emptyset \\
split_{\tau^*Get}(P_0', P_0) &= \{P_2, P_3\} \text{ avec} \\
P_2 &= \{(1,0),(2,1)\} \\
P_3 &= \{(0,1),(0,2),(1,1),(1,2),(2,0)\}
\end{aligned}
$$

A l'issue de ce premier raffinement, nous avons

- $\rho = \{P_1, P_2, P_3\}$
- $\pi = \{P_1\}, \sigma = \emptyset$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_1$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_1, Put, P_0, 1), (P_2, Get, P_0, 1)\}$

Le résultat de ce premier raffinement est donné en figure 2.11.

Deuxième raffinement

La classe contenant l'état initial est la classe P_1. Le prochain raffinement se fait donc par rapport à la transition $(P_1, Put, P_0, 1)$. Comme P_0 est décomposée, cette transition est instable. Le raffinement s'effectuera par rapport à l'ensemble de classes $\{P_1, P_2, P_3\}$.

$$
\begin{aligned}
pre_{\tau^*Put}(P_1) &= pre_{\tau^*Put}(P_3) = \emptyset \\
pre_{\tau^*Put}(P_2) &= P_1
\end{aligned}
$$

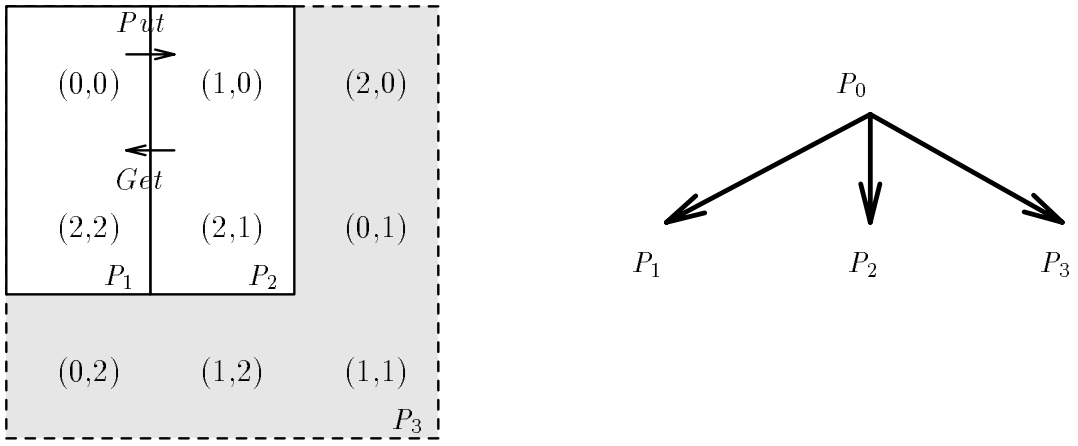


Figure 2.12: Bisimulation $\tau^* a$: résultat de la génération de modèle minimal

Donc la classe P_1 est maintenant stable et $\mathcal{T}_{\mathcal{R}af}$ est mis à jour :

$$\mathcal{T}_{\mathcal{R}af} = \{(P_1, Put, P_2, 1), (P_2, Get, P_0, 1)\}$$

La classe P_2 est insérée dans π et devient la prochaine candidate au raffinement. Le raffinement de P_2 s'effectue par rapport à P_0, donc par rapport à l'ensemble $\{P_1, P_2, P_3\}$.

$$\begin{aligned} pre_{\tau^* Get}(P_2) &= pre_{\tau^* Get}(P_3) = \emptyset \\ pre_{\tau^* Get}(P_1) &= P_2 \end{aligned}$$

La classe P_2 est elle aussi stabilisée. Nous avons maintenant $\pi = \sigma = \{P_1, P_2\}$, donc l'algorithme se termine avec les résultats suivants(figure 2.12) :

- $\rho = \{P_1, P_2, P_3\}$
- $Init_{\mathcal{R}af} = [(0,0)]_\rho = P_1$.
- $\mathcal{T}_{\mathcal{R}af} = \{(P_1, Put, P_2, 1), (P_2, Get, P_1, 1)\}$

2.8 Conclusion

Dans ce chapitre, nous avons présenté une méthode classique de minimisation et de comparaison de modèles par rapport à une relation de bisimulation. Cette méthode est basée sur le raffinement d'une partition des états du modèle, jusqu'à obtention d'une partition qui coincide avec la relation considérée. Les applications classiques de cet algorithme sépare les phases de calcul des états accessibles d'un modèle et calcul du raffinement de partition sur ces états; nous avons présenté un algorithme permettant d'effectuer les deux calculs conjointement. La minimisation d'un modèle s'effectue alors *pendant* sa génération.

A partir de la description générale de cet algorithme, nous avons approfondi les définitions des fonctions et algorithmes nécessaires pour sa mise en œuvre. Nous avons étudié des optimisations pouvant intervenir à différents niveaux du processus de raffinement.

Enfin, nous avons présenté l'adaptation de l'algorithme de génération de modèle minimal à différentes bisimulations; parmi celles-ci, la bisimulation de branchement a nécessité certaines modifications dans les algorithmes, mais nous verrons que la particularité de son opérateur de raffinement permet d'obtenir des performances très intéressantes.

Les éléments manipulés par cet algorithme sont des ensembles d'états et toutes les opérations nécessaires pour une implémentation sont des opérations sur des ensembles d'états. Cet algorithme se prête donc particulièrement bien à une représentation symbolique du modèle. Dans la suite de ce document, nous présentons deux méthodes de représentation symbolique que nous avons étudiées.

Partie II

Méthodes symboliques de représentation

Chapitre 3

Diagrammes de Décision Binaire

Les Diagrammes de Décision Binaire(BDD) [Bry86] sont une méthode de représentation *canonique* et de manipulation *efficace* de fonctions booléennes. Ils permettent en général des représentations beaucoup plus compactes que les représentations classiques de fonctions booléennes. Les BDDs sont maintenant utilisés avec succès dans de nombreux outils : un des premier outils qui a vraiment mis en valeur l'utilisation des BDDs est certainement l'outil PRIAM [BM88] développé dans le laboratoires de BULL. Cet outil permet de comparer la machine réalisée par les concepteurs d'un circuit combinatoire et la machine correspondant aux spécifications de ce circuit. Il effectue cette comparaison par un parcours en largeur d'abord du produit de ces deux machines, qui sont déterministes.

Des travaux ultérieurs se sont portés sur la vérification de formules logiques à l'aide de BDDs. Un exemple est donné dans [BCM+90], où des méthodes d'application des BDDs pour la vérification de formules du Mu-Calcul sont proposées. A partir d'algorithmes généraux de vérification de formules du Mu-Calcul, ils montrent comment dériver des algorithmes de vérification pour les logiques CTL et PLTL, ainsi que pour la comparaison de machines à états finis pour des relations de bisimulation. Une réalisation notable de ces méthodes pour la vérification de formules de CTL dans le cadre de la validation de circuits est le système SMV, élaboré par [McM92]. A l'aide de ce système, qui fait maintenant l'objet d'une utilisation industrielle, il est maintenant possible de travailler couramment sur des systèmes ayant plus de 10^{100} états. La vérification d'un système ayant de l'ordre de 10^{1300} états est même cité dans [McM92].

On retrouve un système de vérification de circuits séquentiels pour des formules de CTL dans les laboratoires de BULL, le système SIAM [Cou91].

En dehors de la vérification formelle, les BDDs ont aussi été utilisés dans d'autres domaines comme la simulation symbolique de systèmes, la maintenance de raisonnement et l'analyse de fiabilité de systèmes [Mad90],... [Bry92]

3.1 Définition

Un Diagramme de Décision Binaire est une forme *canonique*, obtenue par réduction d'un arbre de décision ordonné. Nous décrivons rapidement la méthode de construction de cet arbre, puis la méthode de réduction permettant d'obtenir un BDD canonique.

3.1.1 Arbre de décision ordonné

Un arbre de décision ordonné est un arbre binaire orienté possédant une racine unique. La construction de cet arbre est basée sur le théorème d'expansion de Shannon [Sha38] des fonctions booléennes.

Définition 3.1-1 (Expansion de Shannon [Sha38])
Soit une fonction booléenne $f(x_1, \ldots, x_n)$ *de* $\{0,1\}^n$ *dans* $\{0,1\}$, *l'expansion de Shannon de* f *par rapport à la variable* x_i *est le couple de fonctions* $f_{\overline{x_i}} : \{0,1\}^{n-1} \rightarrow \{0,1\}$ *et* $f_{x_i} : \{0,1\}^{n-1} \rightarrow \{0,1\}$ *telles que :*

$$\begin{aligned} f_{\overline{x_i}} &= f(x_1, \ldots, x_{i-1}, 0, x_{i+1}, \ldots, x_n) \\ f_{x_i} &= f(x_1, \ldots, x_{i-1}, 1, x_{i+1}, \ldots, x_n) \end{aligned}$$

■

Nous pouvons aussi définir la fonction f par rapport à son expansion de Shannon de la manière suivante :

$$f = \overline{x}_i \wedge f_{\overline{x_i}} \vee x_i \wedge f_{x_i}$$

Nous notons $\vec{x} = \{x_1, \ldots, x_n\}$ l'ensemble des variables booléennes utilisées par la fonction.

L'arbre de décision est alors construit de la manière suivante : chaque nœud de l'arbre de décision est construit en appliquant l'expansion de Shannon. Un nœud est étiqueté par une variable booléenne (la *variable d'expansion*) de $\vec{x}$ et les arcs issus d'un nœud correspondent aux deux fonctions (les *cofacteurs*) construites lors de l'expansion.

Les feuilles de cet arbre sont appelées *nœuds terminaux* et sont étiquetées par les valeurs **0** (pour *faux*) et **1** (pour *vrai*). L'ensemble $\vec{x}$ est ordonné et l'étiquetage des nœuds de l'arbre reflète cet ordre : si il y a un arc entre un nœud étiqueté x_i et un nœud étiqueté x_j, alors $x_i < x_j$.

Un exemple de graphe de décision ordonné pour la formule $(a \vee b) \wedge (c \vee d)$, et l'ordre $a < b < c < d$, est donné par la figure 3.1.

Diagramme de Décision Binaire
La construction d'un BDD à partir d'un arbre de décision ordonné dépend de deux règles de réduction :

1. Si deux nœuds u et v de ce graphe sont isomorphes (même variable, mêmes successeurs), on élimine u et on redirige tous les arcs arrivant en u vers v.

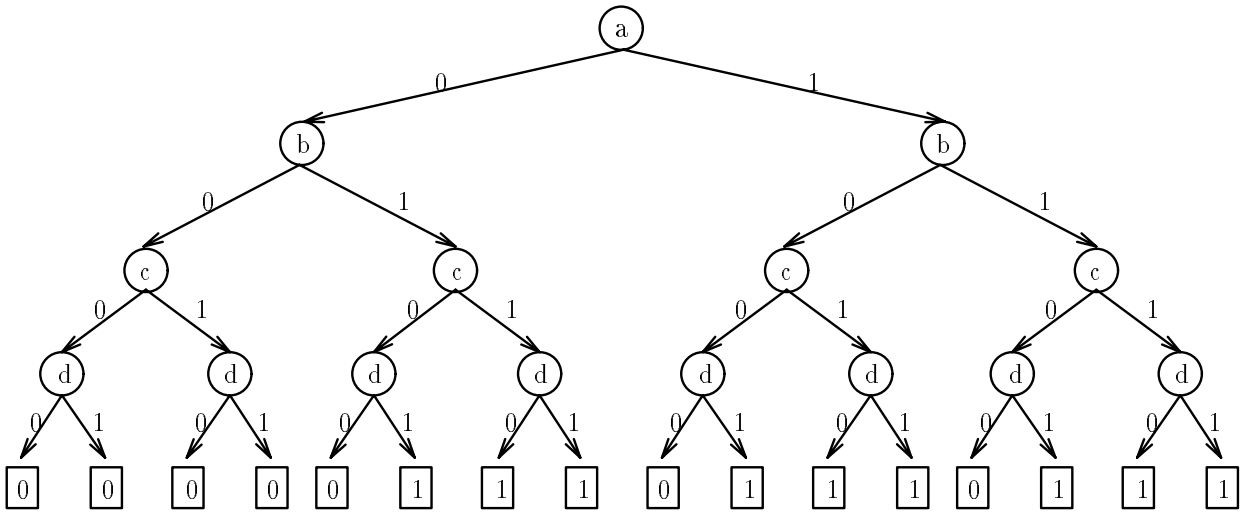


Figure 3.1: Graphe de décision ordonné

2. Si un nœud u a le même successeur v par ces deux arcs de sortie, alors u est éliminé et tous ses arcs incidents sont redirigés vers v.

Ces deux règles sont appliquées dans l'arbre de décision, en partant des feuilles et en remontant jusqu'à la racine (ordre "bottom-up"), jusqu'à ce qu'elles ne produisent plus aucune réduction. La figure 3.2 montre le résultat après application des règles de réduction aux feuilles et aux parents des feuilles. Dans cette figure, seuls les arcs menant au nœud terminal **1** sont représentés. Les nœuds marqués = sont isomorphes et seront donc confondus en un seul nœud (règle 1), provoquant la disparition du nœud marqué * (règle 2).

La figure 3.3 montre le résultat final.

3.1.2 Notations

Dans la suite de ce chapitre, nous utiliserons les notations suivantes :

- Opérateurs usuels : Les opérateurs Or, And, Exclusive-Or et Not seront notés respectivement $+$, $.$, $\oplus$ et $\overline{f}$. L'équivalence de deux fonctions est notée $-$. Les opérateurs n-aires de conjonction et de disjonction seront notés Π et $\sum$.
- Les quantificateurs universel et existentiel seront notés $\exists$ et $\forall$.
- la représentation symbolique d'un ensemble E sera notée $\widehat{E}$. Nous utiliserons cette notation indifféremment pour la fonction caractéristique d'un ensemble et le BDD correspondant.

Dans la suite, nous utilisons les BDDs pour la représentation de modèles. En particulier, nous voulons représenter des ensembles et des relations.

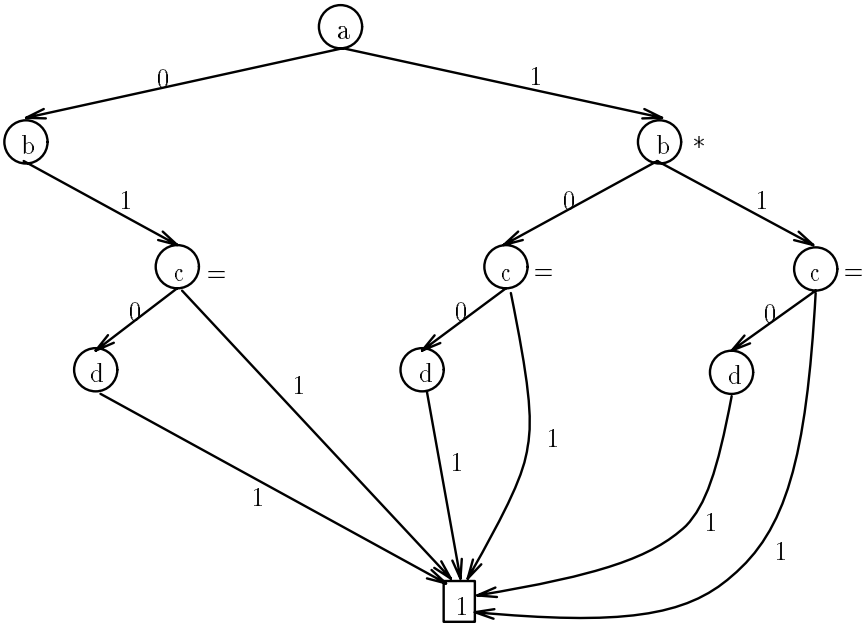


Figure 3.2: En cours de réduction

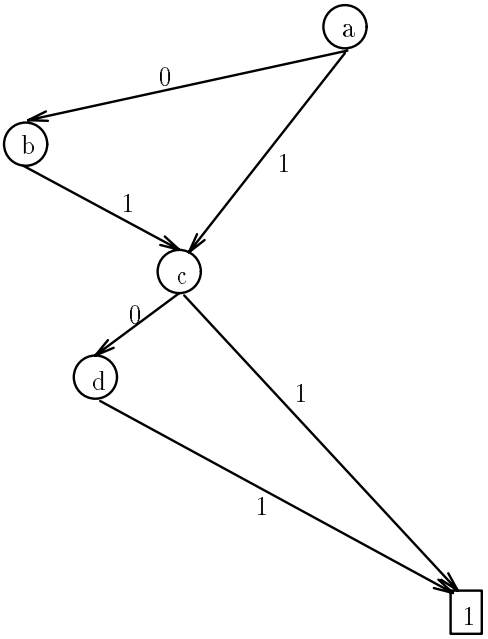


Figure 3.3: Après réduction : un Diagramme de Décision Binaire

3.1.3 Représentation d'un ensemble

Soit Q un ensemble tel que $card(Q) = N$. Nous pouvons représenter un élément de Q par un vecteur de n valeurs booléennes, où $n = \lceil log_2 N \rceil$. Dans la suite, nous noterons $\sigma : Q \rightarrow \{0,1\}^n$ la fonction qui associe à un élément de Q son codage en un vecteur de valeurs booléennes. σ est en général la fonction qui associe à un nombre son codage binaire. Nous noterons σ_i la fonction qui associe à un élément de Q la valeur du i$^{\text{ème}}$ bit de son codage binaire.

La représentation d'un ensemble Q se fait par le biais de sa fonction caractéristique $\widehat{Q}$: $\{0,1\}^n \rightarrow \{0,1\}$ telle que $\forall q \in Q, \widehat{Q}(q) = 1, \forall q \notin Q, \widehat{Q}(q) = 0$. Cette fonction est construite sur un vecteur de variables booléennes $\vec{x} = \{x_1, x_2, \ldots, x_n\}$ que nous appellerons *ensemble support* dans la suite.

Définition 3.1-2 (Fonction caractéristique de Q)

$$\widehat{Q}(\vec{x}) = \sum_{q \in Q} \prod_{1 \leq i \leq n} x_i - \sigma_i(q)$$

■

Les opérations usuelles sur les ensembles peuvent être exprimées comme des opérations booléennes entre leur fonctions caractéristiques :

étant donnés deux ensembles Q_1 et Q_2 et leur fonction caractéristique respective $\widehat{Q_1}$ et $\widehat{Q_2}$, nous avons les définitions suivantes.

$$\begin{aligned} \widehat{\emptyset} &= 0 \\ \widehat{Q_1 \cap Q_2} &= \widehat{Q_1}.\widehat{Q_2} \\ \widehat{Q_1 \cup Q_2} &= \widehat{Q_1} + \widehat{Q_2} \\ \widehat{Q_1 \setminus Q_2} &= \widehat{Q_1}.\overline{\widehat{Q_2}} \end{aligned}$$

Une opération qui nous sera utile par la suite est la construction d'un BDD $\widehat{Q}(\vec{y})$ à partir d'un BDD équivalent $\widehat{Q}(\vec{y})$, c'est-à-dire la *substitution* de l'ensemble support $\vec{x}$ par $\vec{y}$. Nous noterons l'opérateur correspondant $[\vec{x} \rightarrow \vec{y}] : \widehat{Q}(\vec{y}) = \widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}]$. Si les ordres des variables de $\vec{x}$ et $\vec{y}$ sont identiques, alors cette substitution se résume à un renommage des nœuds de $\widehat{Q}$.

3.1.4 Représentation d'une relation

Une relation d'un ensemble Q dans lui-même peut être donnée par l'ensemble des couples d'éléments qu'elle met en relation. De la même manière que pour un élément de Q, nous pouvons représenter un couple d'éléments $(q_1, q_2) \in Q \times Q$ par un vecteur de $2 * n$ valeurs booléennes.

La fonction caractéristique de la relation R sera donc basée sur un ensemble de $2*n$ variables booléennes. Cet ensemble est construit comme l'union de deux ensembles isomorphes de n variables, que nous noterons $\vec{x}$ et $\vec{y}$.

Définition 3.1-3 (Fonction caractéristique d'une relation binaire)

$$\widehat{R}(\vec{x},\vec{y}) = \sum_{(q_1,q_2)\in Q} \prod_{1\leq i\leq n} x_i - \sigma_i(q_1).\prod_{1\leq i\leq n} y_i - \sigma_i(q_2)$$

■

Cette définition s'étend facilement aux relations k-aires.

3.1.5 Image d'un ensemble par une relation

Soit Q un ensemble et $\widehat{Q}$ $(\vec{x})$ le BDD le représentant, le calcul symbolique de l'image de Q par la relation R est donné par la formule suivante :

$$\widehat{R(Q)}(\vec{x}) = (\exists\vec{x}(\widehat{R}(\vec{x},\vec{y}) \wedge \widehat{Q}(\vec{x})))[\vec{y}\rightarrow\vec{x}]$$

Cette définition nous permet d'exprimer les fonctions *pre* et *post*, pour un ensemble d'états Q et une relation de transitions T :

$$\widehat{post}(\widehat{Q})(\vec{x}) = (\exists\vec{x}(\widehat{T}(\vec{x},\vec{y}) \wedge \widehat{Q}(\vec{x})))[\vec{y}\rightarrow\vec{x}]$$

$$\widehat{pre}(\widehat{Q})(\vec{x}) = \exists\vec{y}(\widehat{T}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x}\rightarrow\vec{y}])$$

De la même manière que précédemment, nous omettrons de préciser $(\vec{x})$ quand il n'y a pas d'ambiguité ou de besoin.

3.1.6 Composition de deux relations

Le calcul de la composition de deux relations nécessite l'introduction d'un troisième jeu de variables, que nous noterons $\vec{z}$. Une relation $R = R_1 o R_2$ sera représentée symboliquement par :

$$\widehat{R}(\vec{x},\vec{y}) = \exists\vec{z}(\widehat{R_1}(\vec{x},\vec{y})[\vec{y}\rightarrow\vec{z}] \wedge \widehat{R_2}(\vec{x},\vec{y})[\vec{x}\rightarrow\vec{z}])$$

Cette opération peut se révéler coûteuse, car les BDDs intermédiaires produits pendant le calcul de $\wedge$ portent sur les trois ensembles de variables et peuvent donc être énormes par rapport à la taille du BDD $\widehat{R}$ final.

3.2 Calcul de points fixes

Le calcul d'un plus petit point fixe de la forme $\mu X.(Q \cup post_T(X))$ correspond au calcul de la suite :

$$\begin{cases} \widehat{\varpi_0} = \widehat{Q} \\ \widehat{\varpi_{n+1}} = \widehat{\varpi_n} + \widehat{post}(\widehat{\varpi_n}) \end{cases}$$

Ce calcul s'effectue en largeur d'abord; nous pouvons améliorer ce calcul en ne calculant la fonction *post* que sur les états accédés pour la première fois lors du calcul précédent. Ces

états constituent la *frontière* de l'ensemble des états accessibles.

$$\begin{cases} \widehat{\varpi_0} = \widehat{Q} \\ \widehat{\varpi_1} = \widehat{Q} + \widehat{post}(\widehat{Q}) \\ \widehat{\varpi_{n+1}} = \widehat{\varpi_n} + \widehat{post}(\widehat{\varpi_n}.\overline{\widehat{\varpi_{n-1}}}) \end{cases}$$

Dans la suite, nous noterons $lfp(Q, F)(\vec{x})$ la fonction caractéristique correspondant au résultat d'un plus petit point-fixe $\mu X(Q \cup F(X))$

Remarque 3-1
Le calcul de tous les états atteignables à partir d'un ensemble d'états Q correspond à l'ensemble $lfp(Q(\vec{x}), \widehat{post}(\vec{x}, \vec{y}))(\vec{x})$. Le calcul de l'ensemble des états à partir desquels des états de Q sont accessibles correspond à $lfp(Q(\vec{x}), \widehat{pre}(\vec{x}, \vec{y}))(\vec{x})$. ■

3.2.1 Optimisations pour le calcul de points fixes

Le calcul de point-fixe est un élément central des algorithmes que nous voulons implémenter. Les optimisations qu'il est possible d'apporter dans le cas des BDDs sont de deux natures :

simplification de la relation de transition T : au fur et à mesure du calcul d'un point fixe, il est possible d'agir sur la taille de la représentation de T, en tenant compte des informations déjà connues, en l'occurrence les résultats intermédiaires du calcul en cours. Cette optimisation a déjà été étudiée et utilisée intensivement, et se ramène au problème plus général de *simplification de fonctions booléennes*.

diminution de la profondeur d'itération : l'autre paramètre sur lequel nous pouvons agir est le nombre d'étapes de calcul nécessaires pour obtenir le résultat désiré. Cette optimisation est réalisée par un certain nombre de techniques existantes, qui sont basées sur le calcul d'une relation T^* par divers degrés de composition de la relation T. Nous présentons plus loin deux techniques de calcul de fermeture transitive. Nous présenterons dans le chapitre 4 une technique alternative qui est liée au modèle que nous utilisons.

3.2.2 Simplification de fonctions

L'idée de base de la simplification de fonctions booléennes est la suivante : soit deux fonctions f et g de $\{0,1\}^n$dans $\{0,1\}$; simplifier f connaissant g revient à construire une fonction h équivalente à f sur le domaine de g :

Définition 3.2-1

$$\forall q, g(q) \Rightarrow (h(q) - f(q))$$

■

Traditionnellement, la fonction h est appelée la *restriction* de f par rapport à g. Etant

données deux fonctions f et g, l'ensemble des fonctions h qui sont des restrictions de f par rapport à g est caractérisé par la proposition suivante :

Proposition 3.2-1
Soient f, g et h trois fonctions de $\{0,1\}^n$dans $\{0,1\}$, h est une restriction de f par rapport à g ssi

$$(f \wedge g) \Rightarrow h$$

et

$$h \Rightarrow (\neg g \vee f)$$

■

Preuve

La définition 3.2-1 de la restriction d'une fonction $g \Rightarrow (h - f)$ peut se réécrire :

$$g \Rightarrow ((h \Rightarrow f) \wedge (f \Rightarrow h))$$

ce qui nous donne

$$[(g \Rightarrow (h \Rightarrow f))] \wedge [g \Rightarrow (f \Rightarrow h)]$$

donc

$$[h \Rightarrow (g \Rightarrow f)] \wedge [(g \wedge f) \Rightarrow h]$$

donc

$$[g \Rightarrow h - f] - [h \Rightarrow (\neg g \vee f)] \wedge [(f \wedge g) \Rightarrow h]$$

■

Le problème de la restriction de fonction dans le cas des BDDs revient alors à choisir la fonction h comprise entre $(f \wedge g)$ et $(\neg g \vee f)$ telle que h soit minimale en nombre de nœuds. Néanmoins, la recherche de cette solution optimale est en général exponentielle. En pratique, nous utiliserons certaines heuristiques qui, sans fournir une restriction toujours optimale, fournissent une "bonne" solution; ces heuristiques sont en particulier utilisées par deux opérateurs classiques de restriction pour les BDDs, l'opérateur *constrain ou cofacteur* [CM90] noté $\uparrow$ et l'opérateur *restrict* [CM90] noté $\Uparrow$.

Nous verrons dans le chapitre 4 comment nous pouvons utiliser ces opérateurs pour optimiser le calcul de points fixes avec les BDDs.

3.2.3 Fermeture transitive de la relation de transition

Le calcul de la fermeture transitive d'une relation de transition est couramment utilisé dans les outils de calcul d'équivalence. Il permet notamment de ramener le calcul d'équivalence pour des bisimulations faibles comme la bisimulation observationnelle au calcul de l'équivalence pour la bisimulation forte sur une *forme pré normale* [Fer88]. Un autre intérêt est la possibilité de diminuer le coût de certains points-fixes calculés fréquemment.

Etant donné une relation de transition T, l'idée est de calculer la suite :

$$T_1 = T$$

$T_{i+1} = ToT_i \cup T_i$

jusquà convergence. On notera alors $T^* = T^n(T^n = T^{n-1})$ la limite de cette suite.

Le calcul d'un plus petit point fixe est donné par $T^*(\bot)$.

3.2.4 Iterative squaring

L'*Iterative squaring* est une technique mise au point par [BCM+90]. Elle permet d'effectuer le calcul de la fermeture transitive de manière plus efficace que la manière précédente.

Le calcul de points fixes d'une fonction F se fait en deux étapes :

1. Calculer la suite $F, F^2, \ldots, F^{2^p}, \ldots$ jusquà ce que $F^{2^{p+1}} = F^{2^p}$
2. le calcul du plus petit point fixe sera donné par $F^{2^p}(\bot)$ et celui du plus grand point fixe par $F^{2^p}(\top)$

La fonction F^{2^p} coïncide avec la fonction $Post_{T^*}$, où T^* est la fermeture transitive de la relation de transition T. Du point de vue complexité du calcul, si n est la longueur de la plus longue chaîne présente dans T, nous obtenons $p = \lceil log_2(n) \rceil$.

3.3 Modèle symbolique à base de BDDs

Nous avons présenté les Diagrammes de Décision Binaires et une méthode classique de représentations d'ensembles et de relations avec les BDDs. Nous avons d'autre part présenté certaines techniques classiques pour l'optimisation du calcul de points fixes avec les BDDs. Ces optimisations reposent soit sur la diminution de la taille de la représentation de la relation utilisée dans le calcul (opérateurs de restriction), soit sur la diminution du nombre d'itérations du calcul nécessaires pour l'obtention du point fixe (fermeture transitive de la relation). A partir de ces définitions assez générales, nous allons maintenant donner les méthodes de construction d'un modèle symbolique, à partir du modèle Réseau de Petri présenté au chapitre 1 .

Chapitre 4

Modèle symbolique avec les BDDs

Nous avons défini dans le chapitre précédent l'ensemble des méthodes de codage et des opérateurs sur les BDDs dont nous avons besoin. Nous pouvons maintenant nous intéresser à la représentation d'un Réseau de Petri *sauf* sous la forme d'un ensemble de BDDs. Plus précisément, nous allons construire une représentation symbolique de la relation de transition du graphe d'états lui correspondant.

La définition d'un codage du réseau passe par le choix d'ensembles de variables booléennes, les *ensembles supports*. Ces variables sont celles qui permettront de construire des fonctions booléennes qui seront représentées par des BDDs. Nous commençons par donner des critères de choix de ces ensembles, puis nous donnons une méthode de construction de la représentation en BDDs de chaque unité du réseau, ainsi que certaines optimisations liées à des propriétés du réseau. Nous présentons ensuite deux méthodes de construction de la représentation symbolique de tout le modèle, à partir des représentations des unités. La deuxième partie importante dans une utilisation efficace des BDDs est le choix d'un ordre des variables des ensembles supports. Nous isolons certains critères et définissons une méthode permettant de choisir un "bon" ordre, pas obligatoirement optimal.

4.1 Définition des ensembles supports

Dans la suite, nous considérons un réseau $(\mathcal{Q}, \mathcal{U}_l, \mathcal{T}, \mathcal{G}, \emptyset)$ composé d'une hiérarchie de u unités, chaque unité U_i ayant ses places propres numérotées de 1 à n_i ($n_i = card(places(U_i))$). Nous noterons $n = card(\mathcal{Q})$ le nombre de places du réseau ($n = \sum_{i=0}^{u} n_i$).

4.1.1 Marquages

Nous associons à un ensemble de marquages M sa fonction caractéristique $\widehat{M}$. Pour optimiser le choix de l'ensemble support correspondant, et notamment diminuer le nombre de variables utilisées, nous allons tirer parti des propriétés particulières du Réseau de Petri que nous utilisons.

Chaque méthode que nous présentons ici tire partie de la propriété correspondante, mais aussi des propriétés des méthodes de rang inférieur.

Méthode 1 : Marquage sauf
La propriété du marquage sauf nous donne une borne supérieure sur le nombre de variables booléennes nécessaires pour définir $\widehat{M}$. Comme il y a au plus une marque par place, il suffit de n variables booléennes pour un réseau de n places.

La structuration en unités du réseau nous apporte des informations supplémentaires dont nous pouvons tirer parti pour améliorer cette représentation.

Méthode 2 : Séquentialité des unités
Chaque unité contient au plus une marque dans ses places propres. Il suffit alors de construire pour chaque unité U_i la fonction caractéristique de l'ensemble de ses places propres. Nous noterons $\vec{B_i}$ l'ensemble support de la fonction caractéristique d'une unité U_i. Pour chaque unité qui n'est pas une unité de base, nous devons modéliser la présence ou l'absence de marque dans ses marques propres. Cette information peut être codée de deux manières différentes :

- En ajoutant un élément dans l'ensemble à représenter; dans ce cas, l'ensemble support $\vec{B_i}$ est tel que $\vec{B_i} = \{b_i^1, \ldots, b_i^b\}$ avec $b = \lceil log_2(n_i + 1) \rceil$
- En ajoutant une variable booléenne dans l'ensemble support; dans ce cas, $\vec{B_i} = \{a_i, b_i^1, \ldots, b_i^b\}$ avec $b = \lceil log_2(n_i) \rceil$

Même si elle impose l'introduction systématique d'une nouvelle variable, la deuxième solution est préférable dans le cas des BDDs, car elle permettra des optimisations globales ayant une influence importante sur la taille des BDDs créés.

Méthode 3 : Imbrication des unités
Pour chaque unité contenant une marque, soit la marque est dans les places propres de l'unité, soit certaines de ses sous-unités contient une marque (les autres sont inactives). Il est donc possible de réutiliser les variables de l'ensemble support associé aux places propres d'un unité pour les fonctions caractéristiques de ces sous-unités. Si on considère la relation d'imbrication des unités comme une arborescence, le nombre maximal d'unités actives en même temps est donné par les feuilles de cette arborescence, qui correspondent aux unités de base. Chacune de ces unités ne contient plus que des places propres, nous pouvons donc facilement définir un ensemble support minimal leur correspondant. Ces unités pouvant éventuellement être actives en même temps, il est nécessaire de définir un ensemble support particulier à chacune.

Pour associer à chaque unité U_i un ensemble support utilisé pour les places propres de U_i et pour les sous unités de U_i, nous partons donc des unités de base et synthétisons les ensembles support suivant les règles suivantes :

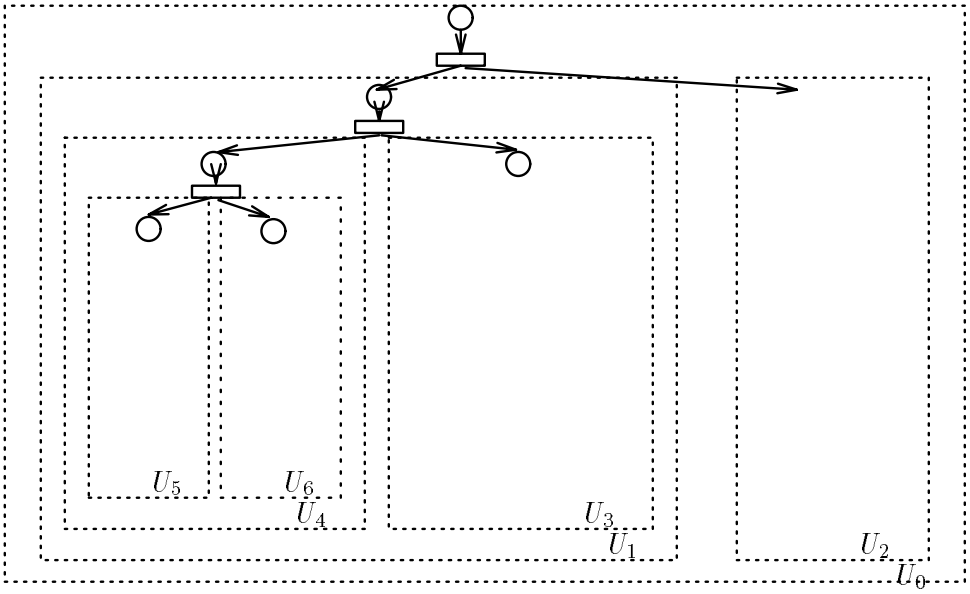


Figure 4.1: Unités d'un Réseau de Petri

$$\frac{U_i = (\widetilde{Q}_i, Q_{0_i}, \widetilde{U}_i), \widetilde{U}_i = \emptyset}{\vec{B}_i = \{b_i^1, \ldots, b_i^b\}, b = \lceil log_2(n_i) \rceil} \quad [B1]$$

$$\frac{U_i = (\widetilde{Q}_i, Q_{0_i}, \widetilde{U}_i), \widetilde{U}_i \neq \emptyset}{\vec{B}_i = \bigcup_{U_j \in \widetilde{U}_i} B_j \cup \{a_i, b_i^1, \ldots, b_i^l\}, l = \lceil log_2(n_i) \rceil - \sum_{U_j \in \widetilde{U}_i} card(B_j)} \quad [B2]$$

L'ensemble $\{b_i^1, \ldots, b_i^l\}$ de la deuxième règle est l'ensemble des variables supplémentaires nécessaires si l'ensemble des places propres de U_i est trop grand pour être codé sur l'union des ensembles de variables des sous unités de U_i. Comme dans la méthode de codage précédente, la variable a_i permet d'indiquer l'activité de l'unité U_i.

Exemple 4-1 (Ensembles supports pour les marquages)
Soit un Réseau de Petri $(\mathcal{Q}, \mathcal{U}_l, \mathcal{T}, \mathcal{G}, \mathcal{V})$ *composé de 7 unités* $\{\mathcal{U}_l, U_1, \ldots, U_6\}$ *imbriquées suivant la figure 4.1 :*

Le nombre de places propres de chaque unité est donné par le vecteur suivant :

$$[1, 2, 5, 3, 7, 2, 2]$$

pour un nombre total de places de 22. Les unités de base sont les unités U_5, U_6, U_3, U_2. *La méthode 1 nécessite alors un ensemble support de 22 variables booléennes.*

$$\vec{x} = \{b_1, \ldots b_{22}\}$$

La méthode 2 associe à chaque unité les ensembles supports suivants :

$B_0 = \{b_0^1\}$
$B_1 = \{b_1^1\}$

$$
\begin{aligned}
B_2 &= \{b_2^1, b_2^2, b_2^3\} \\
B_3 &= \{b_3^1, b_3^2\} \\
B_4 &= \{b_4^1, b_4^2, b_4^3\} \\
B_5 &= \{b_5^1\} \\
B_6 &= \{b_6^1\} \\
\vec{x} &= \bigcup i = 1 \ldots 6 B_i \cup \{a_0, a_1, a_4\}
\end{aligned}
$$

soit un total de 12 variables plus 3 variables a_i.

Si nous intégrons une valeur supplémentaire pour représenter l'absence de marques, nous avons alors

$$
\begin{aligned}
B_0 &= \{b_0^1\} \\
B_1 &= \{b_1^1, b_1^2\} \\
B_2 &= \{b_2^1, b_2^2, b_2^3\} \\
B_3 &= \{b_3^1, b_3^2\} \\
B_4 &= \{b_4^1, b_4^2, b_4^3\} \\
B_5 &= \{b_5^1, b_5^2\} \\
B_6 &= \{b_6^1, b_6^2\} \\
\vec{x} &= \bigcup i = 1 \ldots 6 B_i
\end{aligned}
$$

soit un total de 15 variables.

Enfin, la méthode 3 nous donne les ensembles supports suivants :

$$
\begin{aligned}
B_0 &= B_1 \cup B_2 \\
B_1 &= B_3 \cup B_4 \\
B_2 &= \{b_2^1, b_2^2, b_2^3\} \\
B_3 &= \{b_3^1, b_3^2\} \\
B_4 &= \{b_4^1\} \cup B_5 \cup B_6 \\
B_5 &= \{b_5^1\} \\
B_6 &= \{b_6^1\} \\
\vec{x} &= B_0 \cup \{a_0, a_1, a_4\}
\end{aligned}
$$

pour un total de 8 variables plus 3 variables a_i. ■

En pratique, la méthode 2 s'avère suffisante. En effet, une unité U contenant des sous unités a souvent un ensemble de places propres réduit à une place (qui est la source d'une transition lançant chaque sous unité en parallèle). Par conséquent, la méthode 2 et la méthode 3 donnent des résultats équivalents, puisqu'il faut ajouter une variable a_U indiquant si l'unité est active ou non. Cette variable peut alors être directement utilisée pour indiquer la présence ou l'absence de marque dans la seule place de U.

4.1.2 Contextes

La représentation d'un contexte C se fait par la construction de sa fonction caractéristique $\widehat{C}$. Une difficulté particulière liée à notre modèle est la présence de variables entières. Pour pouvoir construire une représentation de ces variables, nous nous limiterons aux variables de domaine fini.

De la même manière que pour la représentation des marquages, nous allons associer à l'ensemble des variables de $\mathcal{V}$ un ensemble de variables booléennes $\vec{x}_c$ qui serviront de support aux fonctions caractéristiques. La construction de l'ensemble $\vec{x}_c$ dépend du domaine des variables de $\mathcal{V}$; la fonction caractéristique d'une variable entière V est construite par rapport à son domaine D_V. L'ensemble support correspondant à une variable V est donc constitué de $\lceil log_2(\mid D_V \mid)\rceil$ variables booléennes. Nous noterons $\vec{x}_C^V$ l'ensemble support de la variable V.

L'ensemble support d'un contexte est défini comme l'union des ensembles supports de toutes les variables :

$$\vec{x}_c = \bigcup_{V \in \mathcal{V}_{Bool}} \{x_V\} \cup \bigcup_{V \in \mathcal{V}_{Int}} \{x_V^1, x_V^2, \ldots x_V^{\lceil log_2(|D_V|)\rceil}\}$$

4.1.3 Etats

Etant donnés les ensembles supports nécessaires pour la représentation des marquages et des contextes, nous pouvons construire l'ensemble support $\vec{x}$ pour les états :

$$\vec{x} = \vec{x}_m \cup \vec{x}_c$$

Notations
La fonction surchargée $s()$ nous permet d'associer à une unité ou une variable son ensemble support particulier :
soit U une unité,

$$s_U(\vec{x}_m) = \vec{x}_U \quad \text{tel que } \vec{x}_U \subseteq \vec{x}_m$$

Soit $V \in \mathcal{V}$,

$$s_V(\vec{x}_c) = \vec{x}_V \quad \text{tel que } \vec{x}_V \subseteq \vec{x}_c$$

Pour faciliter la représentation des marquages, nous noterons $Idle_U$ le prédicat qui indique qu'aucune place propre de U n'est marquée. $\widehat{Idle_U}(\vec{x})$ sera alors la fonction caractéristique de ce prédicat, codée sur l'ensemble de variables booléennes $\vec{x}$.

Remarque 4-1
Dans le cas des ensembles supports contenant une variable a_i indiquant l'activité d'une unité U_i, nous avons $Idle_{U_i}(\vec{x}) = \overline{a}_i$. ■

De la même manière, nous définissons un prédicat $Busy_U$ qui indique si une unité contient un marquage. La définition de la fonction caractéristique de ce prédicat va dépendre de la définition des ensembles supports des marquages :

Méthodes 1 et 2

$$\widehat{Busy}_U(\vec{x}) = \overline{\widehat{Idle}_U(\vec{x})}$$

Méthode 3

$$\widehat{Busy}_U(\vec{x}) = \overline{\widehat{Idle}_U}(\vec{x}) + \sum_{U' \in Units^*(U)} \widehat{Busy}_{U'}$$

Nous utiliserons l'opérateur binaire $\triangleright$ de *projection* tel que si m est un marquage et U une unité, alors :

$$\begin{cases} m \triangleright U = \{p\} \text{ si } \exists p \in Places(U), p \in m \\ \\ m \triangleright U = \emptyset \text{ sinon} \end{cases}$$

Nous étendons cet opérateur aux transitions du réseau : Soit t une transition et U une unité,

$$t \triangleright U = (\bullet t \triangleright U, t\bullet \triangleright U)$$

Dans la même idée, nous noterons `Affect` la fonction qui associe à une transition t l'ensemble des variables de $\mathcal{V}$ qui apparaissent en partie gauche d'une affectation de l'action de t.

Nous noterons `Use` la fonction qui associe à une transition t l'ensemble des variables de $\mathcal{V}$ qui apparaissent dans les expressions relationnelles et en partie droite des affectations de l'action de t. L'ensemble $\texttt{Use}(t) \cup \texttt{Affect}(t)$ contient toutes les variables qui apparaissent dans l'action de t.

4.2 Représentation symbolique

Maintenant que les ensembles supports sont définis, nous pouvons construire la fonction caractéristique des ensembles d'états que nous voulons représenter.

4.2.1 Représentation des marquages

Le codage des places et des marquages est défini comme suit :

Définition 4.2-1 (Codage d'une place)
Soit p une place de l'unité U et $s_U(\vec{x}_m) = \{x1, \ldots, xu\}$,

$$\widehat{p}(\vec{x}_m) = \prod_{k=1}^{u} x_{i_k} - \sigma_k(p)$$

■

Définition 4.2-2 (Codage d'un marquage)

$$\widehat{m}(\vec{x}_m) = \prod_{p \in m} \widehat{p}(s_U(\vec{x}_m)). \prod_{U, m \triangleright U = \emptyset} Idle_U$$

■

La fonction caractéristique d'un ensemble de marquages M est alors donné par la définition suivante :

Définition 4.2-3 (Fonction caractéristique d'un ensemble de marquages)

$$\widehat{M}(\vec{x}) \quad = \quad \sum_{m \in M} \widehat{m}(\vec{x})$$

■

4.2.2 Représentation des contextes

La fonction caractéristique d'un ensemble I de valeurs d'une variable V est donnée sur l'ensemble support $\vec{x}_C^V = \{x_1, \ldots, x_n\}$ par l'expression suivante :

$$\widehat{I}(\vec{x}_C^V) = \sum_{v \in I} \prod_{1 \leq i \leq n} x_i - \sigma_i(v)$$

Comme dans la section 3.1.3, la fonction $\sigma : D_V \rightarrow \{0,1\}^n$ associe à une valeur du domaine D_V son codage binaire.

4.2.3 Représentation des expressions sur les variables

Les actions des transitions peuvent contenir des *gardes* et des *affectations* définies sur les variables du modèles. Ces gardes et affectations sont données par des expressions de valeur sur les variables, dont nous séparons la représentation suivant le type des variables impliquées.

Expression de valeur booléenne

La construction de la fonction caractéristique $\widehat{E}$ d'une expression de valeur booléenne E se construit suivant les règles suivantes :

$$\widehat{E}(\vec{x}_c) = \begin{cases} \mathbf{0} & \textbf{si} \quad E \equiv \texttt{false} \\ \mathbf{1} & \textbf{si} \quad E \equiv \texttt{true} \\ s_V(\vec{x}_m) & \textbf{si} \quad E \equiv v \\ \widehat{E_1}(\vec{x}_c) & \textbf{si} \quad E \equiv \textbf{not} \;\; E_1 \\ \widehat{E_1}(\vec{x}_c) + \widehat{E_2}(\vec{x}_c) & \textbf{si} \quad E \equiv E_1 \;\textbf{or}\; E_2 \\ \widehat{E_1}(\vec{x}_c).\widehat{E_2}(\vec{x}_c) & \textbf{si} \quad E \equiv E_1 \;\textbf{and}\; E_2 \\ \widehat{E_1}(\vec{x}_c) - \widehat{E_2}(\vec{x}_c) & \textbf{si} \quad E \equiv E_1 = E_2 \end{cases}$$

Expression de valeur entière

Nous avons vu comment définir la fonction caractéristique d'une variable V de domaine D_V donné. Pour représenter les expressions de valeur définis sur les entiers, nous pouvons utiliser une des méthodes suivantes :

- Par construction d'une formule en termes d'opérateurs booléens pour chaque opérateur sur les entiers. Cette méthode permet de traiter de manière générale toute expression

de valeur basée sur des entiers. Si cette correspondance est simple à définir pour les opérateurs $=$, $\neq$, ceci devient plus difficile pour des opérateurs comme $>$ et $+$.

- Par construction de la fonction caractéristique de *chaque* expression de valeur utilisée dans le système à modéliser. Cette méthode de construction est beaucoup moins générale, puisque pour chaque expression de valeur, nous construirons une fonction caractéristique ad-hoc. Nous montrons d'abord comment construire cette fonction pour une expression relationnelle. Soit R une relation entre n variables entières, la fonction caractéristique de R est donnée par les expressions :

$$\widehat{R}(\vec{x}_c) = \sum_{(v_1,\ldots,v_n)\in R} \prod_{1\leq i\leq n} \widehat{\{v_i\}}(\vec{x}_c^{V_i})$$

  $\widehat{\{v_i\}}$ est la fonction caractéristique du singleton contenant v_i et $\vec{x}_c^V$ est le sous ensemble de $\vec{x}_c$ qui sert d'ensemble support à la variable V. La construction de cette fonction consiste alors à énumérer tous les éléments de R, et à construire une fonction caractéristique de chacun de ces éléments. La somme de ces fonctions nous donne alors la fonction caractéristique de R.

  La représentation d'une affectation de la forme $x_i := f(x_1, \ldots, x_n)$ se fait de manière similaire, en considérant la relation $R(x'_i, x_1, \ldots, x_n)$ et en prenant pour la construction de l'ensemble caractéristique de R l'ensemble $\vec{y}_c^{v_i}$ comme ensemble support pour la variable x'_i. L'ensemble support $\vec{y}_c$ est défini plus loin, pour le codage d'une transition.

Cette méthode de représentation des expressions de valeur entières est celle utilisée dans une bibliothèque existante [Kam92], connue sous le nom de *Diagrammes de Décision Multivalués* (MDD). Le terme multivalué ne correspond pas dans le cas de cette bibliothèque à l'utilisation d'arbre de décision n-aires, puisque cette bibliothèque est basée sur une bibliothèque de BDD. Des fonctions pour construire des relations unaires, binaires et ternaires entre variables entières sont intégrées. De plus, des méthodes de construction optimisées (i.e. ne passant pas par l'énumération de tous les éléments de la relation) sont utilisées, notamment pour la construction de comparaison de variables, ou de relations faisant intervenir une constante. Les fonctions définies pour cette bibliothèque sont aisément transposables à notre bibliothèque particulière de BDDs.

4.2.4 Représentation des transitions

A partir de la représentation des marquages et contextes, et des expressions de valeur, nous pouvons maintenant définir la représentation symbolique des transitions du réseau. Nous commençons par définir cette représentation du point de vue transformations de marquages (relation $\longrightarrow_m$), puis transformations de contextes (relation $\longrightarrow_c$). Puis nous combinons ces résultats pour construire la représentation de la relation de transition globale du réseau.

Transitions de $\longrightarrow_m$
Soit une transition $t = (\widetilde{Q_i}, \widetilde{Q_o}, G, \widetilde{O}, A)$, nous notons t_m la partie de cette transition qui

définit les transformations de marquages. t_m définit une relation dont le graphe est l'ensemble $\{(M_1, M_2) \in \mathcal{M} \mid {}^\bullet t \subseteq M_1, M_2 = M_1 - {}^\bullet t \cup t^\bullet\}$.

La représentation de la fonction caractéristique d'une relation nécessite la duplication des ensembles supports. Nous notons $\vec{x}_m$ l'union des ensembles supports utilisées pour le codage des marquages d'entrée d'une transition et $\vec{y}_m$ l'union des ensembles support liés aux marquages de sortie.

La fonction caractéristique de cette relation est alors construite à partir des fonctions suivantes :

$$\begin{aligned}
\widehat{t_\downarrow}(\vec{x}_m, \vec{y}_m) &= \prod_{U \in \widetilde{U}, t \triangleright U = (\{p\}, \{p'\})} \widehat{p}(\vec{x}_m).\widehat{p'}(\vec{y}_m) \\
\widehat{t_\triangle}(\vec{x}_m, \vec{y}_m) &= \prod_{U \in \widetilde{U}, t \triangleright U = (\{p\}, \emptyset)} \widehat{p}(\vec{x}_m).Idle_U(\vec{y}_m) \\
\widehat{t_\nabla}(\vec{x}_m, \vec{y}_m) &= \prod_{U \in \widetilde{U}, t \triangleright U = (\emptyset, \{p'\})} Idle_U(\vec{x}_m).\widehat{p'}(\vec{y}_m).
\end{aligned}$$

La fonction $\widehat{t_\downarrow}$ correspond aux transformations de marquages internes aux places propres d'une unité. $\widehat{t_\triangle}$ correspond dans une unité au passage de la marque d'une place propre aux sous unités. $\widehat{t_\nabla}$ correspond à la désactivation de sous unités d'une unité (et éventuellement au passage de la marque dans les marques propres de l'unité). La fonction $\widehat{t_m}$ est alors égale à la conjonction de ces trois fonctions.

Définition 4.2-4 (Codage d'une transition)
$\widehat{t_m}(\vec{x}_m, \vec{y}_m) = \widehat{t_\downarrow}(\vec{x}_m, \vec{y}_m).\widehat{t_\triangle}(\vec{x}_m, \vec{y}_m).\widehat{t_\nabla}(\vec{x}_m, \vec{y}_m)$

■

Transitions de $\longrightarrow_c$
Soient $\vec{x}_c$ et $\vec{y}_c$ les ensembles supports associés aux contextes, la fonction caractéristique d'une action A est alors construite comme suit :

$$\widehat{A}(\vec{x}_c, \vec{y}_c) = \begin{cases} \widehat{E}(\vec{x}_c) & \textbf{si} \quad A \equiv \textbf{when } E \\ \widehat{v}(\vec{y}_c) - \widehat{E}(\vec{x}_c) & \textbf{si} \quad A \equiv v := E \\ \widehat{A_1}(\vec{x}_c, \vec{y}_c).\widehat{A_2}(\vec{x}_c, \vec{y}_c) & \textbf{si} \quad A \equiv A_1 \ ; A_2 \end{cases}$$

Pour une transition t, nous noterons $\widehat{t_c}(\vec{x}_c, \vec{y}_x)$ la fonction caractéristique de la transformation de contexte correspondante. Si A est l'action de t, on a $\widehat{t_c}(\vec{x}_c, \vec{y}_c) = \widehat{A}(\vec{x}_c, \vec{y}_c)$.

Représentation des transitions de $\longrightarrow$
Etant donnée une transition t du réseau, sa fonction caractéristique est construite sur les ensembles support $\vec{x}$ et $\vec{y}$, tels que $\vec{x} = \vec{x}_m \cup \vec{x}_c$ et $\vec{y} = \vec{y}_m \cup \vec{y}_c$. La fonction caractéristique de t est alors égale à :

$$\widehat{t}(\vec{x}, \vec{y}) = \widehat{t_m}(\vec{x}_m, \vec{y}_m).\widehat{t_c}(\vec{x}_c, \vec{y}_c)$$

La fonction $\widehat{t_m}$ telle qu'elle est construite ici n'impose pas de contraintes sur les unités dont le marquage n'est pas modifié par t. Cette absence de contraintes se traduit par l'apparition

de tous les couples de marquages possibles pour ces unités.

Pour modéliser le fait que le marquage de certaines unités doit rester inchangé, ou que certaines variables ne changent pas de valeur, nous définissons la fonction caractéristique $\widehat{Stable}$ de la relation identité.

Définition 4.2-5 (Stabilité)
Soit $\vec{x} = \{x_1, \ldots, x_n\}$ *et* $\vec{y}\{y_1, \ldots, y_n\}$

$$\widehat{Stable}(\vec{x}, \vec{y}) = \prod_{i \in [1 \ldots n]} (x_k - y_k)$$

■

Remarque 4-2
La définition de $\widehat{Stable}$ utilise le fait que l'ordre relatif des variables de $\vec{x}$ et $\vec{y}$ est le même. ■

La stabilité d'une unité U peut alors se représenter par le BDD $\widehat{Stable}(s_U(\vec{x}_m), s_U(\vec{y}_m))$. De même, la stabilité d'une variable v est représentée par le BDD $\widehat{Stable}(s_v(\vec{x}_c), s_v(\vec{y}_c))$.

Pour une transition t, nous pouvons alors construire la fonction t_{stable} qui caractérise le comportement de toutes les unités et toutes les variables qui ne sont pas modifiées par t. Cette fonction est donnée par l'expression suivante :

$$\begin{aligned} \widehat{t_{stable}}(\vec{x}, \vec{y}) \quad = & \\ & \prod_{t \triangleright U = (\emptyset, \emptyset)} Stable(s_U(\vec{x}_m), s_U(\vec{y}_m) \\ \wedge & \\ & \prod_{v \in (\mathcal{V} \setminus \mathtt{Use}(t) \cup \mathtt{Affect}(t))} Stable(s_v(\vec{x}_c), s_v(\vec{y}_c) \end{aligned}$$

Nous verrons comment cette fonction est utilisée lors de la construction de la relation de transition globale. En particulier, cette utilisation va dépendre du mode de composition utilisé.

4.3 Représentation de la relation de transition globale

Nous avons défini la fonction caractéristique de chaque transition du réseau. Nous pouvons maintenant construire la fonction caractéristique de la relation de transition globale. Comme cette relation de transition est étiquetée, nous devons faire apparaître les étiquettes des transitions dans la représentation symbolique. Plutôt que de coder les étiquettes dans les BDDs représentant les transitions, comme décrit dans [EFT91] , nous avons choisi comme [Bou93] de partitionner la représentation de la relation de transition globale en fonction des étiquettes. Dans la suite, nous définissons comment la représentation symbolique de la relation de transition pour *une* étiquette est construite. Puis nous détaillons comment le calcul de la relation de transition globale (ou des fonctions pre et post associées) est fait.

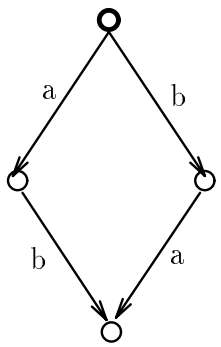


Figure 4.2: Composition par entrelacement

4.3.1 Composition entrelacée

La méthode de construction de la relation de transition globale habituellement utilisée correspond à la sémantique de l'entrelacement : l'expression de l'exécution possible en parallèle de deux actions a et b est représentée par le losange de la figure 4.2.

Ceci correspond en fait à n'autoriser qu'une action globale à la fois. Pour chaque transition t du réseau, on impose que toute variable non modifiée et toute unité U active, mais non impliquée dans la transition reste stable.

La relation de transition correspondant à une étiquette $a \in A$ est donnée par la définition suivante :

Définition 4.3-1 (Relation de transition pour une étiquette)

Pour chaque $a \in A$, nous définissons $\widehat{T_a}$ tel que :

$$\widehat{T_a}(\vec{x},\vec{y}) = \sum_{\xi(t)=a} \widehat{t_s}(\vec{x},\vec{y})$$

où

$$\widehat{t_s}(\vec{x},\vec{y}) = \widehat{t}(\vec{x},\vec{y}).\widehat{t_{stable}}(\vec{x},\vec{y})$$

■

La relation de transition globale, sans considérations d'étiquettes est alors :

$$\widehat{T}(\vec{x},\vec{y}) = \sum_{a \in A} \widehat{t_a}(\vec{x},\vec{y})$$

4.3.2 Composition simultanée

Un certain nombre de travaux [BCL91, McM92] proposent une alternative à la composition classique que nous avons utilisée. L'idée de base est de permettre à plusieurs composantes parallèles d'effectuer *simultanément* une action asynchrone. Cette idée peut être étendue à l'exécution simultanée d'actions qui synchronisent deux ensembles de composantes parallèles,

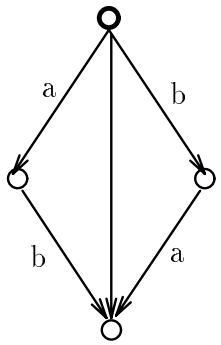


Figure 4.3: Composition simultanée

pourvu que ces ensembles soient disjoints.

Intuitivement, cette idée correspond au dessin 4.3 : dans le système de transition correspondant à deux actions entrelacées, on *ajoute* la transition permettant l'exécution simultanée des deux actions.

Cette idée semble aller a contrario des méthodes habituellement utilisées pour diminuer l'impact du problème de l'explosion des états. En effet, la composition simultanée va encore *ajouter* des transitions, donc faire grossir la relation de transition, alors que des méthodes comme l'utilisation d'ordres partiels cherchent à *éliminer* certaines transitions. Néanmoins, cette forme alternative de composition possède certains avantages :

- Elle va permettre, à l'instar de la *fermeture transitive* ou de l'*iterative squaring* de diminuer le nombre d'itérations dans le calcul d'un point fixe.
- Comme nous utilisons une représentation symbolique de la relation de transition, l'augmentation du nombre de transitions ne se traduit pas automatiquement par une augmentation de la taille de la représentation.

Dans le cas des Réseaux de Petri, ceci correspond à autoriser l'exécution simultanée de plusieurs transitions du réseau. Mais pour que deux transitions puissent s'exécuter simultanément, elles ne doivent pas provoquer de modifications de marquages dans une même unité.

Pour pouvoir déterminer quelles transitions sont composables simultanément, nous définissons une relation de *compatibilité* entre les différentes transitions :

Définition 4.3-2 (Relation de compatibilité)

Soient deux transitions t_1 *et* $t_2 \in \mathcal{T}$

$$t_1 \mathcal{R}_{comp} t_2 \quad - \quad \forall U, \quad (t_1 \triangleright U \neq (\emptyset,\emptyset)) - (t_2 \triangleright U = (\emptyset,\emptyset)) \wedge (\texttt{Use}(t_1) \cap \texttt{Affect}(t_2) = \emptyset) \wedge$$

$$(\texttt{Use}(t_2) \cap \texttt{Affect}(t_1) = \emptyset)$$

■

Cette relation indique que l'intersection entre les ensembles d'unités synchronisées par ces transitions est vide, et qu'il n'y a pas de relation de causalité entre l'affectation d'une variable et son utilisation. Cette relation est symétrique, mais n'est ni réflexive, ni transitive.

Ensembles maximaux de transitions compatibles

On appelle *ensemble de transitions compatibles* un ensemble $Comp$ tel que :

$$Comp = \{t \in \mathcal{T} \mid \forall t, t' \in Comp, t \neq t' \Rightarrow t \mathcal{R}_{comp} t'\}$$

Un ensemble *maximal* d'actions compatibles sera tel que $\forall t \notin Comp, \exists t' \in Comp : t \not\mathcal{R}_{comp} t'$.

Un ensemble maximal de transitions compatibles correspond à une *clique*(sous graphe complet) du graphe de la relation $\mathcal{R}_{comp}$. La recherche de tous les ensembles maximaux de transitions compatibles revient au calcul de toutes les cliques sur ce graphe. En pratique, nous nous limiterons à la recherche d'un *recouvrement* de ce graphe par un ensemble de cliques.

Il peut arriver que la composition simultanée de trop d'actions en même temps provoque un accroissement important de la taille de la représentation de la relation. Dans ce cas, il est intéressant de disposer d'intermédiaires entre la composition entrelacée et la composition simultanée complète. Nous proposons les deux solutions intermédiaires suivantes :

Composition simultanée *limitée*
: Une méthode simple pour le calcul des ensembles T_{Comp} est de limiter leur taille à une certaine constante k, c'est-à-dire de limiter le nombre maximum de d'actions pouvant être composées simultanément.

Composition mixte
: Une autre méthode pour simplifier la composition simultanée est de ne l'appliquer que pour les transitions *asynchrones*, i.e. les transitions portant sur les places propres d'*une et une seule* unité. Les autres transitions (transitions synchrones) sont alors composées par entrelacement. Nous désignerons par la suite ce mode de composition sous le nom de *composition mixte*.

4.3.3 Calcul de la relation de transition

La relation de transition globale pour la composition simultanée se calcule par rapport à la partition $Comps$ du graphe G en ensembles de transitions compatibles. Pour chaque $Comp \in Comps$, on définit $\widehat{T}_{Comp}$ de la manière suivante :

$$\widehat{T}_{Comp} = \prod_{t \in Comp} (\widehat{t} + \widehat{t_{stable}})$$

Pour chaque transition t qui synchronise un ensemble $\widetilde{U}$ d'unités, on donne la possibilité aux unités de $\widetilde{U}$ soit de bouger toutes ensembles, soit de rester stables. Le développement du produit $\prod_{t \in Comp}(\ldots)$ nous donne alors toutes les combinaisons possibles d'exécution simultanée des transitions de $Comp$.

On définit la relation de transition globale obtenue par composition simultanée comme étant la réunion des différents $\widehat{T}_{Comp}$:

$$\widehat{T_{\Diamond}} = \sum_{Comp \in Comps} \widehat{T}_{Comp}$$

Remarque 4-3
L'utilisation de la composition simultanée ne permet plus un partitionnement de la relation de transition par rapport aux étiquettes. En effet, la notion d'étiquette n'a plus le même sens, puisqu'un élément de T_{Comp} peut correspondre à l'exécution simultanée de plusieurs actions. Cette disparition des étiquettes ne pose pas de problèmes pour les applications que nous faisons de la composition simultanée. ■

4.3.4 Application de la composition simultanée

Calcul de l'ensemble des états accessibles
Le calcul des états accessibles du modèle est une application directe de la composition simultanée. Dans ce cas, la disparition des étiquettes(remarque 4-3) ne pose pas de problèmes particuliers.

Calcul des successeurs et prédécesseurs par τ^*
Le calcul des successeurs d'un ensemble d'états Q par les transitions étiquetées τ se réduit au calcul du plus petit point fixe $\mu X.(Q \cup post_{T_\tau}(X))$. De la même manière, le calcul des prédécesseurs d'un ensemble d'états par la relation de transition T_τ se réduit au plus petit point fixe $\mu X.(Q \cup pre_{T_\tau}(X))$.

La composition simultanée peut donc être appliquée pour le calcul des fonctions $pre_{\tau^* a}$ et $post_{\tau^* a}$. Ces fonctions jouent un rôle important dans l'algorithme de génération de modèle minimal, en particulier quand la relation de bisimulation considérée est la $\tau^* a$ bisimulation. C'est en général la fonction la plus coûteuse, le nombre de transitions étiquetées τ dépassant souvent largement le nombre de transitions visibles. Par l'application de la composition simultanée dans ce contexte, nous voulons diminuer au maximum l'impact de ce calcul.

Dans le cas de la bisimulation de branchement, la composition simultanée des τ n'est pas utilisable, comme le montre l'exemple de la figure 4.4. Le système de transitions étiquetées (2) correspond au système de transitions étiquetées (1) augmenté d'une transition qui correspond à la composition simultanée des transitions τ. L'ajout de cette transition dans le système de transitions étiquetées b rend les deux systèmes non équivalents pour la bisimulation de branchement.

Figure 4.4: Contre exemple pour la bisimulation de branchement

	Composition entrelacée ($\widehat{T_\diamond}$)	Fermeture Transitive ($\widehat{T_\diamond}^*$)	Iterative squaring ($\widehat{T_\diamond}^*$)	Composition simultanée ($\widehat{T_\Diamond}$)
Construction de T	C_e	C_e	C_e	C_s
Pré traitement	0	L	$\lceil log_2(L) \rceil$	0
Calcul d'un point fixe ($T^*(\perp)$)	L	1	1	$max(l_i) \leq$ $\leq L$

4.3.5 Comparaisons

L'avantage principal de la composition simultanée est la diminution du nombre d'itérations lors du calcul d'un point fixe. Il est donc important de comparer cette technique avec la méthode de fermeture transitive de la relation exposée dans le chapitre 3

Nous avons distingué comme points de comparaison la construction de la relation T, un éventuel pré traitement pour calculer T^* et le calcul du point fixe. Les coûts de construction de T sont notés C_e(composition entrelacée) et C_s (composition simultanée). Les coûts des calculs sont donnés en nombre de composition de relations pour le pré traitement et d'itérations pour le point fixe. Nous notons L la longueur de la plus longue chaîne présente dans T, et l_i la longueur de la plus longue chaîne présente dans T_i, T_i étant la relation de transition de la composante i du système (unité U_i dans le cas des Réseaux de Petri). Le nombre d'itérations pour le calcul d'un point fixe dans le cas de la composition simultanée est donné entre deux extrêmes : le premier correspond au cas d'un système totalement asynchrone, le deuxième à un système totalement synchrone. Dans ce dernier cas, la composition simultanée se ramène à la composition entrelacée.

D'après cette table, la composition simultanée parait plus intéressante (pour le calcul de point fixe) que la composition entrelacée, mais moins intéressante que l'iterative squaring. La différence majeure entre ces deux techniques se situe au niveau du pré traitement, nécessaire à l'itérative squaring. Ce pré traitement nécessite de calculer la composition de deux relations. Le calcul de la composition de relations nécessite l'introduction d'un troisième jeu de variables (voir chapitre 3), ce qui en fait en général un calcul très coûteux. D'autre part, $T_\diamond \subseteq T_\Diamond \subseteq T_\diamond{}^*$, et les tailles des BDDs représentant ces relations ont tendance à suivre cette ordre. En particulier, la taille de $\widehat{T_\diamond}^*$ est en général prohibitive.

Dans ce cadre, la composition simultanée offre une alternative intéressante, puisqu'elle per-

met une diminution du nombre d'itérations dans le calcul des points fixes (cette diminution dépendant du degré d'asynchronisme du système), moyennant une augmentation en général peu gênante de la taille des BDDs. Des comparaisons de performances entre chaque méthode sont donnés dans le chapitre 8.

4.3.6 Quantification avancée

Pour chacun de ces modes de composition, la relation de transition globale est donnée sous une forme *disjonctive*. Cette forme disjonctive va permettre d'améliorer facilement le calcul des fonctions *pre* et *post* avec les BDDs : en effet, la fonction *pre* se calcule par la formule suivante 3 :

$$\widehat{pre}(\widehat{Q})(\vec{x}) \quad = \quad \exists(\vec{y})(\widehat{T}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}]))$$

Comme $\widehat{T}$ s'exprime sous la forme d'une disjonction de sous formules, et que le quantificateur $\exists$ est distributif pour l'opérateur $\vee$, nous allons pouvoir partitionner ce calcul pour chacune des sous formules de T. Ce partitionnement va être différent suivant la méthode de composition utilisée :

Composition entrelacée

La relation de transition $T_\diamond$ est déjà partitionnée pour chaque étiquette. Nous pouvons continuer ce partitionnement par rapport à T_a.

$$\widehat{T_a} \quad = \quad \sum_{\xi(t_s)=a} \widehat{t_s}$$

donc

$$\begin{aligned}
\widehat{pre_a}(\widehat{Q})(\vec{x}) &= \exists(\vec{y})(\widehat{T_a}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}])) \\
&= \exists(\vec{y}) \sum_{\xi(t_s)=a} (\widehat{t_s}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}])) \\
&= \sum_{\xi(t_s)=a} \exists(\vec{y})(\widehat{t_s}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}]))
\end{aligned}$$

Le calcul de pre_a peut donc se partitionner en une somme de calculs de *pre* pour chaque t_s étiqueté a.

Composition simultanée

$$\widehat{T_\Diamond} \quad = \quad \sum_{Comp \in Comps} \widehat{T_{Comp}}$$

donc

$$\begin{aligned}
\widehat{pre}(\widehat{Q})(\vec{x}) &= \exists(\vec{y})(\widehat{T_\Diamond}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}])) \\
&= \sum_{Comp \in Comps} \exists(\vec{y})(\widehat{T_{Comp}}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}])) \\
&= \exists(\vec{y}) \sum_{Comp \in Comps} (\widehat{T_{Comp}}(\vec{x},\vec{y}) \wedge (\widehat{Q}(\vec{x})[\vec{x} \rightarrow \vec{y}]))
\end{aligned}$$

Le calcul de *pre* peut se partitionner par rapport à chaque ensemble de transitions compatibles.

Ce partitionnement du calcul permet de transformer un calcul portant sur deux BDDs ($\widehat{T}$ et $\widehat{Q}$) de taille importante en un ensemble de calculs où un opérande est de taille très inférieure, évitant ainsi la production de BDDs intermédiaires de taille prohibitive. Le partitionnement permet donc de traiter certains exemples qui ne passe pas habituellement par défaut de mémoire. Par contre, l'expérience a montré que du point de vue temps d'exécution, il est préférable de regrouper au maximum les relations de transitions, pour limiter le nombre d'opérations $\exists$ qui est une opération coûteuse sur les BDDs.

4.3.7 Applications des opérateurs de restriction

Nous séparons l'application des opérateurs de restriction en deux catégories : une méthode *statique* où l'ensemble de restriction est calculé, et les restrictions par rapport à cet ensemble sont effectuées une fois pour toutes ; une méthode *dynamique* où l'ensemble de restriction, et les restrictions résultantes sont appliquées au fur et à mesure des calculs.

Méthode statique

L'expérience montre que les algorithmes de comparaison de modèles et de minimisation de modèles se comportent mieux si on restreint les calculs à l'ensemble des états accessibles. En pratique, le comportement d'un modèle tend à être plus régulier dans sa partie accessible; en particulier, le degré de non déterminisme des états inaccessibles tend à être largement supérieur au degré de non déterminisme dans les états accessibles. Le véritable sur-coût se situe au niveau du nombre d'itérations d'un calcul de point fixe. Comme ce calcul s'effectue en largeur d'abord, le nombre d'itérations est borné par la taille de la plus longue chaîne. Lorsqu'on considère les états inaccessibles, cette chaîne peut être de longueur largement supérieure à la plus longue chaîne existante dans les états accessibles.

Une solution, qui à l'expérience se montre efficace, est de faire précéder l'application des algorithmes de vérification par un calcul des états accessibles. La connaissance des états accessibles va permettre d'effectuer des restrictions à deux niveaux :

- Dans le cas d'un algorithme comme la génération de modèle minimal, la partition initiale est restreinte à l'ensemble des états accessibles. En dehors des améliorations éventuelles des performances de calcul des points fixes, cette restriction permet aussi la simplification de l'algorithme lui-même.
- Une étape supplémentaire consiste à restreindre directement les BDDs de $\widehat{T}$, pour qu'ils ne représentent plus que des transitions entre états accessibles. L'expérience montre néanmoins que cette deuxième restriction peut provoquer un accroissement conséquent de la taille des BDDs.

Méthode dynamique

Une opération intéressante est la restriction de T *pendant* le calcul d'un point fixe. Comme chaque itération du calcul d'un point fixe se fait par rapport aux états "frontières"du calcul en largeur d'abord (chapitre 3), il est possible de restreindre T de deux manières :

- Restriction par rapport à l'ensemble des états pas encore accédés. Dans ce cas, $\widehat{T}$ est restreint à $\overline{Reach}$, où $Reach$ est l'ensemble des états déjà obtenus.
- Reconstruction de $\widehat{T}$ par rapport aux états frontières (nouveaux états accédés lors de la dernière itération . Comme la relation $\widehat{T}$ est une disjonction de plusieurs sous formules, il est possible de réévaluer $\widehat{T}$ à chaque étape, par restriction de ces sous formules à l'ensemble des états frontières.

Utilisation en pratique

En pratique, nous appliquons systématiquement la première étape de l'approche statique, en particulier pour l'algorithme de génération de modèle minimal. Des comparaisons de performances ont montré que ceci apportait toujours une amélioration dans les temps de calcul, cette amélioration étant très souvent spectaculaire. Un exemple extrême est le cas du Scheduler de Milner (voir chapitre 8) où l'algorithme de génération de modèle minimal sans calcul préalable des états accessibles a des performances à peine meilleures que des méthodes énumératives classiques (limitation à un système d'un peu plus de 10 cyclers, environ 100000 états). Le calcul des états accessibles, et la limitation de la partition initiale à ces états, a permis de passer à des systèmes d'une centaine de cyclers (environ 10^{30} états).

Remarque 4-4
L'utilisation de cette restriction statique est surtout efficace pour les calculs en arrière : les calculs en avant se font en général à partir d'états dont l'accessibilité est connue, donc on ne finit pas par explorer des parties inaccessibles du modèle. Par contre, les calculs en arrière provoquent systématiquement l'exploration d'états inaccessibles. Il est alors important de pouvoir restreindre les calculs en cours aux seul états accessibles. ■

La deuxième étape de l'approche statique et les deux méthodes dynamiques sont basées sur une restriction de la relation de transition. L'expérience a montré que très souvent, la relation de transition avait une représentation plus compacte si nous n'opérions pas de restriction. Il n'est donc pas systématiquement intéressant d'utiliser ces modes de restriction. L'application se fera alors au choix de l'utilisateur, en fonction de l'exemple.

4.4 Ordre des variables

Le choix de l'ordre des variables support est crucial pour une utilisation efficace des BDDs. En effet, la taille d'un BDD est extrêmement sensible à l'ordre choisi. Malheureusement, le choix d'un ordre optimal est un problème NP-difficile. Il existe un algorithme [FS87] qui

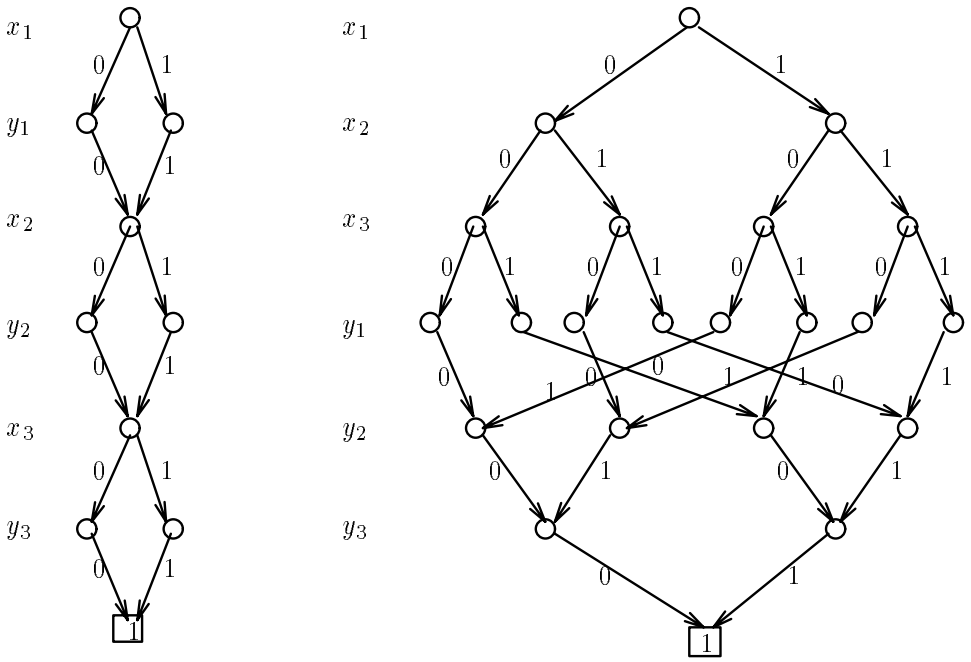


Figure 4.5: BDD de la fonction *Stable*

calcule un ordre optimal des variables pour la représentation d'une fonction donnée, mais sa complexité $O(n^2 3^n)$ (n : nombre de variables) le rend inintéressant en pratique.

Le choix d'un ordre se fait habituellement par la définition d'heuristiques qui sont liées à la classe de fonctions à représenter.

Les choix d'ordre que nous avons faits sont basés sur l'heuristique générale suivante, qui consiste à essayer de rapprocher au maximum dans l'ordre choisi deux variables booléennes en relation l'une avec en l'autre. Cette heuristique se justifie par les arguments suivants; considérons deux variables x_i et x_j avec $i < j$ dans l'ordre des variables d'un BDD, et x_i et x_j sont en relation l'une avec l'autre (par exemple, elle vérifie $x_i = \overline{x_j}$). Dans ce cas, il faudra forcément introduire le nœud x_j sur tous les chemins du graphe de décision partant du nœud x_i. Le nombre de ces chemins va dépendre en partie du nombre de nœuds et donc du nombre de variables qui sont intercalées entre x_i et x_j. Si $j - i = 1$, il n'y aura que deux nœuds x_j (ce qui est le minimum). Sinon, si $j - i = d$, alors nous pouvons avoir jusqu'à 2^{d+1} nœuds x_j.

L'influence d'une relation existante entre variables et la taille du BDD correspondant est illustrée par l'exemple de la figure 4.5 :

Exemple 4-2

Chacun des deux BDDs représente la fonction booléenne $(x_1 - y_1) \wedge (x_2 - y_2) \wedge (x_3 - y_3)$. Le premier BDD est construit sur un ordre entrelacé des variables : $x_1 < y_1 < x_2 < y_2 < x_3 < y_3$. Sa taille en nombre de nœuds est donné par la formule $3 * n$ où n est le nombre de x_i (ici

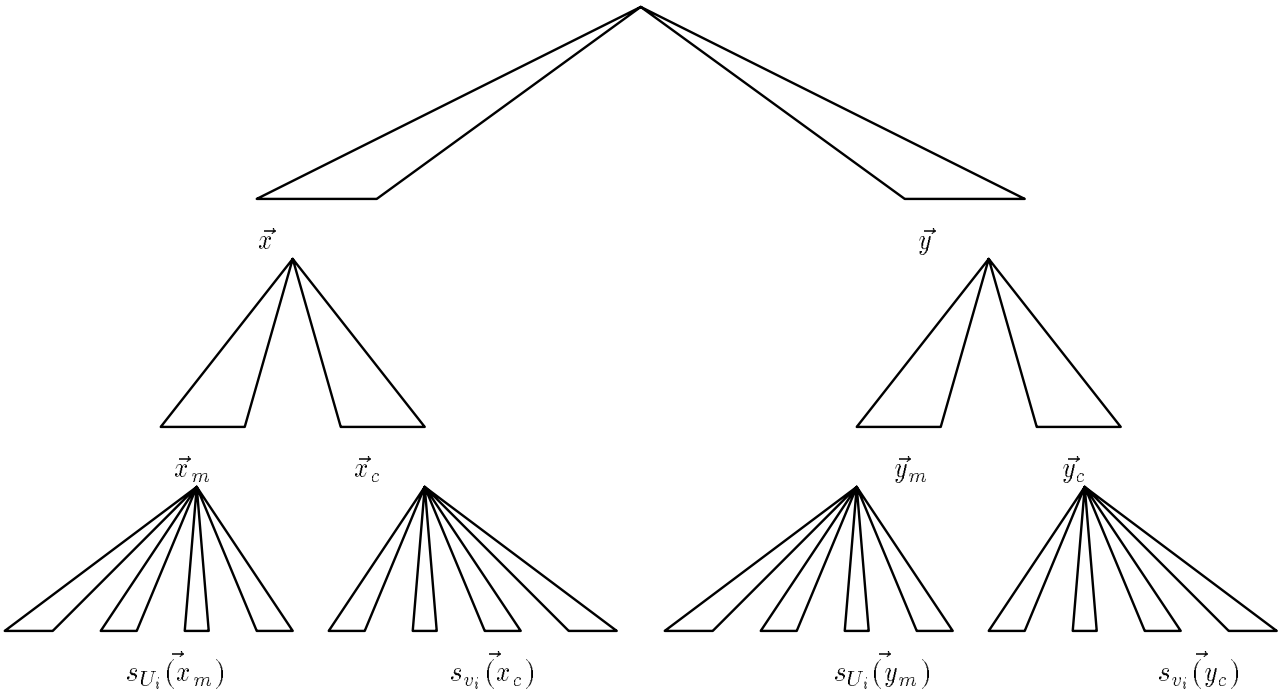


Figure 4.6: Décomposition des ensembles supports

n = 3). Le deuxième BDD code la même fonction avec l'ordre $x_1 < x_2 < x_3 < y_1 < y_2 < y_3$. Sa taille en nombre de nœuds est alors donné par la formule $3 * (2^n - 1)$. Pour une même formule, nous passons donc à un codage donnant un BDD de taille linéaire par rapport à la taille de la formule, à un codage exponentiel.

En particulier, nous pouvons voir que le nombre de variables intercalées entre x_1 et y_1 est de 2 dans le deuxiéme BDD. Nous avons alors $2^3 = 8$ nœuds y_1. ■

4.4.1 Décomposition des ensembles supports

Le codage de la relation de transition repose sur la définition des ensembles supports $\vec{x}$ et $\vec{y}$. Chacun de ces ensembles est construit à partir des ensembles supports définis pour les unités et les variables, suivant la hiérarchie donnée par l'arbre de la figure 4.6.

A chaque nœud de cet arbre, nous associons un ensemble support qui sera représenté par une liste ordonnée de variables booléennes.

Par conséquent, nous avons à déterminer pour chaque feuille de l'arbre un ordre interne qui va permettre de construire cette liste de variables. Puis pour chaque nœud, nous devons décider d'un ordre qui permette d'effectuer l'interclassement des listes des fils de ce nœud.

Nous pouvons dès maintenant décider de l'ordre d'interclassement de $\vec{x}_m$ et $\vec{y}_m$, en tenant compte de l'exemple précédent (figure 4.5). En effet, les méthodes de composition que nous

avons définies font un usage intensif de la fonction $Stable$. L'ordre d'interclassement de $\vec{x}_m$ et $\vec{y}_m$ le plus indiqué est donc l'ordre entrelacé.

De la même manière, nous pouvons décider de l'ordre d'interclassement de $\vec{x}_c$ et $\vec{y}_c$. En effet, la majorité des transitions laisse les variables inchangées, ce qui s'exprime de nouveau par l'introduction de $Stable(\vec{x}_c, \vec{y}_c)$. Nous pouvons de la même manière choisir l'ordre entrelacé comme ordre d'interclassement de $\vec{x}_c$ et $\vec{y}_c$.

Ces deux choix nous donnent alors un ordre entrelacé pour $\vec{x}$ et $\vec{y}$. Il nous reste maintenant à définir un ordre interne à $\vec{x}$, l'ordre interne de $\vec{y}$ étant le même que celui de $\vec{x}$. Pour cela, nous allons d'abord définir les ordres internes respectifs de $\vec{x}_m$ et $\vec{x}_c$.

4.4.2 Ordre interne de $\vec{x}_m$

Nous voulons définir un ordre pour les variables de l'ensemble support $\vec{x}_m$ utilisé pour la représentation des marquages du réseau. Pour cela, nous allons d'abord définir l'ordre interne des feuilles correspondantes, c'est à dire l'ordre interne des $s_U(\vec{x}_m)$ pour chaque unité U.

Ordre interne aux unités

Le codage d'une unité est lié à la numérotation de ses places. Cette numérotation est en général aléatoire et il parait difficile de définir une heuristique satisfaisante quelle que soit la relation de transition à représenter. De plus, même si un bon ordre interne pouvait être construit pour les unités, nous n'avons aucune garantie que cet ordre ne se révèle pas mauvais pour l'ensemble support de $\vec{x}_m$.

Le seul choix que nous ayons fait à ce niveau est de placer en première variable dans l'ordre celle qui correspond au bit de poids fort de la décomposition binaire des numéros de place.

Ordre d'interclassement des unités

L'ordre interne de $\vec{x}_m$ se construit à partir des ordres internes définis pour chaque $s_U()$. Le premier choix que nous avons effectué est de ne pas faire d'interclassement entre les variables de deux $s_{U_i}(\vec{x}_m)$ différents. La définition de l'ordre global se réduit alors à l'ordonnancement des ensembles $s_U(\vec{x}_m)$ entre eux.

Le choix de cet ordre va dépendre des synchronisations qui existent entre les différentes unités. En effet, quand deux unités sont synchronisées, il y a création d'une relation entre les marquages de chacune de ces deux unités. Cette relation entre marquages se traduit par une relation entre les variables des ensembles support de ces unités . Une optimisation de l'ordre des variables est d'essayer de rapprocher les ensembles de variables de ces deux unités dans l'ordre global :

Exemple 4-3

Considérons un réseau composé de 4 unités U_1, U_2, U_3, U_4 en parallèle, telles que les seules transitions synchrones sont t_1 et t_2, avec t_1 qui synchronise U_1 et U_3 et t_2 synchronise

U_2 et U_4. Si on note $s_{U_1}()$ l'ensemble support de chaque U_i, un ordre tenant compte des synchronisations sera $s_{U_1}() << s_{U_3}() << s_{U_2}() << s_{U_4}()$. ■

De manière plus générale, l'ordre relatif des ensembles supports sera choisi par la méthode suivante :
Soit un Réseau de Petri $(\mathcal{Q}, \mathcal{U}_l, \mathcal{T}, \mathcal{G}, \mathcal{V})$, considérons le graphe $G = (V, E)$ avec

- $V = \mathcal{U}_l \cup Units^*(\mathcal{U}_l)$
- E est l'ensemble des arcs (U_i, U_j) tels que $\exists t \in \mathcal{T}, \bullet t \rhd U_i \neq \emptyset \wedge \bullet t \rhd U_j \neq \emptyset$ (t synchronise les unités U_i et U_j).

Nous voulons associer à ce graphe une fonction de numérotation n des sommets qui nous donnera l'ordre relatif des ensembles supports des unités. Pour cela, nous associons à chaque arc $v = (e_i, e_j)$ de G un poids suivant la formule $w(v) = abs(n(e_i) - n(e_j))$.

Le choix d'un ordre minimal se fera en déterminant la fonction de numérotation n telle que la valeur de $\sum_{v=(e_i,e_j)\in V}(abs(n(e_i) - n(e_j)))$ soit minimale. Pour calculer un ordre raisonnable, mais pas forcément optimal, nous allons utiliser un algorithme de tri topologique des sommets du graphe G. L'ordre topologique obtenu nous donne alors l'ordre relatif des ensembles supports $s_{U_i}()$.

4.4.3 Ordre interne de $\vec{x}_c$

De la même manière que pour les unités, l'ordre d'interclassement des ensembles supports pour les variables de $\mathcal{V}$ dépend des relations existantes entre ces variables. En effet, si deux variables x et y sont comparées, alors il est préférable que leurs ensembles supports soient proches dans l'ordre global. Comme dans le cas précédent, nous construisons un graphe reflétant les dépendances entre variables du réseau.

4.4.4 Ordre d'interclassement de $\vec{x}_m$ et $\vec{x}_c$

Nous pouvons considérer deux options pour les ordres relatifs de $\vec{x}_m$ et $\vec{x}_c$:

- Mêler les variables de $\vec{x}_m$ et celles de $\vec{x}_c$. Dans ce cas, il reste à déterminer comment les mêler. Une solution classique consiste alors à les entrelacer.
- Ordonner les deux ensembles séparément, par exemple en mettant toutes les variables de $\vec{x}_m$ avant celles de $\vec{x}_c$.

La première solution consiste à essayer d'optimiser l'ordre, en tenant compte d'éventuelles relations entre le codage des données et le codage du contrôle. Ceci peut être le cas dans notre modèle, si nous considérons que certaines variables sont locales à certaines unités (toute utilisation ou affectation d'un variable se fait lors d'une transition propre à une même unité). Dans ce cas, pour une unité donnée, il peut être intéressant de rapprocher dans l'ordre de $\vec{x}_c$ l'ensemble support des variables de l'ensemble support des places de l'unité.

La deuxième solution correspond à considérer toutes les variables comme globales, et à n'optimiser l'ordre des variables BDD que pour les relations existantes entre variables du réseau. C'est la solution que nous avons actuellement adoptée, essentiellement pour des raisons de simplicité.

Le choix de l'ordre relatif de $\vec{x}_m$ et $\vec{x}_c$ nous donne l'ordre de $\vec{x}$. Nous avons ainsi défini la construction des ensembles supports pour la représentations des Réseaux de Petri en utilisant quelques critères de choix d'un "bon ordre" pour ces ensembles. Ces critères sont évidemment liés à notre modèle, et peuvent de plus se révéler mauvais pour certains exemples. Néanmoins, l'expérimentation a montré qu'en général, les choix que nous avons fait donnaient des résultats satisfaisants.

4.5 Conclusion

Nous avons défini une méthode de construction d'un modèle symbolique à partir du modèle Réseau de Petri que nous utilisons. Nous avons utilisé une méthode de codage du contrôle en BDDs qui est maintenant classique [EFT91, Bou93]. Nous avons étendu cette méthode pour la représentation de la partie donnée du réseau, et notamment la représentation de variables entières. Pour que cette représentation soit possible, nous devons évidemment nous limiter à des variables bornées. Dans le cas de protocoles de communication, cette restriction semble raisonnable, puisque les variables entières servent soit à identifier des composants (numéro d'une station, identification d'un jeton,...), soit à représenter la taille de buffers, qui sont bornés dans des implémentations réalistes de protocoles.

Une étape supplémentaire serait de passer à la représentation de types de données structurés comme des piles, des tableaux, des enregistrements,... Cette approche existe déjà dans des outils de vérification jusque là dédiés à la vérification de circuits, comme l'outil EVER du système de vérification MURφ[ADAY92] .

Chapitre 5

Application des BDDs

Maintenant que nous avons défini un certain nombre d'opérateurs classiques (le calcul des successeurs d'un ensemble d'états, le calcul de points fixes, ...) en terme d'opérations sur les BDDs, nous pouvons passer à des applications plus orientées vers la vérification. Nous allons d'abord nous intéresser à des application simples qui tournent autour du calcul des états accessibles du modèle. Puis nous nous intéresserons à des applications plus orientées vers la vérification, en particulier l'algorithme de génération de modèle minimal que avons présenté au chapitre 2. Nous présentons en fin une méthode souvent utilisée dans le cas des BDDs pour la comparaison de modèles, par exploration du *produit synchrone* de ces modèles.

5.1 Etats accessibles

Le calcul de l'ensemble des états accessibles $Acc(Q)$ correspond au plus petit point-fixe $\mu X.(\{q_{init}\} \cup post(X))$.

La caractérisation des états accessibles d'un modèle est un problème important en pratique, notamment pour la détection d'états indésirés. En particulier, l'algorithme de calcul des états accessibles peut lui-même être modifié pour la recherche d'états particuliers. Ce calcul sera aussi utilisé par la suite pour restreindre la complexité de certains calculs, notamment les calculs de points fixes en arrière.

Dans la suite, nous présentons quelles sont les classes d'états particuliers que nous avons cherché à détecter durant les calculs d'accessibilité.

5.1.1 Etats puits

La recherche d'états puits est un problème classique de la vérification de protocoles. En effet, un grand pourcentage des erreurs dans la conception d'un protocole se traduisent par un blocage indésiré du protocole (rendez-vous avec un partenaire toujours absent, garde jamais vérifiée, ...). Ces blocages s'expriment par la présence d'états puits dans le modèle correspondant.

L'ensemble des puits du modèle est l'ensemble $Sink = \{q \in Q \mid (\forall q' \in Q, q \notin pre_T(q')\}$. L'ensemble des puits *accessibles* est alors obtenu en faisant l'intersection de $Sink$ avec l'ensemble des états accessibles. La formule caractéristique correspondante de l'ensemble des états puits accessibles est $Sink(\vec{x}) = Acc(Q)(\vec{x}) \wedge \overline{\widehat{pre}(Q(\vec{x}))}$.

5.1.2 Etats de divergence

Le calcul des états de divergence est important en pratique, notamment pour la décision de bisimulations telles que la bisimulation de branchement sensible à la divergence. Ici, nous nous intéressons d'abord au simple calcul du prédicat $Div(q)$ qui indique pour un état $q \in Q$ s'il fait partie d'une composante fortement connexe de transitions étiquetées τ. Le calcul de ce prédicat peut se faire de manière uniquement ensembliste, en largeur d'abord et correspond au calcul du plus grand point fixe $\nu X.(Q \cap (pre_\tau(X) \cap post_\tau(X)))$. Ce calcul peut être séparé en deux calculs distincts : le calcul du plus grand point fixe $L_1 = \nu X.(Q \cap (pre_\tau(X))$ et du plus grand point fixe $L_2 = \nu X.(Q \cap (post_\tau(X))$. Le prédicat Div sera alors égal à $L_1 \cap L_2$.

5.2 Génération de Modèle Minimal

Une application particulièrement intéressante est la mise en œuvre de l'algorithme de génération de modèle minimal. Nous savons déjà construire une représentation symbolique de notre modèle. En particulier, nous savons définir les fonctions *pre* et *post* nécessaires à l'algorithme. Pour compléter cette implémentation, il suffit de construire une représentation de la partition ρ et en particulier des classes d'équivalence. Nous associons à chaque classe un BDD qui représente l'ensemble des états de la classe. Toutes les opérations entre classes se ramène alors à des opérations ensemblistes, en termes d'opérateurs BDDs (voir 3).

Nous avons réalisé une implémentation de cet algorithme dans le but de permettre la *minimisation* d'un modèle, mais aussi sa *comparaison* avec un autre. Cette implémentation a permis de mettre en avant une caractéristique particulière de l'utilisation des BDDs comme représentation symbolique : l'algorithme de génération de modèle minimal permet d'effectuer le calcul de l'accessibilité *pendant* le calcul du raffinement de partition. L'expérimentation avec les BDDs a confirmé qu'il est préférable de calculer *a priori* l'ensemble des états accessibles, et d'effectuer le raffinement de partition sur cet ensemble.

Le calcul a priori de cet ensemble a plusieurs avantages :

- la relation de transition peut parfois être simplifiée par application de l'opérateur *Restrict* avec l'ensemble des états accessibles, permettant éventuellement d'obtenir des BDDs plus petits. Néanmoins, l'expérience a montré que cette diminution de la taille des BDDs n'est pas systématique, l'opérateur de restriction pouvant même compliquer et faire grossir les BDDs de la relation de transition.

- comme la partition initiale est définie sur l'ensemble des états accessibles, il n'est plus nécessaire de créer et de mettre à jour des structures spéciales pour conserver

des informations sur l'accessibilité des classes. En fait, l'algorithme de Génération de Modèle Minimal peut être ramené à un classique algorithme de *raffinement de partition*.

Malgré cette considération, notre implémentation actuelle permet les deux modes d'enchaînement de calculs.

5.3 Comparaison de systèmes

Pour effectuer la comparaison de deux systèmes, nous pouvons procéder de deux manières différentes :

- Par raffinement de partition sur l'*union* des deux systèmes. Si les états initiaux des deux systèmes sont à un moment du calcul séparés dans deux classes différentes, les deux systèmes sont déclarés non équivalents et le raffinement est arrêté. Si le raffinement de partition se termine correctement (par stabilisation de la partition), alors les deux systèmes sont déclarés équivalents.
- Par exploration de leur *produit synchrone*. Cette approche est fréquemment utilisée pour la vérification de circuits [CBM89]. Dans [Bou93], une adaptation de cette méthode est présentée pour le calcul de bisimulation entre deux systèmes non déterministes. L'exploration d'un produit des systèmes à comparer est aussi à la base d'algorithmes "à la volée" [Mou92, JJ91]

Raffinement de partition

Pour effectuer la comparaison de systèmes avec l'algorithme de génération de modèle minimal, il suffit de :

- Construire l'union des deux systèmes comme modèle de départ. Pour représenter l'union de ces deux systèmes par des BDDs, il n'est pas nécessaire (et d'ailleurs pas souhaitable) de dissocier les ensembles de valeurs booléennes avec lesquels sont construites les représentations de S_1 et S_2. On prendra alors un ensemble de variables booléennes permettant de représenter le plus gros des deux systèmes. Cet ensemble sera alors utilisé comme support pour chacun des systèmes; on ajoutera à cet ensemble une variable qui permettra d'indiquer quel système est actif.
- Modifier l'algorithme de raffinement de partition, par ajout d'une condition d'arrêt. Cette nouvelle condition d'arrêt intervient lors du raffinement de la classe contenant les états initiaux des deux systèmes : si un raffinement productif de la classe initiale est tel que les états initiaux de S_1 et S_2 sont séparés dans deux classes différentes, alors le raffinement de partition est interrompu et les deux systèmes sont déclarés non équivalents. Si le raffinement de partition se poursuit jusqu'à stabilisation, alors les deux systèmes sont déclarés équivalents.

Produit synchrone

Dans notre cas particulier, l'utilisation de cette méthode va nous permettre d'étendre notre gamme de relations de comparaison, puisqu'elle permet de prendre en compte les *préordres de simulation*. En pratique, l'intérêt des préordres est important :

- Ils permettent de décider de l'*inclusion* d'un système dans un autre et constituent donc un critère de comparaison plus faible, mais souvent bien adapté à la vérification de propriétés de *sûreté*.
- Ils sont en général moins coûteux à calculer que la relation de bisimulation correspondante.

5.3.1 Produit synchrone pour la comparaison

Etant donné deux systèmes S_1 et S_2, nous définissons le produit synchrone comme suit :

Définition 5.3-1 (Le produit synchrone $S_1 \times_\Lambda S_2$)
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées. On note $S_1 \times_\Lambda S_2$, le système de transitions étiquetées $S = (Q, A, T, (q_{01}, q_{02}))$ défini par :

- $Q_{\times_\Lambda} \subseteq (Q_1 \times Q_2)$
- $A \subseteq A_1 \cap A_2$
- $T \subseteq Q \times A \times Q$

et T et Q sont les plus petits ensembles obtenus par application des règles suivantes :

$$(q_{01}, q_{02}) \in Q_{\times_\Lambda} \qquad [R0]$$

$$\frac{(q_1, q_2) \in Q_{\times_\Lambda},\ \lambda \in \Lambda,\ q_1 \xrightarrow{\lambda}_{T_1} q_1',\ q_2 \xrightarrow{\lambda}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q_{\times_\Lambda}, \{(q_1, q_2) \xrightarrow{\lambda}_{T} (q_1', q_2')\} \in T} \qquad [R1]$$

■

Nous allons nous limiter à un cas particulier, i.e. nous considérons qu'un des deux systèmes à comparer est déterministe par rapport au langage Λ. Ce cas de figure, qui est celui dans lequel beaucoup d'outils de vérification de circuits sont placés, permet de définir un critère d'équivalence plus simple :

Proposition 5.3-1 (Cas déterministe)
Soient $S_i = (Q_i, A_i, T_i, q_{0i})_{(i=1,2)}$ deux systèmes de transitions étiquetées, tels que S_1 ou S_2 soit déterministe. *Pour toute équivalence de bisimulation $\sim_\Lambda$, on a :*

$$\forall k \geq 1\ p \sim_\Lambda^k q - (\mathcal{A}ct_\Lambda(p) = \mathcal{A}ct_\Lambda(q) \wedge \forall p'\ \forall q'\ p \xrightarrow{\lambda}_{T_1} p' \wedge q \xrightarrow{\lambda}_{T_2} q' \Rightarrow p' \sim_\Lambda^{k-1} q')$$

■

Cette proposition montre qu'il suffit de découvrir dans le produit synchrone des états n'ayant pas les mêmes possibilités d'évolution ($\mathcal{A}ct_\Lambda(p) \neq \mathcal{A}ct_\Lambda(q)$) pour pouvoir déclarer les systèmes non équivalents.

Nous définissons l'ensemble $Fail$ de tous les états du produit synchrone qui impliquent la non équivalence :

$$\begin{aligned} Fail &= \{(q_1,q_2) \in Q_{\times_\Lambda} \mid \exists (q_1',q_2') \in Q_{\times_\Lambda}, \exists \lambda \in \Lambda, (q_1 \xrightarrow{\lambda}_{T_1} q_1') \not\sim (q_2 \xrightarrow{\lambda}_{T_2} q_2')\} \\ &= \{(q_1,q_2) \in Q_{\times_\Lambda} \mid \mathcal{A}ct_\Lambda(q_1) \neq \mathcal{A}ct_\Lambda(q_2)\} \end{aligned}$$

La condition $Fail = \emptyset$ est une condition suffisante pour l'équivalence de deux systèmes. Si un des deux systèmes est déterministe, cette condition devient nécessaire et suffisante.

Préordres de simulation

Le calcul de l'inclusion d'un système dans un autre pour un préordre de simulation se fait par simple modification de la définition de l'ensemble $Fail$:

Proposition 5.3-2 (Etats $Fail$ pour un préordre)

$$\begin{aligned} Fail &= \{(q_1,q_2) \in Q_{\times_\Lambda} \mid \exists (q_1',q_2') \in Q_{\times_\Lambda}, \exists \lambda \in \Lambda, (q_1 \xrightarrow{\lambda}_{T_1} q_1') \not\Rightarrow (q_2 \xrightarrow{\lambda}_{T_2} q_2')\} \\ &= \{(q_1,q_2) \in Q_{\times_\Lambda} \mid \mathcal{A}ct_\Lambda(q_1) \not\subseteq \mathcal{A}ct_\Lambda(q_2)\} \end{aligned}$$

■

5.3.2 Mise en œuvre pour différentes relations

Nous donnons maintenant les définitions du produit synchrone et de l'ensemble $Fail$ pour différentes équivalences de bisimulation et relation de préordre.

Bisimulation et simulation forte

Le cas de la bisimulation forte se traite par une application directe des définitions données précédemment. Le produit synchrone est défini pour l'ensemble de langages $\Lambda = A_1 \cup A_2 \cup \{\tau\}$:

$$\frac{(q_1,q_2) \in Q,\ \exists a \in A_\tau,\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1',q_2')\} \in Q, \{(q_1,q_2) \xrightarrow{a}_T (q_1',q_2')\} \in T} \quad [R1]$$

Préordre et équivalence de sûreté, bisimulation $\tau^* a$

Le préordre de sûreté $\sqsubseteq_s$ est le préordre de simulation obtenu pour l'ensemble de langages suivants :

$$\Lambda_{\tau^* a} = \{\tau^* a \mid a \in A_1 \cup A_2\}$$

qui est l'ensemble de langages permettant la définition de la bisimulation $\tau^* a$(voir chapitre 2).

Ce préordre a une grande importance en pratique, puisqu'il permet de comparer un système avec un *graphe de sûreté* [Rod88]. Les graphes de sûreté permettent de modéliser exactement les *propriétés de sûreté*, qui expriment que toute action effectuée par un système est une action correcte.

Si on considère un système de transitions étiquetées S et un graphe de sûreté S_s, alors $S \sqsubseteq_s S_s$ si et seulement si S vérifie la propriété de sûreté exprimée par S_s.

L'équivalence de sûreté $\sim_s$ est définie à partir du préordre de sûreté :

$$\sim_s = \sqsubseteq_s \cap \sqsubseteq_s^{-1}$$

Le produit synchrone pour ces relations est défini de la manière suivante :

$$\frac{(q_1, q_2) \in Q_{\times_{\tau^* a}},\ \tau^* a \in \Lambda,\ q_1 \xrightarrow{\tau^* a}_{T_1} q_1',\ q_2 \xrightarrow{\tau^* a}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q_{\times_{\tau^* a}}, \{(q_1, q_2) \xrightarrow{\tau^* a}_{T} (q_1', q_2')\} \in T} \qquad [R1]$$

Le calcul de chaque transition $(q_1, q_2) \xrightarrow{\tau^* a}_T (q_1', q_2')$ correspond donc à un calcul de la fonction $Post^*_\tau$ dans chacun des systèmes. Néanmoins, le calcul de l'ensemble $Acc(Q_{\times_\Lambda})$ pour la $\tau^* a$ bisimulation peut se faire d'une autre manière, qui s'avère plus performante, en particulier quand le nombre de transitions visibles dans les systèmes S_1 et S_2 est faible devant le nombre de transitions cachées. L'idée est de calculer l'ensemble des états accessibles du produit synchrone $S_1 \times S_2$, et de ne conserver de cet ensemble que les couples (q_1, q_2) accessibles par une transition visible. Plus formellement, on définit le produit synchrone $S_1 \times S_2$ comme suit :

$$\frac{(q_1, q_2) \in Q_\times,\ \exists a \in \mathcal{A},\ q_1 \xrightarrow{a}_{T_1} q_1',\ q_2 \xrightarrow{a}_{T_2} q_2'}{\{(q_1', q_2')\} \in Q_\times, \{(q_1, q_2) \xrightarrow{a}_{T} (q_1', q_2')\} \in T} \qquad [R1]$$

$$\frac{(q_1, q_2) \in Q_\times,\ q_1 \xrightarrow{\tau}_{T_1} q_1'}{\{(q_1', q_2)\} \in Q_\times, \{(q_1, q_2) \xrightarrow{\tau}_{T} (q_1', q_2)\} \in T} \qquad [R2]$$

$$\frac{(q_1, q_2) \in Q_\times,\ q_2 \xrightarrow{\tau}_{T_2} q_2'}{\{(q_1, q_2')\} \in Q_\times, \{(q_1, q_2) \xrightarrow{\tau}_{T} (q_1, q_2')\} \in T} \qquad [R3]$$

Les règles R2 et R3 correspondent à la composition asynchrone des τ.

Le calcul des états accessibles dans $Acc(Q_\times)$ de ce nouveau produit synchrone va nous permettre d'obtenir l'ensemble $Acc(Q_{\times_\Lambda})$. Il suffit de distinguer dans le produit les couples (q_1, q_2) accessibles par une transition visible, comme le montre la proposition suivante :

Proposition 5.3-3
$Acc(Q_{\times_\Lambda}) = \{(q_1, q_2) \in Acc(Q_\times) \mid \exists a \in A_1 \cup A_2, \exists (q_1', q_2') \in Q_\times, (q_1', q_2') \xrightarrow{a}_T (q_1, q_2)\}$ ■

Remarque 5-1
L'intérêt principal du calcul des états accessibles dans $S_1 \times S_2$ par rapport à $S_1 \times_\Lambda S_2$ est la conservation des états du produit qui ne sont pas accessibles par une action visible. En effet, conserver ces états permet d'éviter de repasser plusieurs fois par les mêmes états du modèle lors de calculs différents de la fonction $post_{\tau^* a}$. Le stockage d'états supplémentaires ne provoque pas forcément une augmentation des besoins mémoire, puisque nous utilisons une représentation symbolique. En fait, dans le cas des BDDs, nous pouvons même avoir un BDD pour l'ensemble $Acc(Q_\times)$ *plus petit* que celui représentant $Acc(Q_{\times_\Lambda})$. ■

L'ensemble $Fail$ pour ces relations est défini comme suit :

Préordre de sûreté
$$Fail = \{(q_1, q_2) \in Q_\times \mid \mathcal{A}ct_{\tau^* a}(q_1) \not\subseteq \mathcal{A}ct_{\tau^* a}(q_2)\}$$

$\tau^* a$ bisimulation
$$Fail = \{(q_1, q_2) \in Q_\times \mid \mathcal{A}ct_{\tau^* a}(q_1) \neq \mathcal{A}ct_{\tau^* a}(q_2)\}$$

5.4 Conclusion

Nous avons présenté plusieurs applications de notre représentation symbolique de Réseaux de Petri. La première application consiste à calculer les états accessibles d'un modèle, ce qui permet aussi de déterminer certains états caractéristiques, comme les états *puits* ou les états de *divergence*. Nous avons d'autre part présenté l'application de cette représentation pour la génération de modèle minimal, et pour la comparaison de modèles.

L'ensemble de ces applications nous donne toute une gamme de méthodes et de relations pour la vérification comportementale de systèmes. De plus, nous avons adapté l'algorithme de minimisation à différentes bisimulations, et les algorithmes de comparaison de modèles aussi bien pour des relations de bisimulation que pour des relations de simulation. Cette dernière application a été faite avec dans l'idée d'utiliser au maximum un des points forts des BDDs, qui est dans notre cas le calcul de points fixes en avant. Nous avons au maximum évité un des points faibles, qui est la composition de relations. C'est pourquoi nous avons restreint l'exploration de produit synchrone au cas où un des deux systèmes est déterministe.

Dans le chapitre 8, nous décrivons dans quel cadre l'ensemble de ces applications ont été implémentées. Les performances des implémentations qui sont présentées dans le même chapitre ont montré les possibilités des BDDs dans une boîte à outils de vérification formelle, en complément avec des méthodes plus classiques.

Chapitre 6

Polyèdres

Dans le chapitre 3 , nous avons utilisé une représentation symbolique pour la représentation du contrôle, et nous l'avons adaptée la représentation des contextes, dans le cadre restreint de variables booléennes et de variables entières bornées. Nous allons maintenant nous intéresser à des représentations numériques, qui sont naturellement mieux adaptés à la représentation des variables entières. Ces représentations sont habituellement utilisés dans le cadre de systèmes d'*analyse sémantique* des programmes; ces systèmes d'analyse permettent de déterminer *statiquement* des propriétés vérifiées à l'*exécution* par les *variables* d'un programme.

Ces analyses peuvent être *non relationnelles*, i.e. elles ne permettent pas d'établir des relations entre variables. Parmi les analyses non relationnelles, nous trouvons en particulier la propagation de constantes [Kil73], le calcul des signes de variables [CC76], l'analyse de congruences arithmétiques et enfin l'analyse des intervalles [Bou92] qui généralise les méthodes de propagation de constantes.

Les analyses *relationnelles* permettent d'établir des relations *linéaires* entre variables. Parmi ceux-ci, nous distinguons les systèmes d'équations linéaires :

$$\sum_{i=1}^{n} a_i x_i = b$$

où a_i et b sont des vecteurs de rationnels. Ces systèmes sont utilisés dans [Kar76] pour la découverte d'égalités linéaires entre variables.

Plus récemment, on trouve un système d'analyse basé sur des *égalités linéaires de congruence* [Gra91] :

$$\sum_{i=1}^{n} a_i x_i \equiv b[m]$$

où m est un entier naturel. Ce système contient l'analyse des égalités linéaires et l'analyse des congruences.

Enfin, un dernier type de représentation qui généralise toutes les précédentes consiste en des

systèmes *inéquations linéaires*, aussi connus sous le nom de *polyèdres convexes*. Ces systèmes sont de la forme :

$$\sum_{i=1}^{n} a_i x_i \leq b$$

Cette représentation est utilisée pour l'analyse de relations d'inégalités linéaires entre variables [CH78, Hal79]. Dans ces travaux, cette analyse est utilisée pour des programmes impératifs, en particulier pour calculer des approximations supérieures d'invariants du programme, ou pour calculer des conditions nécessaires (sur les variables du programme) pour qu'un ensemble d'états soit atteint.

Dans la suite de ce chapitre, nous allons nous intéresser aux polyèdres convexes et aux opérateurs permettant de les manipuler.

Nous passerons ensuite à la définition d'un système permettant l'*analyse* de notre modèle Réseau de Petri, à l'instar des travaux présentés dans [Hal79] et [Bou92]

6.1 Définition

Un polyèdre peut être défini comme l'ensemble des solutions d'un système d'équations et d'inéquations linéaires. Même si dans ces solutions, nous pouvons avoir des solutions réelles, nous restreindrons par la suite les polyèdres à l'espace à n dimensions des rationnels, $\mathbb{Q}^n$.

Un polyèdre convexe peut être représenté de manière équivalente par une des deux formes suivantes :

Un *système de contraintes*
Un système de contraintes est un *système d'équations et d'inéquations linéaires*

$$P = \{x \mid Ax = b, A'x \geq b'\}$$

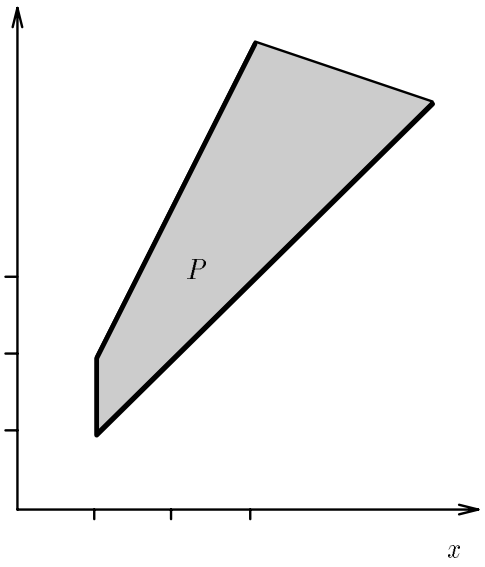
avec $x = \{x_i \mid i \in [1 \ldots n] \wedge x_i \in \mathbb{Q}\}$, A et A' sont des matrices d'entiers relatifs, b et b' des vecteurs d'entiers relatifs.

Un *système générateur*
Un système générateur est une représentation paramétrique de P :

$$P = \{x \mid x = S\lambda + R\mu + D\nu, \lambda, \mu \geq 0, \sum \lambda = 1\}$$

avec S, R et D sont des ensembles de vecteurs de rationnels.

- S est l'ensemble des *sommets*
- R est l'ensemble des *rayons*
- D est l'ensemble des *droites*



$$P = \left\{ (x,y) \mid \begin{pmatrix} x & \geq & 1 \\ y & \geq & x \\ x & \geq & 2y \end{pmatrix} \right\}$$

$$S = (\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix})$$

$$R = (\begin{pmatrix} 1 \\ 1 \end{pmatrix} \begin{pmatrix} 1 \\ 2 \end{pmatrix})$$

$$D = ()$$

Figure 6.1: Un exemple de polyèdre

Dans la suite, nous donnerons un polyèdre P soit par son système de contraintes ($P = \{x \mid Ax \leq b\}$), soit par son système générateur ($P = (S, R, D)$).

Aucune de ces représentations n'admet de forme normale. Il est possible néanmoins d'en calculer une forme *minimale*, i.e. une forme ne possédant pas d'éléments redondants. Nous verrons plus loin quel algorithme nous utilisons pour cette minimisation.

6.1.1 Quelques définitions

Soit un polyèdre $P = \{x \mid Ax \leq b\}$de système générateur $P = (S, R, D)$, nous avons les définitions suivantes :

Définition 6.1-1 (Saturation d'une contrainte)
Soit p un point et r un rayon, on dit que p (resp. r) sature une contrainte $ax \leq b$ si et seulement si $av = b$(resp. $ar = 0$). ■

Définition 6.1-2 (Contrainte redondante)
Une contrainte c_1 est dite redondante par rapport à une contrainte c_2 si et seulement si tout élément du système générateur de P saturant c_1 sature aussi c_2. Si c_1 est redondante par rapport à c_2, et c_2 redondante par rapport à c_1, alors c_1 et c_2 sont dites mutuellement redondantes ■

Définition 6.1-3 (système de contraintes minimal)
Un système de contraintes est dit minimal *si aucune de ses contraintes n'est redondante par rapport à une autre* ■

Définition 6.1-4 (Sommet ou rayon redondant)
Un sommet $s_1 \in S$ (resp. rayon $r_1 \in R$) est dit redondant *par rapport à $s_2 \in S$ (resp.*

$r_2 \in R$) si et seulement si toute contrainte saturée par s_1 (resp. r_1) l'est aussi par s_2 (resp. r_2) ■

Définition 6.1-5 (système générateur minimal)
Un système de contraintes est dit minimal *si aucun de ses sommets (resp. rayons) n'est redondant par rapport à un autre sommet (resp. rayon)* ■

Quelques polyèdres particuliers

Définition 6.1-6 (Polytope)
Un polytope *est un polyèdre borné :*
son système générateur (S, R, D) est tel que $R = D = \emptyset$ ■

Définition 6.1-7 (Polyèdre entier)
Un polyèdre est dit entier ssi tous ses sommets sont entiers. ■

6.2 Opérations sur les polyèdres

La définition des différents opérateurs agissant sur les polyèdres tirent profit de la représentation duale de ceux-ci.

6.2.1 Opérateurs binaires

Intersection :
L'intersection de deux polyèdres correspond à l'ensemble des points vérifiant les systèmes de contraintes des *deux* polyèdres. Autrement dit, l'intersection de deux polyèdres $P_1 = \{x \mid A_1 x \leq b_1\}$ et $P_2 = \{x \mid A_2 x \leq b_2\}$ est définie par la réunion de leur système de contraintes :

$$P_1 \cap P_2 = \{x \mid A_1 x \leq b_1, A_2 x \leq b_2\}$$

Union :
L'union de deux polyèdres n'est en général pas un polyèdre. Nous approcherons cette union par le calcul de l'*enveloppe convexe*.

Enveloppe convexe :
L'enveloppe convexe d'un ensemble de vecteurs est l'ensemble de toutes les combinaisons convexes de ces vecteurs. Pour construire l'enveloppe convexe de deux polyèdres $P_1 = (S_1, R_1, D_1)$ et $P_2 = (S_2, R_2, D_2)$, nous calculons la réunion des deux systèmes générateurs :

$$P_1 1 \cup P_2 = \{x \mid x = S\lambda + R\mu + D\nu, \lambda, \mu \geq 0, \sum \lambda = 1\}$$

où

- $S = S_1 \cup S2$
- $R = R_1 \cup R_2$
- $D = D_1 \cup D_2$

6.2.2 Comparaison de polyèdres

Comparaison au vide :
Un polyèdre *vide* est défini par un système de contraintes contenant deux contraintes incompatibles (ex: $x \geq 4$ et $x \leq 0$) ou par un système générateur sans aucun éléments.

Comparaison à $\mathbb{Q}^n$:
Un polyèdre est égal à $\mathbb{Q}^n$ si son système de contraintes est vide, ou si son système générateur (S, R, D) est tel que : SS est réduit à un sommet quelconque de $\mathbb{Q}^n$,
$R = \emptyset$
$D = \{(1, 0, \ldots, 0), (0, 1, 0, \ldots, 0), \ldots, (0, 0, \ldots, 1)\}$

Inclusion :
Le calcul de l'inclusion d'un polyèdre P_1 dans un polyèdre P_2 se fait par le calcul de la satisfaction des contraintes de P_2 par chaque élément du système générateur de P_1 :
Soient $P_1 = (S_1, R_1, D_1)$ et $P_2 = \{x \mid A_2 x \leq b_2\}$ $P_1 \subseteq P_2$ –
$\forall s \in S_1, A_2 s \leq b_2$,
$\forall r \in R_1, A_2 r \leq 0$,
$\forall d \in D_1, A_2 d = 0$

Egalité :
Ni le système de contraintes, ni le système de contraintes ne sont des formes normales pour la représentation de polyèdres. Le calcul de l'égalité de deux polyèdres se fait donc par le calcul de la double inclusion : Soient P_1 et P_2, $P_1 = P_2$ – $P1 \subseteq P_2 \wedge P_2 \subseteq P_1$

6.2.3 Transformations

Soit un polyèdre P et une fonction affine f telle que $f(x) = Cx + d$. Nous allons définir l'*image* de P par la transformation f et l'*image inverse* de P par f. Nous donnons ensuite la définition de la *projection* d'un polyèdre suivant une variable donnée.

Image par une transformation affine
Ce calcul se fait par application de f sur le système générateur de P.

$$f(P) = (f(S), f(R), f(D))$$

Image inverse par une transformation affine
L'image inverse se calcule par application de f sur le système de contraintes de P :

$$f^{-1}(P) \quad = \quad \{y \mid f(y) = x, x \in P\}$$

$$\begin{aligned} &= \{y \mid A.f(y) \geq b\} \\ &= \{y \mid A.C.y \geq b - Cd\} \end{aligned}$$

Projection :

La projection d'un polyèdre suivant une dimension x correspond à l'élimination de la variable x dans le système de contraintes du polyèdre.

Si P est un polyèdre convexe de $\mathbb{Q}^n$, la *projection* selon la i-ème variable de Pde $\mathbb{Q}^{n-1}$ est le polyèdre correspondant à l'ensemble de points suivant :

$$\begin{aligned} proj(P, x_i) &= \{(x_1, \ldots x_{i-1}, x_{i+1}, \ldots, X_n) \in R^{n-1} \mid \\ &\quad \exists z \in R : (x_1, \ldots, x_{i-1}, z, x_{i+1}, \ldots, X_{n-1}) \in P\} \end{aligned}$$

Le calcul de la projection d'un polyèdre P se fait par modification de son système générateur : soit $P = (S, R, D)$ le système générateur de P, alors le système $(S, R, D \cup \{x_i = 0\})$ constitue un système générateur de $proj(P, x_i)$.

6.2.4 Passage d'une représentation à l'autre

La plupart des opérateurs présentés nécessitent l'une ou l'autre (voire les deux) des deux formes de représentation de polyèdres. Pour pouvoir passer d'une forme de représentation à l'autre, nous utilisons un algorithme élaboré initialement par Chernikova [Che68], puis amélioré et mis en œuvre par [Ver92]

Cet algorithme permet le calcul d'un système générateur à partir d'un système de contraintes (et réciproquement) et de plus permet la minimisation de chacune des deux représentations. Son utilisation permet de réduire la définition de la plupart des opérateurs nécessaires à des unions ou intersections d'ensembles de vecteurs

6.3 Calcul de points fixes

L'ensemble des méthodes proposées dans les chapitres précédents sont basées essentiellement sur des calculs de points fixes. Dans le cas des BDDs, ces calculs de points fixes se faisaient sur un treillis de domaine fini, puisque la méthode de codage que nous avons utilisée ne permettait que de représenter des modèles finis.

L'utilisation de polyèdres comme méthode de représentation de modèles change les données du problème, puisque le treillis des polyèdres est de hauteur infinie. Les calculs de point fixe sur de tels modèles peuvent ne pas converger. D'autre part, même si les calculs convergent, il est possible que le nombre d'itérations (donc le temps de calcul) soit prohibitif pour une utilisation efficace.

Pour garder la possibilité de travailler sur ces modèles, il faut envisager de ne calculer que des approximations de ces points fixes.

Un cadre théorique permettant le calcul de ces approximations a été défini par Patrick et Radhia Cousot [Cou81]. En particulier, ils proposent une méthode qui permet une accélération de la convergence de points fixes et éventuellement leur calcul en un temps fini. Cette méthode est basée sur la définition d'un opérateur d'*élargissement*.

L'idée générale de cette séquence de calcul est de "sauter au delà"du plus petit point fixe grâce à l'opérateur d'élargissement. Ce calcul est souvent complété par une séquence de calcul décroissante, qui permet de revenir au plus près de ce plus petit point fixe grâce à un *opérateur de rétrécissement*, sans jamais passer en deçà. L'ensemble des théorèmes rappelés dans les sections suivantes ainsi que leur preuve se trouvent dans [Cou81].

6.3.1 Opérateur d'élargissement

L'opérateur d'élargissement est défini de la manière suivante :

Définition 6.3-1 (Opérateur d'élargissement)

Soit un treillis complet $(L, \bot, \top, \sqsubseteq, \sqcap, \sqcup)$. *On appelle* opérateur d'élargissement *l'opérateur* $\triangledown\ L \longmapsto L$ *tel que :*

$$\forall x, y \in L, x \sqcup y \quad \sqsubseteq \quad x \triangledown y$$

et pour toute suite croissante $x_0 \sqsubseteq x_1 \sqsubseteq \cdots$ *la suite* $y_0 \sqsubseteq y_1 \sqsubseteq \cdots$ *définie par :*

$$\begin{cases} y_0 = x_0 \\ y_{n+1} = y_n \triangledown x_{n+1} \end{cases}$$

est croissante, mais constante à partir d'un certain rang :

$$\exists n_0 \in I\!N : \forall n \in I\!N, n \geq n_0 \Rightarrow y_n = y_{n_0}$$

■

L'introduction de cet opérateur dans le calcul d'un point fixe se fait alors comme suit :
soit F une fonction continue sur le treillis complet L et φ son plus petit point fixe. φ est la limite supérieure de la suite croissante :

$$\begin{cases} \varphi_0 = \bot \\ \varphi_{n+1} = F(\varphi_n) \end{cases}$$

Un exemple d'utilisation de $\triangledown$ est donnée dans le calcul du post-point fixe Φ, qui est définie comme la limite de la suite :

$$\begin{cases} \Phi_0 = \bot \\ \Phi_{n+1} = \Phi_n \triangledown F(\Phi_n) \end{cases}$$

Cette définition peut être légèrement améliorée. En effet, tout post-point fixe de F est supérieur au plus petit point fixe de F; on peut donc arrêter le calcul de cette suite dès qu'un post-point fixe est atteint.

$$\begin{cases} \Phi_0 = \bot \\ \Phi_{n+1} = \Phi_n \textbf{ si } (F(\Phi_n) \sqsubseteq \Phi_n \\ \Phi_{n+1} = \Phi_n \triangledown F(\Phi_n) \textbf{ sinon} \end{cases}$$

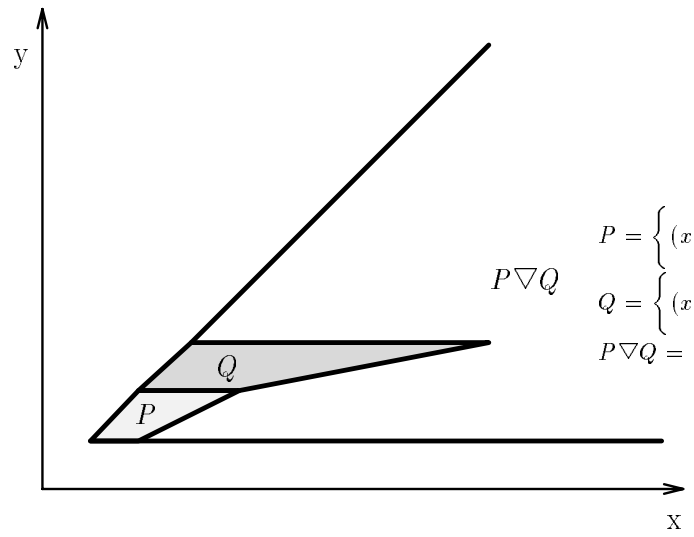


Figure 6.2: Opérateur d'élargissement

6.3.2 Opérateur d'élargissement pour les polyèdres

La définition d'un opérateur d'élargissement adapté aux polyèdres est issue de [Hal79]. L'idée de base d'un élargissement $P\nabla\ Q$ est de ne garder comme contraintes que les contraintes de P vérifiées par Q.

Exemple 6-1
Soient $P = \{(x,y) \mid 1 \leq x \leq 2, x \leq y \leq 2x\}$ et $Q = \{(x,y) \mid 2 \leq x \leq 3, x \leq y \leq 3x\}$ alors (figure 6.1)

$$P\nabla Q = \{(x,y) \mid 1 \leq x, x \leq y\}$$

■

Une optimisation apportée par [Hal79] consiste à choisir parmi les différents systèmes de contraintes minimaux définissant P_1 celui contenant le plus de contraintes satisfaisant P_2. Cette optimisation permet d'améliorer sensiblement la précision de l'opérateur d'élargissement, comme le montre l'exemple suivant :

Exemple 6-2

Soit $P = \{(x,y) \mid 1 \leq x \leq 2, y = 1\}$(Figure 6.3,(a)) et $Q = \{(x,y) \mid 1 \leq y \leq x \leq 3\}$(Figure 6.3,(b)). Si nous appliquons directement l'opérateur d'élargissement, nous obtenons (Figure 6.3,(c))

$$P\nabla Q = \{(x,y) \mid 1 \leq x, 1 \leq y\}$$

Considérons pour P le système de contraintes $\{(x,y) \mid 1 \leq y \leq x \leq 2, y \leq 1\}$. Ce nouveau système de contraintes n'est pas minimal, mais équivalent au précédent. Le résultat de

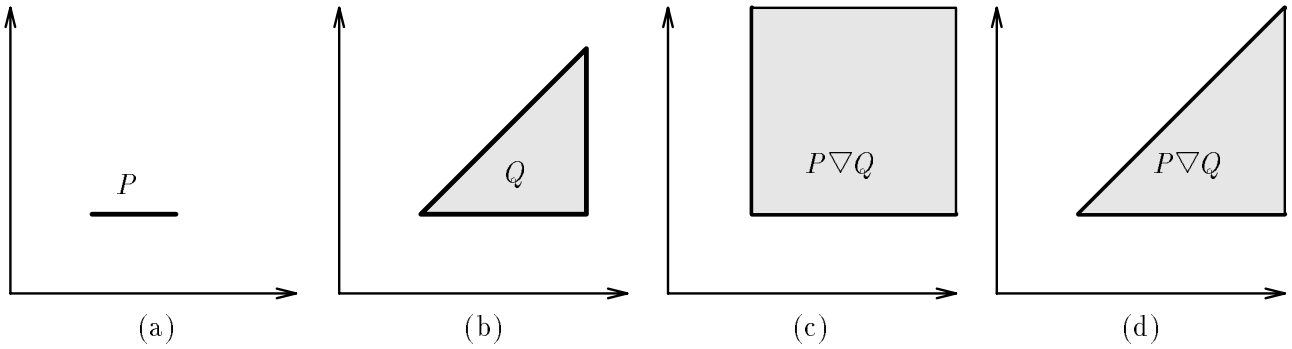


Figure 6.3: Elargissement optimisé

l'élargissement de P par Q devient alors (Figure 6.3,(d) :

$$P \triangledown Q = \{(x, y) \mid 1 \leq y \leq x\}$$

■

Pour tenir compte de cette optimisation, nous modifions le calcul de $P_1 \triangledown P_2$, qui se fait alors par le calcul des ensembles suivants :

Contraintes de P_1 satisfaites par P_2 :
Soit l'ensemble d'indices $M \subseteq \{1, \ldots, m_1\}$ tel que :
$\forall i \in M, \forall x \in P_2, A_1^i x \leq b_i$

Cet ensemble correspond à la définition de l'élargissement sans optimisation

Contraintes de P_2 mutuellement redondantes avec les contraintes de P_1 :
Soit l'ensemble d'indices $N \subseteq \{1, \ldots, m_2\}$ tel que :
$\forall i \in N, \exists j \in \{1, \ldots, m_1\}$:
$P_1 = \{x \mid A_1^k \leq b_1^k, k \in \{1, \ldots, j-1, j+1, \ldots, m_1\}, a_2^i x \leq b_2^i\}$

Cet ensemble permet d'améliorer l'élargissement, en ajoutant dans $P \triangledown Q$ les contraintes de P_2 mutuellement redondantes avec celles de P_1.

Le résultat de l'élargissement est alors donné par la réunion de ces deux ensembles de contraintes, ce qui donne la définition suivante pour l'élargissement optimisé :

$$P_1 \triangledown P_2 = \begin{cases} P_2 \text{ si } P_1 = \emptyset \\ \{x \mid A_1^M x \leq b_1^M \text{ et } A_2^N x \leq b_2^N\} \text{ sinon} \end{cases}$$

6.3.3 Représentation du contrôle

Les polyèdres nous servent à construire une représentation symbolique du modèle Réseau de Petri. Ils permettent en particulier de représenter plus directement les variables entières et donc de traiter efficacement la partie données de notre modèle. Pour compléter cette

représentation symbolique, nous devons aussi introduire le traitement de la partie contrôle. Pour cela, deux solutions sont possibles :

Codage du contrôle dans les variables :
Cette solution revient à étendre la partie donnée du modèle avec des variables codant le contrôle. Dans notre cas, l'idée est de transformer l'ensemble du réseau en systèmes à *commandes gardées* plat, en associant à chaque unité une variable contenant le numéro de la place marquée. Cette solution a l'avantage d'unifier le traitement du contrôle et des données dans une même représentation, comme c'était le cas précédemment dans le cas des BDDs. Néanmoins, ceci présente certains inconvénients :

- le nombre de variables de définition des polyèdres est augmenté en proportion de la complexité du contrôle. Or les performances des opérateurs et la taille de la représentation mémoire dépend beaucoup du nombre de variables.
- les transformations de contrôle sont des combinaisons gardes-affectations qui ne supportent pas la moindre approximation. Or nous voulons appliquer un opérateur d'élargissement pour assurer la convergence des calculs de points fixes.

Traitement du contrôle à part
Une autre solution consiste à séparer le traitement du contrôle et des données. Le contrôle peut être représenté par un modèle classique, comme un système de transitions ou éventuellement par un modèle symbolique à base de BDDs. La séparation du traitement des données et du contrôle, qui est dans la lignée du choix fait pour le Réseau de Petri, est la solution que nous avons adoptée.

La représentation symbolique que nous voulons construire est donc partitionnée par rapport au contrôle.

6.4 Modèle partitionné

Nous avons décrit dans les paragraphes précédents un calcul de point fixe avec utilisation d'un opérateur d'élargissement pour assurer la convergence. Nous allons maintenant définir un calcul de points fixe avec élargissement sur un modèle partitionné par rapport au contrôle.

Soit $\mathcal{M}$ l'ensemble des points de contrôle du programme, nous pouvons réécrire l'équation de point fixe $\Phi = F(\Phi)$ comme un système d'équations :
$\Phi_1 = F_1(\{\Phi_m \mid m \in \mathcal{M}\})$
$\Phi_2 = F_2(\{\Phi_m \mid m \in \mathcal{M}\})$
$\ldots$
$\Phi_{|\mathcal{M}|} = F_{|\mathcal{M}|}(\{\Phi_m \mid m \in \mathcal{M}\})$
où chaque F_i est une restriction de F.

Ce système se résout par application parallèle des fonctions F_i à chaque étape du calcul. Mais ce calcul systématique du résultat de chaque fonction à chaque étape peut être particulièrement inefficace, pour les raisons suivantes :

Calculs inutiles :
Chacune de ces équations correspond à un point de contrôle du programme. Les équations des points de contrôle du début du programme peuvent en général se stabiliser très tôt, celles de fin de programme se stabilisant en dernier.

Calculs inefficaces :
Il est plus efficace avant de recalculer une équation d'attendre qu'un maximum des termes dont elle dépend aient déjà changés.

Pour améliorer l'efficacité globale de ce calcul, il est nécessaire de pouvoir décider d'un ordre des calculs et notamment de pouvoir appliquer le calcul des F_i en *séquence* et non pas seulement en parallèle. La garantie d'un résultat correct pour un ordre de calcul quelconque est donnée par la notion d'*itération chaotique* [Cou78]

Définition 6.4-1 (Itération chaotique croissante)
Soit F une fonction continue sur un treillis complet $(L, \bot, \top, \sqsubseteq, \sqcap, \sqcup)$, et $(O_k)_{k\in I\!N}$ un ensemble de sous ensembles de $\{1,\ldots,|\mathcal{M}|\}$ tel que chaque élément de $\{1,\ldots,|\mathcal{M}|\}$ apparaît infiniment souvent, une itération chaotique est alors définie par :

$$\begin{array}{lcll} \Phi^0 & = & (\bot,\ldots,\bot) & \\ \Phi_i^{k+1} & = & \Phi_i^k & si\ i \notin O_k \\ \Phi_i^{k+1} & = & \Phi_i^k \sqcup F_i(\{\Phi_m^k \mid m \in \mathcal{M}\}) & si\ i \in O_k \end{array}$$

■

Cette définition nous assure la convergence du système d'équations, quel que soit l'ordre choisi pour l'application des équations de ce système, tant que chaque équation est appliquée infiniment souvent.

Nous pouvons généraliser cette notion aux cas de treillis de hauteur non finie ou ne vérifiant pas la condition de chaîne, par l'introduction comme précédemment d'opérateurs d'élargissement. Une utilisation naïve de l'opérateur d'élargissement consisterait à effectuer un élargissement dans chaque équation. Une équation du système 6.4-1 serait alors de la forme :

$$\Phi_i^{k+1} = \Phi_i^k \triangledown F_i\{\Phi_m^k \mid m \in \mathcal{M}\})) \text{ si } i \in E_k$$

Comme l'opérateur d'élargissement est une grande source d'imprécisions, il est nécessaire de restreindre au maximum son utilisation. Pour pouvoir déterminer un ensemble suffisant de points d'élargissement, nous devons construire le *graphe de dépendance* des équations :

Définition 6.4-2 (Graphe de dépendance)
soit $\Phi = F(\Phi)$ le système d'équations de la forme :

$$\begin{array}{l} \Phi_1 = F_1(\{\Phi_m \mid m \in \mathcal{M}\}) \\ \Phi_2 = F_2(\{\Phi_m \mid m \in \mathcal{M}\}) \\ \ldots \\ \Phi_{|\mathcal{M}|} = F_{|\mathcal{M}|}(\{\Phi_m \mid m \in \mathcal{M}\}) \end{array}$$

Le graphe de dépendance de ce système est un graphe orienté $\mathcal{D} = (\mathcal{M}, \mathcal{T})$ *tel que*

$$\forall m_1, m_2 \in \mathcal{M}, (m_1, m_2) \in \mathcal{T} -$$

le calcul de $\Phi_{m_2} = F_{m_2}(\{\Phi_m \mid m \in \mathcal{M}\})$ *dépend de la valeur de* Φ_{m_1}

■

En particulier, $\forall i \in [1 \ldots \mid \mathcal{M} \mid], F_i(\{\Phi_m \mid m \in \mathcal{M}\}) = F_i(\{\Phi_j \mid \exists (m_j, m_i) \in \mathcal{T}\}$.

Une bonne intuition pour choisir les points d'élargissement dans ce graphe de dépendance est de s'assurer que chaque circuit de ce graphe passe au moins par un point d'élargissement.

Nous pouvons alors reprendre la définition de l'itération chaotique et en déduire la définition de l'*itération chaotique approchée supérieurement* [Cou78].

Définition 6.4-3 (Itération chaotique croissante approchée supérieurement)
Soit F *une fonction continue sur un treillis complet* $(L, \bot, \top, \sqsubseteq, \sqcap, \sqcup)$ *et un ordre d'itération* $(O_k)_{k \in \mathbb{N}}$, *soit* $W \subseteq \mathcal{M}$ *un ensemble de points tels que chaque circuit de* $(\mathcal{M},\mathcal{T})$ *passe au moins par un point de* W. *Une itération chaotique croissante approchée supérieurement est alors définie par :*

$$\begin{array}{lcll}
\Phi^0 & = & (\bot, \ldots, \bot) & \\
\Phi_i^{k+1} & = & \Phi_i^k & \text{if } i \notin O_k \text{ ou } F_i(\{\Phi_m \mid m \in \mathcal{M}\}) \sqsubseteq \Phi_i^k \\
\Phi_i^{k+1} & = & \Phi_i^k & \text{if } i \notin O_k \\
\Phi_i^{k+1} & = & \Phi_i^k \sqcup F_i(\{\Phi_m \mid m \in \mathcal{M}\}) & \text{if } i \in O_k - W \text{ et } F_i(\{\Phi_m \mid m \in \mathcal{M}\}) \not\sqsubseteq \Phi_i^k \\
\Phi_i^{k+1} & = & \Phi_i^k \triangledown F_i(\{\Phi_m \mid m \in \mathcal{M}\}) & \text{if } i \in W \cap O_k \text{ et } F_i(\{\Phi_m \mid m \in \mathcal{M}\}) \not\sqsubseteq \Phi_i^k
\end{array}$$

■

Remarque 6-1
Cette définition nous donne les moyens d'agir sur la *vitesse* de convergence du calcul, par le choix d'un bon ensemble de points d'élargissement et la détermination d'un ordre d'itération adapté. De plus, si l'opérateur d'élargissement n'est pas monotone, alors ces paramètres vont aussi influer sur la *précision* du calcul([Cou78], Remarques 4.1.2.0.7). ■

6.4.1 Calcul de l'ensemble W des points d'élargissement

Le calcul de l'ensemble W revient à trouver un ensemble minimal de sommet de $\mathcal{D}$ tel que chaque circuit de D passe par un de ces sommets. Ce problème se réduit au problème du *minimum feedback vertex set* [GJ79], qui est un problème NP-complet. Nous devons donc nous restreindre à trouver des solutions approchées pour pouvoir rester dans le cas de graphes de dépendance quelconques.

Chaque circuit du graphe est nécessairement contenu dans une composante fortement connexe, donc il y aura au minimum autant de points d'élargissement que de composantes fortements connexes. L'idée de base est alors de décomposer $\mathcal{M}$ en *composantes fortements connexes* et de choisir dans chaque composante fortement connexe un sommet qui sera son point d'élargissement. Mais cette méthode ne nous donne pas forcément un ensemble *suffisant* de points d'élargissement; en effet, certains circuits d'une composante fortement connexe peuvent ne pas passer par le point d'élargissement choisi. Dans l'exemple 6.4, le

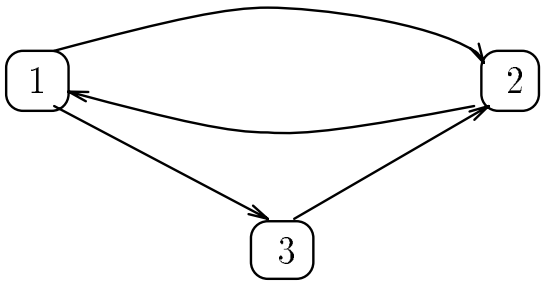


Figure 6.4: Points d'élargissement

sommet 3 n'est pas suffisant comme point d'élargissement, puisque le circuit (1 2) ne passe pas par 3.

Nous pouvons alors continuer la décomposition en composantes fortements connexes, en déconnectant chaque composante fortement connexe déjà obtenue. Il suffit pour cela d'ôter le sommet choisi comme point d'élargissement de chaque composante, et d'appliquer de nouveau l'algorithme de décomposition récursivement sur chaque composante jusqu'à ce que la décomposition ne donne plus que des sommets simples. A chaque étape de la décomposition, de nouveaux points d'élargissement sont choisis et ôtées des composantes.

Cet algorithme correspond à celui proposé dans [Bou92]; il permet de déterminer un ensemble *admissible* de points d'élargissement dans un temps au pire quadratique par rapport au nombre de sommets du graphe.

Cette décomposition récursive permet d'assurer que chaque circuit du graphe passe bien par un point d'élargissement et nous fournit donc un ensemble *admissible* de points d'élargissements. La qualité de cet ensemble (et son cardinal) dépend alors du choix qui est fait à chaque étape d'un sommet dans une composante. Comme le problème de l'optimalité de cet ensemble est NP-complet, on peut suspecter qu'un algorithme permettant un choix optimal de point d'élargissement dans une composante est lui-même exponentiel.

Nous pouvons alors décider de choisir arbitrairement un sommet dans la composante, ou alors de choisir un sommet particulier. Un bon candidat semble alors être le point d'entrée de la composante lors d'un parcours en profondeur d'abord. En effet, comme le montre [Bou92], le choix de ce sommet comme point d'élargissement pour la composante permet de trier les sommets du graphe suivant un ordre permettant d'assurer que toute équation qui ne correspond pas à un point d'élargissement sera appliquée *après* les équations dont elle dépend.

L'algorithme est basé sur la solution de Tarjan [Tar83] pour le calcul de la partition d'un graphe en composantes fortements connexes. La description qui en est donnée en figure 6.5 est tirée de [Bou92, p.43]. Cet algorithme utilise les variables globales suivantes :

- $DFN : \mathcal{M} \rightarrow I\!N$ est un tableau qui associe à chaque sommet son *numéro en profondeur d'abord*
- $Numero : integer$ permet la numérotation du sommet en cours d'exploration

- $m_0 \in \mathcal{M}$ est le sommet initial du graphe

L'appel initial à cet algorithme est par la fonction *Partition*.

Nous donnons un exemple de décomposition d'un graphe donné suivant cet algorithme.

Exemple 6-3

La décomposition en composantes fortements connexes du graphe de la figure 6.6 donne le résultat suivant :

$$0\ (\underline{1}\ (\underline{2}\ 6\ 7))(\underline{3}\ (\underline{4}))\ (5)$$

La notation $(\ldots)$ correspond à une composante fortement connexe; chaque sommet souligné est le point d'entrée d'une composante, l'ensemble des sommets soulignés correspond donc à un ensemble *admissible* de points d'élargissement. ■

Remarque 6-2
L'ensemble *minimal* de points d'élargissement est l'ensemble $\{2\ 4\}$ dans cet exemple. ■

Tri topologique

Un avantage de cet algorithme est le tri des sommets du graphe selon un ordre *topologique* [Bou92]; si on note $<_t$ l'ordre sur les sommets fournis par l'algorithme et W l'ensemble des point d'élargissements choisis, alors :

$$\forall i, j \in [1 \ldots |\ \mathcal{M}\ |], m_i \longrightarrow_{\mathcal{T}} m_j \wedge m_j \notin W \Rightarrow m_i <_t m_j$$

Ce tri permet d'obtenir un calcul efficace des équations :
pour toute équation $\Phi_{m_j} = F_{m_j}(\{\Phi_m \mid m \in \mathcal{M}\})$ telle que $\exists (m_i, m_j) \in \mathcal{T}$ (Φ_{m_j} dépend de Φ_{m_i}), si $m_j \notin W$, alors m_i sera calculé avant m_j.

6.4.2 Stratégie d'itération des calculs

La décomposition en composantes fortements connexes nous donne un ordre avec lequel effectuer les calculs, puisque elle trie les sommets et composantes suivant un ordre topologique. Il reste néanmoins à déterminer une stratégie de calculs pour la stabilisation de chaque composante. Nous distinguons deux stratégies :

Stratégie itérative
La stratégie itérative consiste à essayer de stabiliser la composante globalement, sans considération de ces sous composantes, pour n'en retenir que l'ensemble des points d'élargissements calculés. L'ordre des itération est alors :

$$0\ (\underline{1}\ \underline{2}\ 6\ 7)^*(\underline{3}\ \underline{4}\ )^*\ 5$$

où la notation * signifie qu'il faut itérer le calcul dans la composante fortement connexe jusqu'à stabilisation.

```
function Partition
    foreach m ∈ M do DFN[m] := 0
    Numero := 0
    P := empty
    Visit(m0,P)
endfunc

function Composante(m : sommet)
    var
        m1 : sommet
        P : partition

    P := empty
    foreach (m, m1) ∈ T do
        if DFN[m1] = 0 then Visit(m1,P)

    return (m⌣ P)
endfunc

function Visit(m : sommet, P : partition)
    var
        Numero,min,head : integer
        loop : boolean
        m1,m2 : sommet

    push(m)
    head := DFN[S] := ++Numero
    Loop := false
    foreach (m, m1) ∈ T do
        if DFN[m1] = 0 then min := Visit(m1,P)
        else min := DFN[m1]
    if min ≤ head then
        head := min
        loop := true
    fi
    if head = DFN[m] then
        DFN[m] := +∞
        pop(m2)
        if loop then
            while m ≠ m2 do
                DFN[m2] := 0
                pop(m2)
            od
            P := Composante(m) ⌣ P
        else
            P := m ⌣ P
endfunc
```

Figure 6.5: Algorithme des sous composantes fortement connexes

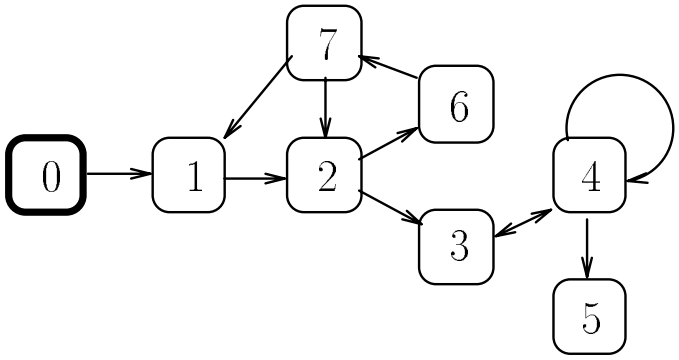


Figure 6.6: Exemple de calcul de composantes fortements connexes

Stratégie récursive
La stratégie récursive consiste à donner la priorité à la stabilisation des composante fortement connexe les plus internes. Si nous considérons l'exemple 6-3, cette stratégie correspond à l'expression suivante :

$$0\ (\underline{1}\ (\underline{2}\ 6\ 7)^*)^*(\underline{3}\ (\underline{4})^*\ )^*\ 5$$

Dans chacun des ces ordres, nous utilisons l'ordre topologique $<_t$ sur les sommets comme ordre d'itération à l'intérieur de chaque composante. Nous présentons en figure 6.7 un algorithme d'itération des équations du système, qui utilise la stratégie récursive. Dans cet algorithme, nous utilisons certaines variables dont la signification est la suivante :

- $W \subseteq 2^{\mathcal{M}}$ est l'ensemble des points d'élargissements
- l'ordre O_t est l'ordre d'itération obtenu par application de l'algorithme de décomposition
- la fonction $CFC : \mathcal{M} \rightarrow 2^{\mathcal{M}}$ associe à un sommet la composante fortement connexe dont elle est la tête dans la décomposition du graphe que nous avons calculée
- $Stable \subseteq 2^{\mathcal{M}}$ est l'ensemble des sommets marqués stable.
- Φ_m et Φ'_m représentent respectivement le contexte courant et le contexte en cours de calcul d'un sommet m. Ces contexte sont initialement vides, à l'exception de Φ'_m du sommet initial

6.5 Conclusion

Nous avons présenté une représentation symbolique de modèle basées sur des *polyèdres convexes*. Cette méthode de représentation, qui permet de décrire des contraintes sur les variables d'un programme et aussi des relations entre ces variables, est couramment utilisé

```
procedure Stabilise(C : ensemble de sommets)

while C \ Stable ≠ ∅ do
    Soit m ∈ C \ Stable tel que m est minimal dans C \ Stable pour l'ordre O_t

    Stable := Stable ∪ {m}
    if Φ'_m ⋢ Φ_m then  -- Le nouveau contexte n'est pas inclus dans le précédent
        if m ∈ W then
            Φ_m := Φ_m ▽ Φ'_m
        else
            Φ_m := Φ_m ⊔ Φ'_m
        fi
        -- propagation du nouveau contexte
        foreach t = (m, m_2) ∈ T do
            if F_t(Φ_m) ≠ ⊥ then
                Φ'_{m_2} := Φ'_{m_2} ⊔ F_t(Φ_m)
                Stable := Stable \ {m_2}
            fi
        od
    fi
    Φ'_m := ⊥
    -- Stabilisation de la sous composante contenant m
    if m est la tête d'une sous composante de C then
        Stabilise(CFC(m)-{m})
    fi
od
```

Figure 6.7: Algorithme de stabilisation

pour l'analyse sémantique de programmes impératifs. Nous allons dans le chapitre suivant l'adapter à l'analyse de programmes décrits par notre modèle Réseau de Petri.

Pour appliquer ces méthodes, nous devons être capable de calculer des points fixes. Comme le treillis des polyèdres est un treillis de hauteur infinie, ces points fixes peuvent ne pas converger. Nous avons présenté des techniques générales qui permettent de forcer et accélérer cette convergence et d'améliorer la précision du calcul. L'utilisation de ces techniques peut être étendu à des modèles dont l'espace d'états est partitionné par le contrôle. Le calcul de point fixe est alors donné par un système d'équations. Il reste alors à déterminer quelles sont les équations à modifier, par introduction d'un opérateur *élargissement*, pour forcer la convergence du calcul.

Pour cela, nous utilisons un algorithme, qui à partir du graphe de dépendance des équations, est capable de découper ce graphe en composantes fortements connexes et de fournir un ensemble *admissible* de points d'élargissement. Même si cet ensemble n'est pas optimal, les résultats expérimentaux sont satisfaisants. Cet algorithme nous fournit d'autre part un ordre sur les équations qui respectent la relation de dépendance entre équations du système. La combinaison de cet ordre et de la décomposition du graphe nous permet d'améliorer significativement les performances de l'algorithme de stabilisation.

Chapitre 7

Application des polyèdres

Nous présentons dans ce chapitre une application que nous faisons des polyèdres. Il s'agit de l'analyse en avant, pour des programmes décrits par des Réseaux de Petri, tels qu'ils ont été présentés au chapitre 1. Nous définissons d'abord la méthode de construction d'un système d'équations et de leur graphe de dépendance à partir d'un réseau. Puis nous présentons quelques optimisations liées à l'utilisation de l'élargissement pour ce genre de modèles. Nous donnons enfin deux exemples d'application de ce système d'analyse approchée en avant.

7.1 Analyse d'un automate interprété

La construction d'un système d'équations et de son graphe de dépendance correspond à la construction d'un *automate interprété* tel que nous l'avons défini au chapitre 1.

Si nous considérons un automate interprété $\mathcal{C} = (\mathcal{M}, \mathcal{T}, init)$, l'analyse en avant consiste alors à *collecter* en chaque marquage $m \in \mathcal{M}$ l'ensemble des valuations que peuvent prendre les variables. Pour représenter la collection des valuations d'un marquage m, nous utilisons un polyèdre, qui correspond à l'*enveloppe convexe* de cette collection. Par conséquent, l'utilisation de polyèdres pour la représentation de ces ensembles constitue une première approximation.

Le calcul du contexte d'un marquage m est donné par l'équation associée à ce marquage :

$$\Phi_m = \bigsqcup_{(m_1, m, \gamma, \alpha) \in T} \{\alpha(\Phi_{m_1} \sqcap \gamma)\}$$

Remarque 7-1
L'opération $\sqcup$ de cette formule correspond au calcul de l'enveloppe convexe des polyèdres. Par conséquent, ce calcul des contextes constitue une approximation supérieure. ■

La relation de transition de cet automate est bâtie sur la relation de transition entre marquages que nous avons définie au chapitre 1. La construction de l'automate correspond donc à une exploration du graphe de marquages et mémorisation des marquages et transitions

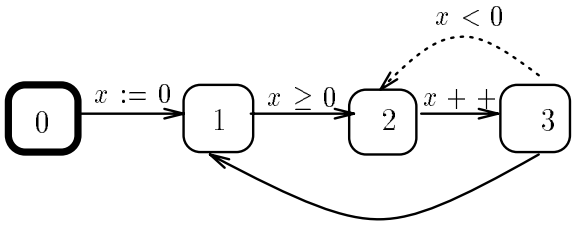


Figure 7.1: Exemple de transition inaccessible

rencontrés. Nous détaillerons certaines techniques pour la mise en œuvre de cette génération dans la partie *Mise en œuvre* de ce chapitre.

Dans la suite, nous continuerons à parler de *graphe de dépendance* et de *système d'équations* lors de la description de la construction et analyse de cet automate interprété.

7.1.1 Analyse du graphe de dépendance

Chaque itération du système d'équations correspond à un parcours du graphe de dépendance, suivant un ordre d'itération fixé. Nous avons vu un algorithme permettant de fixer un tel ordre ; cet algorithme nécessite de connaître *a priori tout* le graphe de dépendance, c'est à dire d'explorer la relation de transition $\longrightarrow_m$ sur les marquages sans tenir compte des contextes. Or pendant l'analyse, certaines transitions et donc certaines branches de ce graphe peuvent se révéler mortes (gardes infranchissables).

La prise en compte de ces transitions peut diminuer considérablement les performances de l'algorithme de décomposition du graphe. Ce problème peut avoir des conséquences gênantes s'il n'est par exemple pas possible de générer le graphe entier, alors qu'un graphe élagué de ses branches mortes serait tout à fait acceptable. De plus, ces branches mortes peuvent provoquer le choix de points d'élargissement inutiles, par rapport aux sommets accessibles du graphe de dépendance. Un exemple simple d'un tel problème est illustré par la figure 7.1. Dans cet exemple, la transition entre les sommets 3 et 2 est infranchissable ; si l'algorithme de décomposition est appliqué en prenant en compte cette transition, le sommet 2 sera inutilement pris comme point d'élargissement supplémentaire.

Enfin, cette exploration a priori du graphe de dépendance est dissociée du processus de résolution du système d'équations, puisque cette exploration se fait sans considérer les résultats des équations. Il parait intéressant de combiner la décomposition du graphe de dépendance avec les itérations de résolution du système d'équations, puisque ces deux calculs font appel à des algorithmes similaires de parcours de graphe.

7.1.2 Décomposition *à la volée* du graphe de dépendance

Cette méthode consiste en une génération et une décomposition du graphe de dépendance au fur et à mesure des itérations de résolution du système d'équations. Chaque itération est

alors réalisée lors d'un parcours en *profondeur d'abord* des sommets du graphe de dépendance. Si le résultat du calcul d'un contexte Φ_m est vide, alors l'exploration n'est pas poursuivie pour les transitions de $\longrightarrow_m$ issues de m. Ceci permet d'éviter l'exploration et génération de branches mortes. Par contre, lors d'itérations ultérieures, des branches jusque là mortes et donc non explorées peuvent devenir actives (une garde devient franchissable).

Le calcul des points d'élargissement et d'un ordre d'itération devient plus difficile; nous ne pouvons plus appliquer des algorithmes prenant en compte le graphe de dépendance dans sa totalité comme l'algorithme 6.5. Une possibilité est alors de calculer les points d'élargissement *dynamiquement*, en utilisant par exemple l'algorithme de décomposition d'un graphe en sous composantes fortements connexes de Tarjan [Tar83]. Cet algorithme permet d'ajouter des sommets et des transitions à un graphe dont une décomposition est déjà connue et de recalculer la nouvelle décomposition en temps $O(\mid \mathcal{T} \mid log \mid \mathcal{M} \mid)$, à comparer avec la complexité en $O(\mid \mathcal{M} \mid^2)$ de l'algorithme 6.5. Malheureusement, ceci ne permet pas d'obtenir un ensemble *admissible* de points d'élargissement([Bou92, p.46]).

Une autre possibilité est un algorithme de détection des *arêtes arrière*, qui nous fournit les sommets m tel qu'il existe au moins une transition $(m', m) \in \mathcal{T}$ et il existe un chemin menant de m à m' (m' est un descendant de m lors du parcours en profondeur d'abord). Cette méthode permet d'obtenir un ensemble admissible de points d'élargissement, lors d'un simple parcours en profondeur d'abord. Malheureusement, elle ne fournit pas d'ordre satisfaisant sur les sommets, puisqu'elle dépend de l'ordre d'exploration en profondeur d'abord qui est arbitraire.

Une dernière idée est d'essayer de guider cette exploration en profondeur d'abord, en tirant parti de la connaissance que nous avons du réseau de Petri qui détermine le graphe de marquages, et en particularité de la décomposition en *unités* de ce réseau. Pour exploiter cette décomposition, nous pourrions appliquer l'algorithme 6.5 pour calculer un ordre d'itérations et une décomposition ainsi qu'un ensemble de points d'élargissement locaux à chaque unité. Comme chaque circuit du graphe de dépendance contient au moins un circuit d'une unité, nous pouvons déduire un point d'élargissement dans le graphe à partir des points d'élargissement locaux. De même, il semble possible de définir un ordre global d'itération acceptable, à partir des ordres locaux.

Néanmoins, nous nous en tiendrons à l'algorithme 6.5 sur le graphe de dépendance global, en raison des avantages déjà cités, mais aussi pour son bon rapport facilité de mise en œuvre/avantages offerts. De plus, même si certaines branches sont générées (et éventuellement certains points d'élargissement inutilement ajoutés), ces branches mortes ne seront pas explorées durant la stabilisation du système d'équations.

7.2 Stratégie d'élargissement

Nous avons donné une méthode de calcul d'un ensemble admissible de points d'élargissements et d'un ordre d'itérations. Cette méthode de calcul permet de tenir compte de la structure du graphe. Associer un ordre d'itérations approprié avec un ensemble suffisant de points

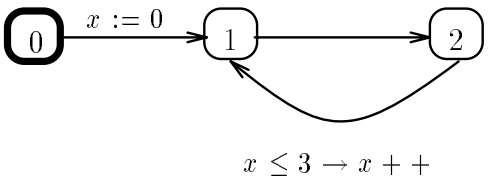


Figure 7.2: Exemple d'analyse

d'élargissement permet de diminuer sensiblement le nombre d'élargissements effectués durant le calcul. Comme l'opérateur d'élargissement que nous allons utiliser pour les polyèdres n'est pas monotone, ces optimisations influent aussi sur la *précision* des calculs.

Nous pouvons encore essayer d'améliorer cette précision. En effet, la précision du résultat d'un élargissement va dépendre de la précision de définition de ses opérandes. Une possibilité pour améliorer la définition de ces opérandes est de retarder l'élargissement, comme le montre l'exemple 7-1

Exemple 7-1

Considérons le graphe de la figure 7.2. Si nous calculons les contextes en considérant le sommet 1 comme point d'élargissement, nous avons comme système d'équations :

$$\begin{aligned}
\Phi_0 &= \{\bot\} \\
\Phi_1^{k+1} &= \Phi_1^k \triangledown (\Phi_0^k[0/x] \sqcup (\Phi_1^k \sqcap \{x \leq 3\})[x+1/x]) \\
\Phi_2^{k+1} &= \Phi_1^k
\end{aligned}$$

Conditions initiales

$$\Phi_0^0 = \Phi_1^0 = \Phi_2^0 = \bot$$

1ère itération

$$\begin{aligned}
\Phi_1^1 &= \bot \triangledown (\Phi_0^0[0/x] \sqcup (\Phi_2^0 \sqcap \{x \leq 3\})[x+1/x]) \\
&= \bot \triangledown \{x = 0\} \\
&= \{x = 0\} \\
\Phi_2^1 &= \{x = 0\}
\end{aligned}$$

2ème itération

$$\begin{aligned}
\Phi_1^2 &= \Phi_1^1 \triangledown (\Phi_0^0[0/x] \sqcup (\Phi_2^1 \sqcap \{x \leq 3\})[x+1/x]) \\
&= \{x = 0\} \triangledown \{0 \leq x \leq 1\} \\
&= \{0 \leq x\} \\
\Phi_2^2 &= \Phi_1^2
\end{aligned}$$

Le calcul se termine donc avec $\{0 \leq x\}$ comme contexte du sommet 1 alors qu'un calcul sans élargissement nous donne $\Phi_1 = \{0 \leq x \leq 4\}$.

■

Pour améliorer ces résultats, nous proposons deux solutions :

Commencer les élargissements après un nombre k d'itérations
Plus le nombre k sera grand, plus on peut s'attendre à ce que le calcul tende vers les résultats qui seraient obtenus sans élargissement. Cette méthode s'applique globalement, pour tous les points d'élargissement. Néanmoins, le choix de k est complètement arbitraire, puisque nous ne disposons pas a priori d'informations sur le nombre d'itérations nécessaires pour assurer que certains contextes soient non vides. La vitesse de convergence des calculs sera d'autant plus basse que k est grand. Il s'agit donc d'un classique compromis entre vitesse et précision des calculs. En pratique, nous laisserons la liberté à l'utilisateur de fixer la valeur de k, qui sera prise par défaut à 0.

Elargissement limité
Une autre solution est de limiter l'élargissement par rapport aux gardes des transitions permettant d'arriver à un point d'élargissement. En effet, ces gardes définissent les conditions d'accès à ce point et donc les valeurs que peuvent contenir son contexte. Dans l'exemple précédent, ceci consisterait à limiter le premier élargissement à $x \leq 4$ (garde $x \leq 3$ sur laquelle est appliquée l'affectation $x++$). Il ne faut cependant pas limiter aveuglément l'élargissement, sous peine d'empêcher éventuellement la convergence du calcul.

Pour mettre en œuvre la deuxième solution, nous utiliserons une version spécialisée de l'opérateur d'élargissement : l'opérateur d'élargissement *limité* $\triangledown_C$ proposé par Nicolas Halbwachs.

Définition 7.2-1 (Elargissement limité)
Soit P et Q deux polyèdres, soit C un ensemble de contraintes, soit $C_s \subseteq C$ l'ensemble des contraintes de C vérifiées par P et *Q. Alors l'élargissement limité est tel que :*

$$(P \triangledown_C Q) = (P \triangledown Q) \sqcap C_s$$

■

Lors d'un élargissement limité $P \triangledown_C Q$, si P et Q vérifient une même contrainte c de C, alors nous contraignons le polyèdre $P \triangledown_C Q$ à respecter c.

Exemple 7-2
Si nous remplaçons dans l'exemple 7-1 la ligne

$$\Phi_1^{k+1} = \Phi_1^k \triangledown \Phi_0^k[0/x] \sqcup (\Phi_1^k \sqcap \{x \leq 3\}[x+1/x]$$

par

$$\Phi_1^{k+1} = \Phi_1^k \triangledown_{C_1} \Phi_0^k[0/x] \sqcup (\Phi_1^k \sqcap \{x \leq 3\}[x+1/x]$$

avec $C_1 = \{x \leq 4\}$ alors les résultats deviennent :

$$\Phi_1^2 = \{0 \leq x \leq 4\}$$

qui dans ce cas est le même résultat que le calcul obtenu sans élargissement. ■

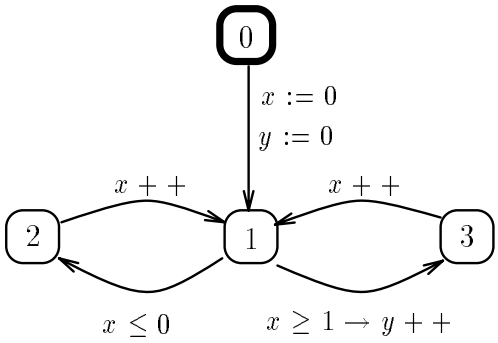


Figure 7.3: Exemple d'analyse

7.2.1 Collection de contraintes de limitation

Nous avons montré comment la collection des gardes des transitions arrivant à un point d'élargissement peut s'associer avec l'utilisation d'un opérateur d'élargissement limité, pour améliorer la précision des résultats.

Néanmoins, ce système n'est pas suffisant, comme le montre l'exemple 7-3.

Exemple 7-3

Considérons l'automate étendu de la figure 7.3.

Suivant la stratégie d'élargissement utilisée, nous obtenons les résultats d'analyse suivants :

Elargissement normal ($k = 0$)

$$\begin{aligned} \Phi_1 &= \boxed{\{0 \leq y\}} \\ \Phi_2 &= \{x \leq 0, 0 \leq y\} \\ \Phi_3 &= \{1 \leq x, 1 \leq y\} \end{aligned}$$

Elargissement retardé ($k \geq 2$)

$$\begin{aligned} \Phi_1 &= \boxed{\{x \leq y + 1, 0 \leq y\}} \\ \Phi_2 &= \{x \leq 0, 0 \leq y\} \\ \Phi_3 &= \{1 \leq x \leq y\} \end{aligned}$$

Elargissement limité

Ensemble de limitation pour le sommet 1 :

$$C = \emptyset$$

donc nous obtenons les mêmes résultats que pour l'élargissement normal :

$$\Phi_1 = \boxed{\{0 \leq y\}}$$

$$\begin{aligned}\Phi_2 &= \{x \leq 0, 0 \leq y\}\\ \Phi_3 &= \{1 \leq x, 1 \leq y\}\end{aligned}$$

La différence entre l'élargissement normal et l'élargissement retardé vient de la première itération, où un élargissement a lieu alors que le contexte Φ_3 est encore vide. Par conséquent, la relation qui s'établit entre x et y lors du parcours du cycle (1 3 1) n'est pas encore établie, et l'élargissement envoie x à l'infini.

L'élargissement limité n'a lui rien changé au résultat, puisque son ensemble de limitation est vide. ■

Sur cet exemple, nous voyons que si l'élargissement retardé a permis d'améliorer les résultats, l'élargissement limité n'a rien changé, car les gardes limitant l'accès au sommet 3 étaient hors de portée. Pour pouvoir traiter de manière plus précise ce genre d'exemples, nous voulons généraliser l'idée de collecter des contraintes de limitation aux points d'élargissement. Une extension est alors de *propager* ces contraintes jusqu'aux points d'élargissements.

Cette propagation de contraintes repose sur les principes suivantes :

Initialisation
A chaque marquage m, nous associons un ensemble de contraintes C_m, initialement vide.

Propagation
Lors d'une première exploration d'un marquage ($\Phi_m = \emptyset$) ou si son ensemble de contraintes est non vide ($C_m \neq \emptyset$), l'ensemble des contraintes C_m est propagé aux successeurs de m : pour chaque transition (m, m_2, γ, α), nous accumulons dans C_{m_2} les contraintes de $C_m \cap \gamma$ transformées par α.

Mémorisation
Si le marquage en cours de traitement est un point d'élargissement, alors nous calculons le sous ensemble de C_m des contraintes qui satisfont Φ_m. Ce sous ensemble devient le nouveau C_m. Si m n'est pas un point d'élargissement, alors l'ensemble C_m est vidé à la fin du traitement de m. Cette remise à zéro permet d'une part d'éviter le stockage d'ensembles de contraintes dans un marquage qui ne l'utilise pas, mais aussi de détecter quand de nouvelles contraintes de limitation sont générées (une branche jusque là morte devient franchissable et propage ses contraintes).

Utilisation
Si le marquage m est un point d'élargissement, alors l'ensemble C_m est utilisé pour limiter l'élargissement.

Remarque 7-2
L'ensemble C_m est bien un ensemble de contraintes et n'est pas forcément un polyèdre. En particulier, il peut contenir des contraintes redondantes, mais aussi des contraintes incompatibles. ■

Contraintes de limitation globales

Nous pouvons compléter les ensembles de contraintes de limitation avec des contraintes déterminées a priori. En particulier, nous pouvons fournir des bornes sur certaines variables, de manière à "suggérer" à l'opérateur d'élargissement de ne pas les dépasser. Ces limitations globales pourraient être, dans le cas de protocoles, un nombre de messages maximum dans un buffer, un intervalle de variation pour le contenu d'un message,.... Nous pouvons même envisager d'introduire dans cet ensemble certaines relations linéaires que nous pensons devoir être satisfaites par le programme.

Cet ensemble de limitation global est alors introduit à chaque point d'élargissement, comme valeur initiale de leur ensemble de limitation local.

Algorithme avec propagation de contraintes

Une version modifiée de l'algorithme de stabilisation des équations est donnée en figure 7.4. Les lignes contenues dans des boîtes sont les commandes ajoutées pour permettre la propagation des contraintes.

7.3 Mise en œuvre

Pour une mise en œuvre complète de l'algorithme d'analyse, nous devons détailler certains points plus précis de la construction du système d'équations et du graphe de dépendance à partir du modèle Réseau de Petri.

7.3.1 Graphe de dépendance

Pour construire et représenter le graphe de dépendance, nous pouvons considérer les solutions suivantes :

Utilisation des Bdds
: Avec les résultats des chapitres précédents, nous savons comment à partir d'un Réseau de Petri générer un modèle symbolique sous la forme de Bdds. Nous savons à partir de ce modèle symbolique calculer l'ensemble des états accessibles du modèle. Il n'est donc pas très difficile de modifier cet algorithme d'exploration pour se limiter à l'exploration du graphe de marquages du réseau.

Méthode énumérative "classique"
: Nous pouvons réutiliser directement certaines méthodes de représentations du graphe utilisée par un logiciel comme Cæsar, qui permet à partir d'un Réseau de Petri de construire le modèle correspondant, mais aussi de générer son graphe de marquages. La représentation des états et des algorithmes optimisés sont donnés dans [Gar89],

procedure *Stabilise(C :* ensemble de marquages*)*

while $C \setminus Stable \neq \emptyset$ **do**
Soit $m \in C \setminus Stable$ tel que m est minimal dans $C \setminus Stable$ pour l'ordre O_t

$Stable := Stable \cup \{m\}$
$Vide := (\Phi_m = \bot)$
if $\Phi'_m \not\sqsubseteq \Phi_m$ **then**
if $m \in W$ **then**
if $\overline{Vide}$ **then** $C_m := \{c \in C_m \mid c \text{ satisfait } \Phi_m\}$
$\Phi_m := \Phi_m \nabla_{C_m} \Phi'_m$
else
$\Phi_m := \Phi_m \sqcup \Phi'_m$
fi *-- propagation du nouveau contexte*
foreach $t = (m, m_2, \gamma, \alpha) \in \mathcal{T}$ **do**
if $\gamma \sqcap \Phi_m \neq \bot$ **then**
$\Phi'_{m_2} := \Phi'_{m_2} \sqcup \alpha(\Phi_m \sqcap \gamma)$
$Stable := Stable \setminus \{m_2\}$
if $Vide$ *ou* $C_m \neq \emptyset$ **then** $C_{m_2} := C_{m_2} \cup \alpha(C_m) \cup \alpha(\gamma)$
fi
od
if $m \notin W$ **then** $C_m := \emptyset$
$\Phi'_m := \bot$
-- Stabilisation de la sous composante contenant m
if m est la tête d'une sous composante de C **then**
Stabilise(CFC(m) - {m})
fi
fi
od

Figure 7.4: Algorithme de stabilisation avec propagation de contraintes

dont nous pouvons directement nous inspirer pour la construction de notre graphe de dépendance.

L'utilisation des BDDs semble attractive, car nous avons déjà défini les méthodes et algorithmes nécessaires pour la représentation des Réseaux de Petri avec les BDDs. Mais les algorithmes que nous voulons utiliser pour l'analyse en avant (décomposition du graphe, algorithme de stabilisation) sont tous basés sur une exploration *sommet par sommet, en profondeur d'abord*, alors que les algorithmes adaptés au BDDs travaillent sur des *ensembles* et donc en *largeur d'abord*.

Nous pourrions tenter de transformer nos algorithmes pour les adapter aux BDDs. Mais lors de l'analyse en avant, il faut aussi effectuer des transformations sur les contextes, qui sont représentés par des polyèdres. Pour effectuer un calcul sur un ensemble de sommets du graphe, il faut aussi effectuer la transformation associée à tous les contextes. Or nous ne savons pas effectuer ces transformations directement sur un ensemble de polyèdres, mais seulement polyèdre par polyèdre. Par conséquent, nous devons travailler sommet par sommet.

Nous pourrions alors conserver l'utilisation des BDDs, avec les algorithmes que nous avons actuellement. Chaque calcul effectué avec les BDDs se ferait alors sommet par sommet, ce qui est généralement inefficace pour ce mode de représentation, puisque le coût de calcul de l'image d'un sommet est du même ordre que le coût du calcul de l'image d'un ensemble de sommets.

Enfin, un dernier argument en faveur de la deuxième solution est la taille des graphes de dépendance que nous serons amené à traiter. Nous pouvons prévoir que les performances de l'algorithme seront davantage conditionnées par les transformations de contextes que par le calcul des successeurs d'un sommet. Le seul cas où la représentation du contrôle est un facteur limitatif est le cas d'un graphe de dépendance comprenant une proportion importante de branches mortes. Dans ce cas, il sera probablement plus intéressant de mettre en œuvre un algorithme "à la volée" (voir chapitres précédents) qu'une représentation symbolique.

Par conséquent, nous avons choisi de représenter le graphe de dépendance sans utiliser les BDDs, en nous inspirant de solutions proposées dans [Gar89] pour la représentation mémoire et les algorithmes de construction.

7.3.2 Représentation des variables

Les types de variables autorisés dans notre modèle sont les types booléens et entiers relatifs. La représentation des entiers relatifs ne présentent pas de problèmes particuliers, puisque nous utilisons l'enveloppe convexe d'un ensemble de points pour les représenter. Nous pouvons néanmoins utiliser le fait que ces variables sont entières, et transformer les inégalités strictes de la forme $\sum a_i x > b$ en inégalités de la forme $\sum a_i x \geq b + pgcd(a_i, b)$, qui sont vérifiées par le même ensemble de points entiers.

La représentation des variables booléennes peut se faire indépendamment des variables entières; nous aurions donc un contexte booléen et un contexte entier associés à chaque sommet

du graphe. Ce contexte booléen peut alors être représenté par un BDD, et les opérations sur les variables booléennes sont effectuées par les opérateurs BDD équivalents.

Une autre possibilité est de représenter les booléens par des variables entières, en associant à chaque opérateur des fonctions sur les entiers. Le problème est alors l'impossibilité de représenter les opérateurs $\wedge$ et $\vee$ par des expressions linéaires.

Quand la deuxième approche est possible, elle permet d'étendre aux variables booléennes une caractéristique intéressante des polyèdres, celle de permettre la représentation de *relations* entre variables.

7.4 Résultats d'analyse

La mise en œuvre de l'analyse en avant va nous permettre de déterminer certaines informations sur le système analysé, en particulier une approximation supérieure de l'ensemble des états accessibles. Comme cette analyse se fait sur un système partitionné, nous pouvons obtenir des résultats plus détaillés. En particulier, nous pouvons nous intéresser aux résultats suivants :

Contexte local à un marquage
Le résultat le plus immédiat est le contexte de chaque marquage. Pour chaque marquage, nous avons une approximation supérieure de l'ensemble des valeurs que peuvent prendre les variables en ce point. Ceci permet en particulier de déterminer une approximation inférieure de l'ensemble des marquages inaccessibles (contexte vide)

Contextes d'une transition du réseau
Un autre résultat intéressant est le calcul des contextes d'entrée et de sortie d'une transition du réseau. En effet, chaque transition du réseau correspond à un *ensemble* d'arcs de C. Soit $\{t_1, t_2, \ldots, t_n\} \subseteq \mathcal{T}$ l'ensemble des transitions correspondant à une transition t du réseau; le calcul du contexte d'entrée (resp. de sortie) d'une transition du réseau se fait alors en calculant l'union de tous les marquages d'entrée (resp. de sortie) :

Contexte d'entrée : $\bigsqcup_{t_i=(m_i,m_i'),1\leq i\leq n} \Phi_{m_i}$

Contexte de sortie : $\bigsqcup_{t_i=(m_i,m_i'),1\leq i\leq n} \Phi_{m_i'}$

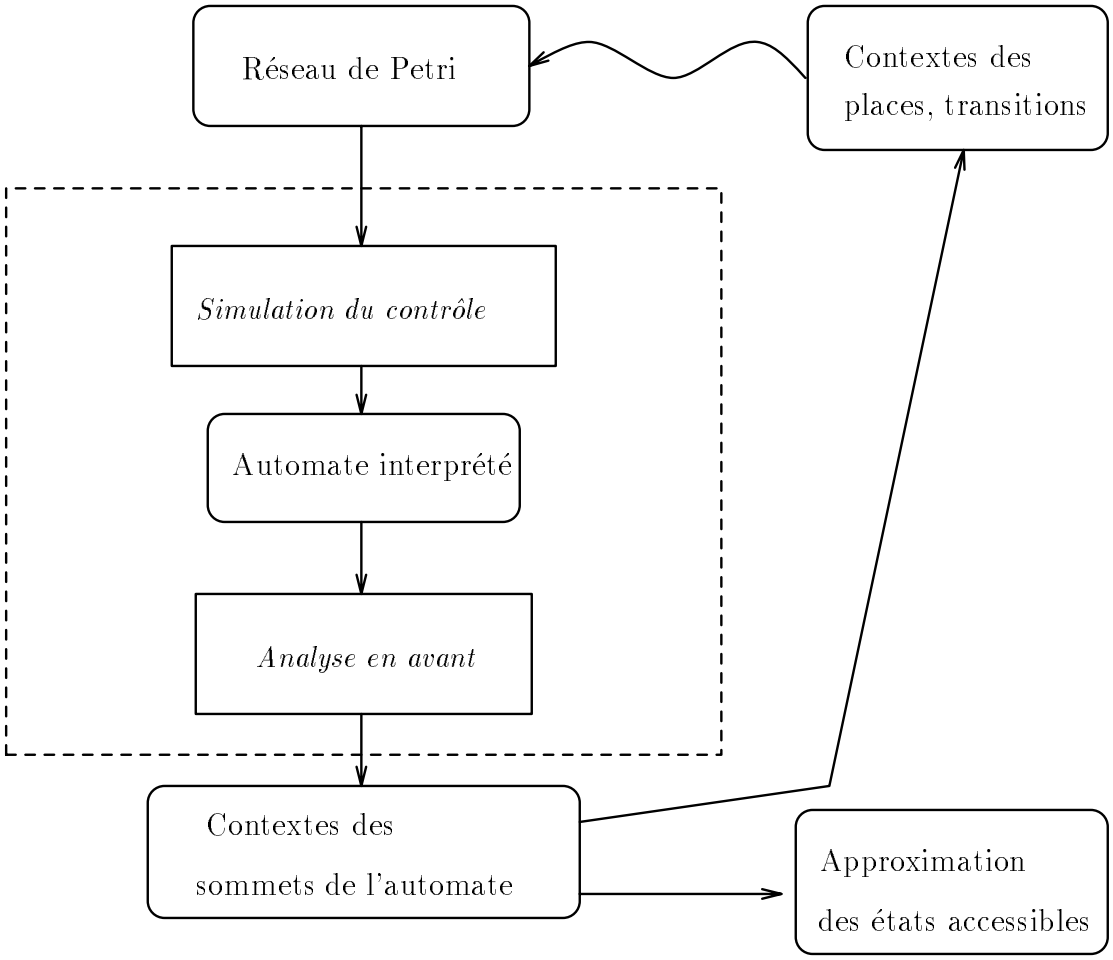
Ce calcul présente plusieurs intérêts :

- Il permet de déterminer les conditions de franchissement d'une transition. En particulier, elle permet de connaître les transitions infranchissables et donc les branches mortes du réseau.
- Si la transition est décorée par une offre de la forme "! X", ceci permet de connaître une approximation du domaine de variation de X.

Approximation supérieure de l'ensemble des états accessibles
Le calcul de l'union de tous les contextes nous donne une approximation supérieure des

domaines de valeurs que peuvent prendre les variables. Cette approximation supérieure nous permet de caractériser l'ensemble des états accessibles du modèle sous-jacent. De plus, comme nous utilisons un formalisme relationnel, ceci peut nous permettre de découvrir des invariants sous la forme d'expressions linéaires sur les variables du programme.

Le principe de notre système d'analyse en avant peut être synthétisé par la figure suivante :



7.5 Exemples d'applications

Nous allons illustrer les possibilités offertes par l'analyse en avant sur deux exemples. Les résultats d'analyse que nous présentons ci-après ont été calculés par l'outil MAGEL, qui est le résultat de la mise en œuvre des méthodes exposées dans ce document. Une présentation plus complète de MAGEL est faite dans le chapitre 8.

Exemple 7-4
Cet exemple est un protocole de lecteurs-rédacteurs. Une ressource commune (disque, mé-

moire, ...) est partagée par un ensemble de lecteurs et de rédacteurs potentiels. L'écriture et la lecture de cette ressource sont mutuellement exclusifs. De plus, seul un rédacteur peut écrire à un moment donné. Par contre tous les lecteurs qui le désirent peuvent lire en même temps. La priorité est donnée aux rédacteurs sur les lecteurs; toutes les demandes de rédaction sont comptabilisées (les rédacteurs sont en attente) et un lecteur ne peut lire que quand toutes ces demandes de rédaction sont satisfaites. Nous donnons une description des variables utilisées dans la modélisation de ce protocole :

ECRITURE $\in \{0, 1\}$
$ECRITURE = 1$ un rédacteur est en train d'écrire, 0 sinon

N_LECTEURS $\in [0 \ldots M]$
N_LECTEURS donne le nombre de lecteurs en cours de lecture

N_R_DEMANDS $\in [0 \ldots N]$
N_R_DEMANDS donne le nombre de redacteurs en attente d'écriture

Résultats de l'analyse
Les résultats suivants correspondent au cas général de M lecteurs et N rédacteurs. Ces résultats sont donnés par le calcul de l'union de tous les contextes obtenus après analyse; nous avons alors une approximation supérieure de l'ensemble des contextes atteignables.

$$P = \left\{ \begin{array}{lcl} \text{N\_R\_DEMANDS} & \geq & 0 \\ \text{N\_LECTEURS} & \geq & 0 \\ \text{ECRITURE} & \geq & 0 \\ \text{N\_R\_DEMANDS} + \text{ECRITURE} & \leq & N \\ \text{N\_LECTEURS} + M * \text{ECRITURE} & \leq & M \end{array} \right\}$$

La variable ECRITURE ne peut prendre que les valeurs 0 et 1. Par conséquent, la dernière contrainte de cet invariant montre que la variable N_LECTEURS peut prendre une valeur entre 0 et M *si et seulement si* ECRITURE est égal à 0. Donc l'exclusion mutuelle est assurée.

Une constatation intéressante sur cet exemple est le calcul d'informations sur les données à partir du contrôle : la plupart des contraintes de l'invariant n'apparaissent nulle part dans le programme. En particulier, les relations entre variables exprimées par les deux dernières contraintes apparaissent uniquement grâce à l'analyse, ne sont pas triviales à déterminer, et montrent l'intérêt d'utiliser un treillis relationnel pour une analyse sémantique.

■

Exemple 7-5

Cet exemple est un moniteur de tâches. Ce moniteur gère plusieurs classes de tâches, chaque classe C répond aux caractéristiques suivantes :

- une classe correspond à un travail spécifique, dont la durée est fixe pour toutes les tâches d'une même classe,

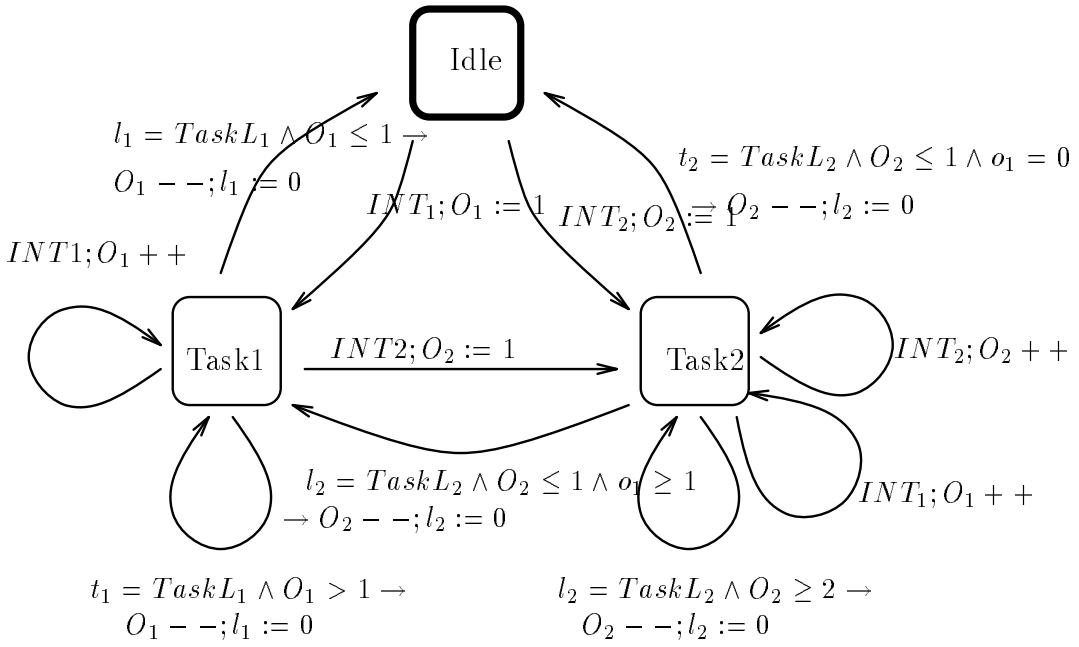


Figure 7.5: Moniteur de tâches

- à chaque classe est associée une interruption particulière, ces interruptions sont générées à des intervalles de temps dont la durée minimale est constante

A chaque classe C, nous associons un couple de valeurs $(TaskL_C, IntL_C)$ qui donnent respectivement la durée de la tâche et l'intervalle INT_C entre deux interruptions. Enfin, un ordre est défini entre les classes : une classe d'ordre n a priorité sur toute classe d'ordre strictement inférieur à n. Son interruption est prioritaire et peut interrompre l'exécution d'une tâche de classe inférieure.

Pour comptabiliser le nombre de tâches en attente, nous associons à chaque classe C une variable O_C. Chaque nouvelle interruption va incrémenter cette variable, chaque tâche terminée va la décrémenter.

La figure 7.5 présente l'automate d'un moniteur de deux classes de tâches, pour lequel les tâches de classe 2 sont prioritaires.

Nous associons à ce moniteur un générateur d'interruptions qui se chargera de produire les interruptions de chaque classe selon les intervalles de temps fixés. Pour modéliser la progression du temps, nous utilisons une variable T_C pour chaque classe C. Toutes les variables T_C progressent de façon synchrone. T_C est remise à zéro lors de la génération d'une interruption de la classe C.

Le générateur d'interruptions est modélisé par la figure 7.6.

Nous voulons vérifier que si l'intervalle entre deux interruptions de la classe 2 est plus grand que la durée d'exécution de la tâche de la classe 2 ($IntL_2 > TaskL_2$), alors il n'y a jamais de tâche de la classe 2 en attente ($0 \leq O_2 \leq 1$).

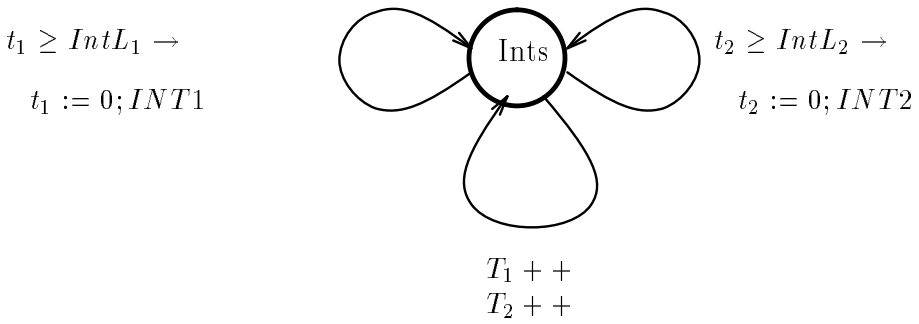


Figure 7.6: Générateur d'interruptions

Une remarque importante est que les variables du générateur d'interruptions ne sont pas bornées, une analyse sans extrapolations ne peut pas terminer.

Nous avons écrit cet exemple sous la forme d'un programme LOTOS, puis nous avons utilisé le compilateur CÆSAR pour produire un Réseau de Petri interprété sur lequel nous avons appliqué l'outil MAGEL. Nous avons essayé l'exemple pour diverses combinaisons des valeurs $TaskL_C$ et $IntL_C$. Parmi ces combinaisons, nous nous intéressons à celle vérifiant ($IntL_2 > TaskL_2$).

L'invariant global obtenu s'avère inintéressant, puisque nous obtenons $\mathbb{Q}^n$ comme résultat de l'analyse. Pour pouvoir obtenir des résultats significatifs, nous allons nous intéresser aux informations calculées pour chaque transition du réseau. En effet, la seule affectation capable de faire croître la valeur de O_2 est liée à l'interruption INT_2. Nous pouvons donc particulariser l'analyse en repérant dans le Réseau de Petri les transitions correspondant à l'action INT_2. Nous pouvons trouver trois transitions de ce type, deux correspondent à une première interruption INT_2 et se contentent d'initialiser O_2 à 1. Pour la troisième transition que nous notons t et qui correspond à l'incrémentation de O_2, nous obtenons les résultats suivants :

Marquages sources

$$P = \left\{ \begin{array}{rcl} 0 & \leq & l_1 \\ 0 & \leq & l_2 \leq TaskL_2 \\ 0 & \leq & l_1 \leq TaskL_1 \\ t_2 & = & l_2 \\ 0 & \leq & O_1 \\ O_2 & = & 1 \end{array} \right\}$$

Marquages cibles

$$P' = \emptyset$$

Ces résultats montrent que :

- quand nous arrivons à cette transition, nous avons $O_2 = 1$

- le franchissement de cette transition n'a jamais été autorisé, puisque le contexte des marquages successeurs est vide.

Si nous examinons plus attentivement le réseau construit, nous pouvons voir que cette transition a été augmentée de la garde $t_2 \geq IntL_2$ lors de la composition du moniteur et du générateur d'interruptions. Or le contexte des marquages sources nous indique que $0 \leq t_2 \leq TaskL_2$. Comme nous avons pris comme hypothèse que $TasKL_2 < IntL_2$ alors la garde $t_2 \geq IntL_2$ ne peut pas être satisfaite.

■

7.6 Conclusion

Nous avons présenté la construction d'un modèle basé sur les polyèdres, à partir de notre modèle Réseau de Petri. Ce modèle peut être considéré comme *semi symbolique*, puisque le contrôle y est représenté explicitement, et les données symboliquement.

L'utilisation de ce modèle est particulièrement intéressante dans le cadre d'algorithmes d'analyse, qui vont permettre de déterminer certaines propriétés globales ou locales à certains points de contrôle. En particulier, l'analyse en avant nous permet de calculer des approximations supérieures d'invariants du programme. Ceci rentre donc dans le cadre de la vérification partielle de propriétés; si nous exprimons certaines propriétés en terme de contraintes linéaires sur les variables du programme, nous pourrons démontrer la validité d'une propriété (si le polyèdre correspondant est contenu dans l'approximation supérieure d'un invariant), mais pas son invalidité.

Chapitre 8

Mise en œuvre

Nous décrivons maintenant l'ensemble des applications qui ont été réalisées et le contexte dans lequel elles sont venues s'intégrer. Cette mise en œuvre a été effectuée en deux volets distincts, mais présentant de larges parties communes. Le premier volet concerne l'outil MAGEL, qui permet la minimisation, comparaison ou analyse de programmes LOTOS via le Réseau de Petri construit par CÆSAR. Le deuxième volet concerne l'intégration de certaines parties de MAGEL dans l'outil de vérification ALDÉBARAN.

Nous commençons par présenter l'architecture et les choix de conception faits pour l'outil MAGEL. Nous présentons ensuite les modifications nécessaires à l'adaptation des bibliothèques aux formalismes d'entrée d'ALDÉBARAN.

8.1 L'outil MAGEL

L'outil MAGEL est un outil de minimisation, de comparaison ou d'analyse de programmes à base de processus communicants. Dans sa version actuelle, il est construit de manière à recevoir en entrée des Réseaux de Petri produits par le compilateur CÆSAR à partir de spécifications LOTOS.

8.1.1 Architecture

L'outil MAGEL se divise en deux parties; une partie de génération de modèle minimal réalisée à l'aide de BDDs, une partie d'analyse sémantique utilisant comme représentation des polyèdres. L'architecture globale de MAGEL est donné en figure 8.1.

8.1.2 Décomposition modulaire

Le fonctionnement de MAGEL repose sur l'utilisation de certaines bibliothèques de base, dont

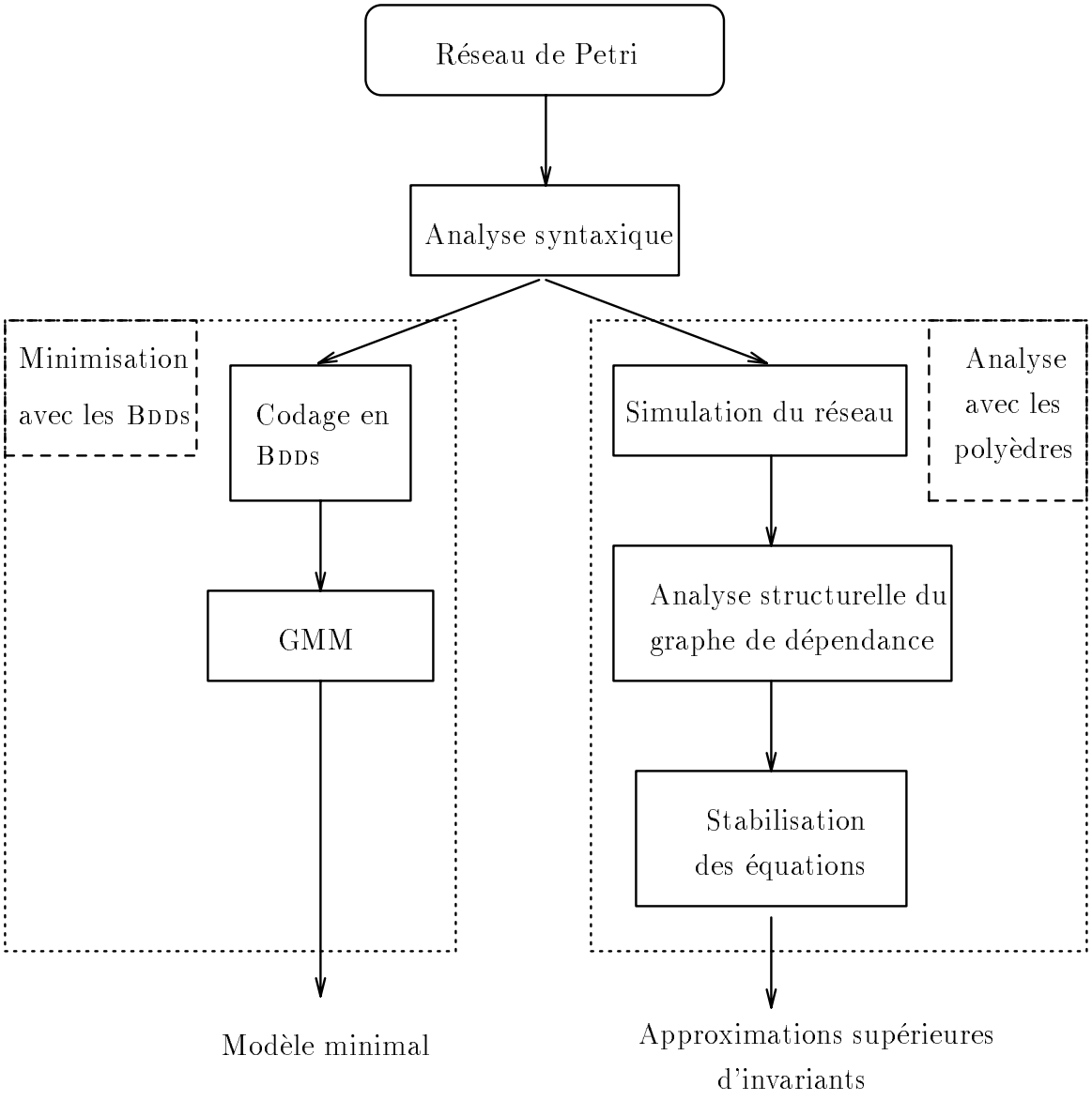


Figure 8.1: L'outil MAGEL

nous décrivons les principales. Toutes ces bibliothèques ont été développées en C++.

Les deux premières bibliothèques implémentent les formalismes de représentation symbolique que nous avons considérés :

Bibliothèque BDDs (7000 lignes) :

Cette bibliothèque a été initialement conçue par Christophe Ratel [Rat92]. Elle implémente une version de BDDs particulières, les TDGs (Typed Decision Diagrams) [Bil87] : il s'agit de BDDs dont chaque arc est décoré par un signe + ou - qui indique l'interprétation (f ou $\neg f$) que l'on doit donner à la sous formule correspondante. Dans [BM88], il est montré comment conserver la canonicité d'une telle forme de BDDs. Les gains en pratique sont en terme de taille de la représentation, les TDGs étant toujours de taille inférieure ou égale à celle des BDDs. D'autre part, la négation d'une formule s'effectue en temps constant (par changement du signe au niveau de la racine), alors que cet opérateur est linéaire dans le cas des BDDs "ordinaires". Dans la suite, nous continuerons néanmoins à utiliser la notation BDD pour désigner les BDDs ou les TDGs. Pour les besoins de notre implémentation, nous avons étendu cette bibliothèque avec quelques opérateurs et amélioré certains opérateurs clés, notamment au niveau de la gestion mémoire.

Bibliothèque *polyèdres* (11000 lignes):
Pour réaliser cette bibliothèque, nous avons utilisé une implémentation existante d'un algorithme qui permet le calcul du système de contraintes minimal d'un polyèdre à partir de son système générateur (et vice versa). Cet algorithme a été initialement conçu par [Che68], puis a ensuite été amélioré et implémenté par [Ver92]. C'est l'implémentation de Hervé Leverge qui est à la base de notre réalisation. L'ensemble de la bibliothèque a été conçue suivant une hiérarchie de classes C++, implémentant tous les éléments nécessaires, de vecteur à polyèdre, en passant par des matrices génériques, matrices de contraintes, matrices d'éléments générateurs.

En dehors de l'utilisation de ces bibliothèques symboliques, nous avons réalisé deux bibliothèque de vérification et d'analyse, et une bibliothèque d'analyse syntaxique du Réseau de Petri de CÆSAR. Une idée directrice lors de la réalisation de ces bibliothèques a été de les rendre les plus indépendantes possible du mode de représentation symbolique choisi et de l'outil dans lequel la bibliothèque s'intègre.

Bibliothèque *Analyse syntaxique* (1000 lignes)
Cette bibliothèque permet la lecture et l'analyse syntaxique d'un Réseau de Petri interprété. Elle est actuellement dédiée à l'analyse des réseaux produits par CÆSAR. Elle construit pendant l'analyse un arbre abstrait qui est destiné à être utilisé indifféremment par le module de minimisation ou le module d'analyse sémantique.

Bibliothèque *génération de modèle minimal* (3000 lignes) :
Cette bibliothèque récupère un modèle symbolique au travers d'une interface indépendante du mode de représentation choisi. Cette interface décrit un modèle abstrait sous

la forme d'une ensemble de classes C++ qui héritent des classes du mode de représentation choisi. Les opérations nécessaires pour la génération de modèle minimal comme l'intersection, la complémentation et le calcul de la fonction *pre* sur un ensemble d'états sont définis au travers de cette interface en terme d'opérations sur la représentation symbolique choisie. A partir de ce modèle abstrait, la bibliothèque fournit en sortie un modèle minimal au travers de la même interface.

Bibliothèque *Analyse Sémantique* **(3000 lignes)**
Cette bibliothèque construit à partir de l'arbre abstrait fourni par la phase d'analyse syntaxique un ensemble d'équations et leur graphe de dépendance. A partir de ce graphe et ces équations, un modèle abstrait est construit. Comme dans le cas précédent, ce modèle est décrit par une interface de classes C++ indépendante du mode de représentation symbolique choisi. Enfin, ce module délivre en sortie les résultats obtenus après résolution du système d'équations, au travers de la même interface.

Néanmoins, dans l'état actuel de l'outil, cette bibliothèque reste fortement dépendante du formalisme d'entrée Réseau de Petri utilisé. D'autre part, la seule utilisation qui en a été faite pour l'instant est liée aux polyèdres. Un petit travail d'adaptation et de mise au point reste nécessaire pour passer à un autre formalisme de représentation symbolique.

La réunion de ces bibliothèques avec quelques modules utilitaires nous donne l'outil MAGEL, qui correspond donc environ à 30000 lignes de C++.

Chacune de ces bibliothèques peut être utilisée indépendamment des autres dans d'autres applications. C'est le cas en particulier de la bibliothèque BDD, dont la version originale est utilisée dans l'outil LESAR [Rat92] et dont la version courante, utilisée dans cette application, est aussi intégrée à un prototype de vérification symbolique de formules CTL à l'aide d'abstractions [Loi94], à un outil de vérification de propriétés sur des automates booléens BAC[Hal94] et dans le compilateur ARGOS pour la résolution de systèmes d'équations booléennes [Jou94].

La bibliothèque polyèdre que nous avons développée est aussi utilisée dans un outil d'analyse approchée de systèmes hybrides [HPR94].

Enfin, les bibliothèques BDD et génération de modèle minimal sont partagées par les outils MAGEL et ALDÉBARAN.

Après cette description de l'outil MAGEL, nous allons nous intéresser aux Réseaux de Petri tels qu'ils sont produits par CÆSAR, et en particulier aux différences existantes entre les réseaux de CÆSAR et le modèle réseau que nous avons considéré jusqu'ici.

8.2 les Réseaux de Petri de CÆSAR

Les Réseaux de Petri produits par CÆSAR présentent un certain nombre de différences par rapport au formalisme Réseau de Petri présenté au chapitre 1. Certaines de ces différences

sont liés à l'utilisation du langage LOTOS comme langage d'entrée, d'autres sont des opérateurs ou combinaison d'opérateurs supplémentaires par rapport au modèle que nous utilisons.

Nous présentons ici l'ensemble de ces différences, et les choix que nous avons faits pour leur traitement.

8.2.1 Type des variables

Le langage LOTOS est étendu pour sa partie donnée par le langage ACT-ONE, qui est un langage de description de *types abstraits algébriques*. L'utilisation des types abstraits permet de définir des structures de données comme des listes, des files, des tableaux. Nous ne pouvons pas représenter directement ces structures dans les formalismes de représentation symbolique que nous avons choisis.

Une méthode de traitement de ces types abstraits dans le cadre de l'analyse avec les polyèdres est présenté dans [Hal79, p. 93]. Il propose d'associer à un type abstrait une interprétation numérique permettant d'exprimer chaque objet d'un type T sous la forme d'un ensemble de valeurs numériques et chaque opération sous la forme de relations linéaires sur ces valeurs numériques. Cette association se fait par la construction d'une fonction d'abstraction α qui permet de passer du type T à sa représentation numérique et d'une fonction de concrétisation γ qui permet la transformation inverse. En général, ces fonctions définissent une interprétation *approchée* des objets d'un type donné. Cette méthode est donc difficilement applicable à la génération de modèle minimal, pour laquelle nous voulons rester dans un cadre exact.

Dans la version courante de MAGEL, nous avons choisi de limiter les types de variable du modèle Réseau de Petri que nous utilisons aux entiers et aux booléens. Tout réseau de CÆSAR proposé en entrée de MAGEL contenant d'autres types de données sera donc rejeté. La construction de ces deux fonctions de construction d'une abstraction numérique d'un type abstrait restent donc à la charge de l'utilisateur, ce qui revient dans ce cas à réécrire un type abstrait T comme un ensemble de valeurs numériques et à modifier les appels aux opérateurs de T pour les transformer en expressions linéaires sur les variables de type T.

Exemple 8-1
Considérons la définition du type abstrait suivant, qui correspond aux entier modulo 4.

```
(* Type entier dont les opérateurs + et - sont redéfinis modulo 4 *)

type MODULO_4 is NATURAL
      opns
           _{+}_
           _{-}_ : NAT, NAT -> NAT
           eqns
              forall X, Y : NAT
              ofsort NAT
              (X + Y) lt 4 => X {+} Y = X + Y;
              (X + Y) ge 4 => X {+} Y = X + Y - 4;
              X lt Y => X {-} Y = (X + 4) - Y;
              X ge Y => X {-} Y = X - Y;
endtype
```

Dans ce cas, toutes les occurrences dans un programme LOTOS de l'opérateur $\{+\}$ nécessitent la duplication de la ligne pour tenir compte des deux équations du $\{+\}$: La ligne `[X eq Y] -> Proc(X + 2)]` est alors remplacée par

```
([X eq Y] and ((X + 2) lt 4)] -> Proc(X+2)
[]
([X eq Y] and ((X + 2) ge 4)] -> Proc(X-2)
```

■

Exemple 8-2
Un ensemble (resp. une pile, une liste, ...) peut être approché par son cardinal (resp. sa hauteur, sa longueur,...). Dans le cas d'un ensemble d'entiers, nous pouvons ajouter comme informations un intervalle le contenant. ■

8.2.2 ε-transitions

Une particularité importante des réseaux de CÆSAR est la présence de transitions spéciales, les ε-transitions. Ces transitions ont été introduites dans les réseaux de CÆSAR pour faciliter leur construction, notamment au niveau de la composition de sous réseaux. Elle représentent une évolution interne au réseau, mais ne correspondent à aucun comportement, observable ou non, du système modélisé par le réseau; le franchissement de ces transitions ne doit pas produire d'arc dans le graphe correspondant au réseau.

Nous allons distinguer le traitement des ε-transitions suivant l'algorithme que nous voulons utiliser.

Analyse avec les polyèdres

En ce qui concerne l'analyse approchée avec les polyèdres, nous avons choisi délibérément d'ignorer le caractère spécial de ces transitions. Les ε-transitions seront donc considérées comme les autres transitions lors de la génération du système d'équations et du graphe de dépendance de ces équations. Ce choix s'appuie sur les arguments suivants :

- Les ε-transitions sont souvent des transitions décorées par des affectations de variables ou par des gardes. Ces ε-transitions permettent en quelque sorte de factoriser certaines des modifications de contexte qui nous intéressent lors de l'analyse avec les polyèdres.
- Garder les ε-transitions permet de retrouver aisément la correspondance entre états du graphe de dépendance que nous construisons et places dans le réseau. Nous voulons pouvoir appliquer l'analyse à un niveau local, c'est à dire au niveau des places et transitions du réseau. Si nous construisons notre graphe de dépendance en éliminant au passage les ε-transitions, la correspondance entre des résultats d'analyse locaux et des places du réseau devient plus difficile à maintenir.

Génération de Modèle Minimal avec les BDDs

Comme les ε-transitions ne doivent pas apparaître dans le graphe du réseau, elles n'apparaissent pas non plus dans le modèle minimal. Une idée pour le traitement de ces ε-transitions est d'appliquer une fermeture transitive de la relation ε au niveau du graphe correspondant au réseau. Ceci revient à calculer la relation de transition entre états (voir 1 suivant la nouvelle règle :

$$\frac{\exists a \neq \varepsilon, \exists \langle M\ ,\ C\rangle, \langle M_1\ ,\ C_1\rangle \xrightarrow{\varepsilon^*} \langle M\ ,\ C\rangle \wedge \langle M\ ,\ C\rangle \xrightarrow{G} \langle M_2\ ,\ C_2\rangle}{\langle M_1\ ,\ C_1\rangle \xrightarrow{G} \langle M_2\ ,\ C_2\rangle} \qquad \text{[E1]}$$

Cette méthode s'apparente au calcul d'une forme *pré normale* du système, qui est utilisée lors du calcul de raffinement de partition pour l'équivalence observationnelle ou la bisimulation $\tau^* a$ (voir le chapitre 5).

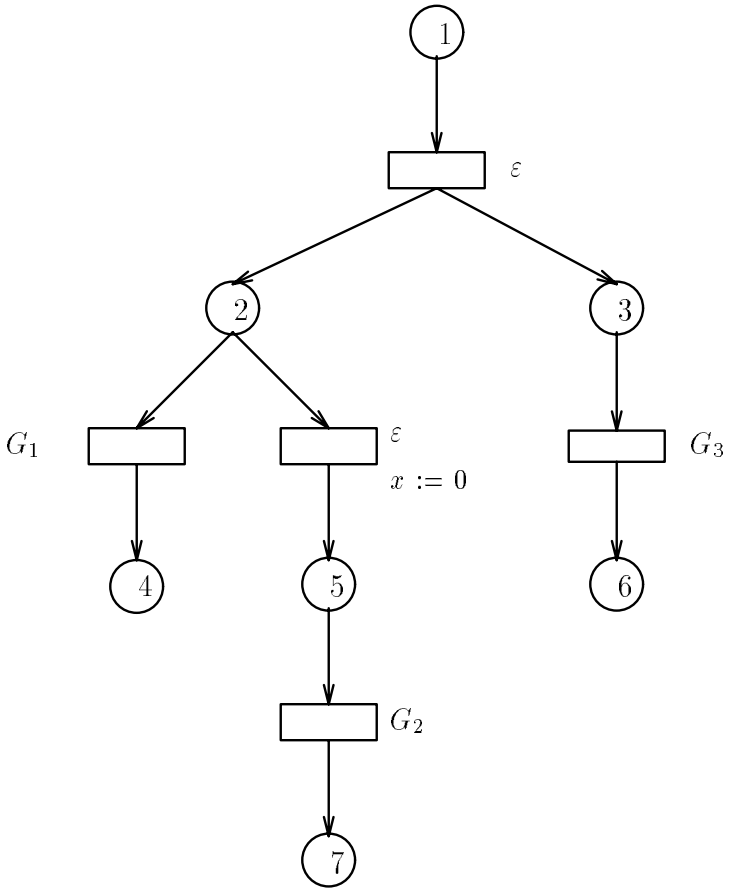
Néanmoins, cette fermeture transitive n'est pas correcte du point de vue de la construction du graphe correspondant au réseau. En effet, le graphe produit par cette méthode peut ne pas être équivalent pour la bisimulation forte au graphe correspondant au comportement décrit par le réseau, comme le montre l'exemple suivant, issu de [Gar89, p.84] :

Exemple 8-3

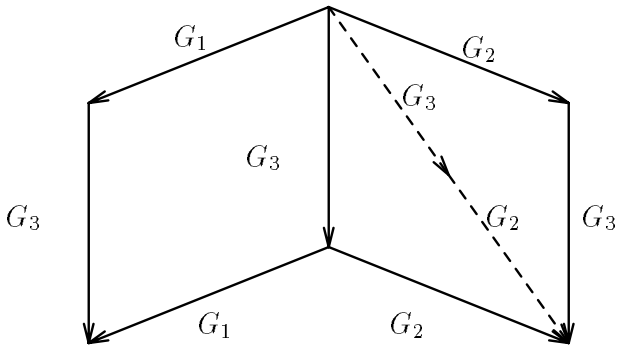
Nous considérons le système correspondant à la composition parallèle asynchrone de ces deux processus. L'expression LOTOS qui correspond à ce système est la suivante :

$(G_1; \mathbf{stop}[](\mathbf{let} X : NAT = 0\ \mathbf{in}\ G_2; \mathbf{stop}))|||G_3; \mathbf{stop}$

Le réseau construit par CÆSAR pour la mise en parallèle sans synchronisation de ces deux processus est donné par la figure suivante.



Si nous appliquons la règle $E1$ pour la génération du graphe de ce réseau, nous obtenons le résultat suivant :



D'après la sémantique de LOTOS, l'exécution de l'action G_3 n'a pas d'influence sur l'exécution des actions G_1 et G_2. Or le graphe obtenu avec l'application de la règle $E1$ contient un chemin où après l'exécution de G_3, seule l'exécution de G_2 est possible. En fait, l'évolution

qui correspond à ce comportement est la suivante :

$$\{1\} \xrightarrow{\varepsilon} \{2,3\} \xrightarrow{\varepsilon} \{4,3\} \xrightarrow{G_3} \{4,7\}$$

A partir du marquage $\{4,7\}$, la seule évolution possible est par la transition étiquetée G_2. ■

L'exécution d'ε-transitions ne doit pas provoquer de différence de comportements au niveau du graphe. En particulier, le franchissement d'une chaîne d'ε-transitions dans le réseau n'a de sens que si elle est immédiatement suivie du franchissement d'une transition significative (étiquetée par τ ou par une étiquette visible).

Cette contrainte a une conséquence immédiate : l'exécution d'une chaîne d'ε-transitions ne peut être interrompue par aucune autre transition évoluant de manière asynchrone dans une autre unité. Nous voyons tout de suite qu'elle n'est pas respectée dans le cas de l'exemple 8-3, car l'exécution de la transition $\{2,3\} \xrightarrow{\varepsilon} \{4,3\}$ n'est pas suivie du franchissement d'une transition observable dans la même unité, ce qui a provoqué un comportement observable dans le graphe du réseau.

Par conséquent, il est plus correct d'effectuer le calcul de la fermeture transitive par ε^* au niveau du réseau. Cette fermeture transitive pourrait se faire à priori, avant construction de la représentation symbolique du réseau. Néanmoins, il est préférable pour des raisons d'efficacité d'effectuer cette transformation lors de la construction de la représentation symbolique de chaque transition du réseau.

8.2.3 Traitement des offres

Le langage LOTOS permet la communication de valeurs lors d'un rendez vous. Ces communications de valeur sont regroupées sous le terme d'*offres* et correspondent dans le programme LOTOS à des *émissions de valeur* de la forme "! V" et des *réceptions* de la forme "? X : S" où V est une expression de valeur, X une variable et S une sorte (voir chapitre 1).

Chaque transition du réseau produit par CÆSAR peut donc être décorée par une liste d'offres. Ces offres ont une syntaxe standard liée à la méthode de construction suivante : à la fin de la génération du réseau, chaque offre de réception "? X : S" qui n'a pas été résolue par une synchronisation avec une offre d'émission est remplacée par l'offre "! X" et l'itérateur "**for** X **among** S" est introduit dans l'action de la transition. Par conséquent, seules des offres de la forme "! V" sont présentes dans le réseau. Une transition du Réseau de Petri de CÆSAR est donc un tuple $(\widetilde{Q_i}, \widetilde{Q_o}, G, A, \widetilde{O})$, où $\widetilde{O}$ est une liste d'offres de la forme "! V".

La présence d'offres dans les transitions nécessite d'introduire certains changements dans l'algorithme de génération de modèle minimal. En effet, si nous considérons le système de transitions étiquetées $S = (Q, A, T, init)$ correspondant à ce réseau, l'ensemble A des actions de S est construit à partir de l'ensemble des portes des transitions du réseau, mais aussi à partir des offres de ces transitions, grâce à la fonction *eval_label* suivante : soit une transition $t = (\widetilde{Q_i}, \widetilde{Q_o}, G, A, \widetilde{O})$ telle que $\widetilde{O} = \{!V_1, \ldots, !V_n\}$, soit C le contexte obtenu *après* exécution

de l'action A,

$$eval\_label(G, <!V_1, \ldots, !V_n>, C) = \begin{cases} \varepsilon \text{ si } G = \varepsilon \\ i \text{ si } G \text{ est cachée} \\ \text{“}G!eval(V_1, C) \ldots !eval(V_n, C)\text{”} \text{sinon} \end{cases}$$

Exemple 8-4
Soit une transition t de porte G, décorée par la liste d'offres “!B !N ” où $B \in \mathbb{B}$ et $N \in [1..2]$. Alors les transitions du système de transitions étiquetées S correspondant à t pourront prendre comme étiquette[1] une des valeurs de l'ensemble $\{\text{“}G\ !TRUE\ !1\text{”}, \text{“}G\ !TRUE\ !2\text{”}, \text{“}G\ !FALSE\ !1\text{”}, \text{“}G\ !FALSE\ !2\text{”}\}$.

■

Si il existe des transitions du réseau ayant une liste d'offres non vide, alors l'ensemble A des actions (et par conséquent l'ensemble de langages Λ sur lequel nous définissons nos relations de bisimulation) ne peut pas être déduit par une simple collection des portes des transitions du réseau.

Cette limitation ne pose pas de problèmes dans le cadre de l'analyse du réseau avec les polyèdres, puisque nous ne considérons pas les labels des transitions. Par contre, nous avons besoin de connaître l'ensemble A des actions visibles pour l'algorithme de génération de modèle minimal. Pour déterminer cet ensemble, nous avons deux solutions :

Production *à priori* de tous les labels possibles :
Comme le domaine des variables que nous considérons est fini, nous pouvons générer toutes les combinaisons possibles d'offres pour toutes les transitions ayant une liste d'offres non vides. Un inconvénient majeur de cette méthode est le cardinal de l'ensemble A des actions ainsi construit; si nous considérons pour les naturels l'intervalle $[0\ldots255]$ comme domaine, une simple transition ayant une liste d'offres de la forme “! X ! Y” et donc une action contenant “**for** X,Y **among** NAT” va provoquer la création de 256^2 labels. En général, cette solution se révèle praticable seulement si les offres sont définies par des valeurs constantes ou des variables booléennes.

Calcul *à priori* des états accessibles
Lors du calcul des états accessibles, le problème de connaître l'ensemble A des actions ne se pose pas. Une stratégie possible est donc de calculer l'ensemble des états accessibles du modèle, puis de déduire de cet ensemble les combinaisons d'offres possibles pour chaque transition du réseau ayant une liste d'offres non vide. À partir de ces informations, il devient possible de construire un ensemble A d'actions et d'appliquer l'algorithme de génération de modèle minimal sans considérer un nombre astronomique d'actions.

Nous avons vu dans le chapitre 3 que le calcul des états accessibles *avant* de commencer la génération de modèle minimal permettait en général d'obtenir de meilleures performances. Le calcul à priori des états accessibles pour déterminer l'ensemble A des

[1] en fonction des valeurs des variables lors du franchissement de ces transitions

actions est donc compatible avec de bonnes performances de l'outil pour la minimisation.

8.3 Intégration de MAGEL à ALDÉBARAN

Nous avons intégré une partie des bibliothèques de MAGEL dans l'outil de vérification ALDÉBARAN. En particulier, nous avons adapté les bibliothèques de gestion des BDDs et de génération de modèle minimal pour leur utilisation avec les formalismes d'entrée d'ALDÉBARAN. Cette intégration a été faite dans le but de fournir une alternative symbolique aux méthodes énumératives d'ALDÉBARAN.

8.3.1 ALDÉBARAN

ALDÉBARAN [Fer88] est un outil de vérification permettant la *minimisation* et la *comparaison* de systèmes de transitions étiquetées ou de systèmes de processus communicants. Ces opérations sont basées sur deux générations d'algorithmes :

- un algorithme de *raffinement de partition* basé sur la solution de Paige et Tarjan [PT87]. L'implémentation de cet algorithme et son adaptation à diverses relations d'équivalence constituent la première génération d'ALDÉBARAN. En particulier, les relations implémentées sont la bisimulation forte, l'équivalence observationnelle, la bisimulation τ^*a et l'équivalence de sûreté. Les systèmes à minimiser ou comparer sont des systèmes de transitions étiquetées fournis en extension selon le format ALDÉBARAN [Fer88]

  Enfin, lorsque deux systèmes comparés ne sont pas identiques, un diagnostic est fourni sous la forme de séquences d'exécution des deux systèmes, menant à un couple d'états ne vérifiant pas la relation.

- un algorithme de *vérification à la volée* [Mou92]. Cet algorithme permet de comparer deux systèmes par exploration *à la volée* de leur produit synchrone. Il a été adapté à certaines relations déjà présentes dans ALDÉBARAN comme la bisimulation forte, la bisimulation τ^*a, l'équivalence de sûreté et étendus à d'autres relations comme la bisimulation de délai [NMV90], au préordre de sûreté et à la bisimulation de branchement[2]. De plus cet algorithme a été implémenté pour des systèmes de transitions étiquetées mais aussi pour des systèmes de processus communicants que nous présenterons plus loin.

  Enfin, un mécanisme de construction de diagnostic a été adapté à cette méthode de vérification.

ALDÉBARAN symbolique

[2] si un des deux systèmes n'a pas de transitions étiquetées τ

L'intégration des modules de génération de modèle minimal et de gestion des BDDs a permis de construire une version *symbolique* d'ALDÉBARAN. En particulier, cette adaptation permet de *minimiser* des systèmes pour les équivalences de bisimulation forte, $\tau^* a$ et de branchement, et de *comparer* des systèmes pour les mêmes équivalences, ainsi que pour les préordres de simulation forte et de sûreté[3] . Les équivalences observationnelle, de sûreté, de délai et par modèle d'acceptation n'ont pas été considérées; les équivalences déjà implémentées permettent une couverture suffisante des besoins pour la comparaison et la minimisation de systèmes.

Le plus gros travail dans cette adaptation consiste à accepter en entrée de l'outil les formalismes d'entrée existants d'ALDÉBARAN. Pour cela, nous allons d'abord présenter de manière plus précise les formalismes.

8.3.2 Formalismes d'entrée d'ALDÉBARAN

Les formalismes d'entrée d'ALDÉBARAN sont de deux sortes :

- Les *systèmes de transitions étiquetées*
- Les *systèmes de processus communicants*

L'utilisation de ces formalismes d'entrées, qui se situent à un niveau plus bas dans la chaîne de compilation de CÆSAR, permet d'étendre la gamme des exemples que nous pouvons traiter et de passer à des programmes LOTOS qui utilisent des types autres que booléens et entiers. Le prix à payer pour ceci est que ces formalismes sont sujets au problème de l'explosion des états, puisque les données sont traitées en extension, et que dans le cas des systèmes de transitions étiquetées, le parallélisme des processus LOTOS est modélisé explicitement, par entrelacement des actions.

8.3.3 le formalisme *système de transitions étiquetées*

Le modèle d'entrée de base d'ALDÉBARAN est un système de transitions étiquetées qui est en général le résultat de la compilation d'un programme LOTOS par CÆSAR. La représentation de ce système de transitions étiquetées par des BDDs passe par la transformation de ce système de transitions étiquetées en un Réseau de Petri. Le principe de cette transformation est décrit plus loin, pour les systèmes de processus communicants.

8.3.4 le formalisme *système de processus communicants*

Ce modèle est constitué d'un ensemble de systèmes de transitions étiquetées $S_i = (Q_i, A_i, T_i, init_i)$ et d'une expression de composition F qui décrit la composition parallèle et les synchronisations de ces automates. Ce modèle est généralement obtenu par découpage d'un programme LOTOS selon ses différents processus. Chaque processus est alors

[3] si un des systèmes à comparer est déterministe

compilé séparément en un système de transitions étiquetées. Une expression de composition de ces systèmes de transitions étiquetées est extraite à partir des opérateurs de composition parallèle présents dans le programme LOTOS. Cette expression de composition a donc une structure d'arbre, qui reflète l'imbrication des processus les uns dans les autres.

Ce formalisme offre des possibilités intéressante du point de vue de la minimisation. En effet, l'ensemble des relation d'équivalence utilisées par ALDÉBARAN pour la comparaison ou la minimisation d'automates sont des *congruences* pour l'opérateur de mise en parallèle : si on considère deux systèmes S_1 et S_2 et une congruence $\sim_\Lambda$, il est équivalent de composer S_1 et S_2, *puis* d'appliquer un algorithme de raffinement de partition avec $\sim_\Lambda$ sur le résultat, ou de d'abord minimiser S_1 et S_2 par rapport à $\sim_\Lambda$, *puis* de les composer et d'appliquer le même algorithme de raffinement de partition.

Lorsque nous travaillons avec des systèmes de processus communicants, il est donc possible de minimiser a priori chacune des composantes du système pour l'équivalence qu'on veut utiliser par la suite. Ceci permet de diminuer la taille du modèle global correspondant et donc de traiter des exemples plus importants.

La contrepartie est liée à la taille du modèle de chaque composante. En effet, ces systèmes de transitions étiquetées sont sujets au problème de l'explosion des états et sont en général beaucoup plus gros que le réseau de Petri correspondant. Il peut même arriver que le système de transitions étiquetées d'un processus soit plus gros que le système de transitions étiquetées global. Ce phénomène est dû au relâchement de contraintes de synchronisation, un processus isolé est donc plus "libre" (plus général) que le même processus synchronisé avec d'autres processus. Il est possible alors d'effectuer la génération d'une composante sous contraintes pour limiter la taille de son modèle. Ces contraintes sont exprimées sous la forme d'un processus donnant une abstraction de l'environnement.

8.3.5 Modification du formalisme d'entrée

Ces nouveaux formalismes peuvent être aisément utilisés comme formalismes d'entrée de notre implémentation. Pour cela, nous définissons une méthode de codage de chacun de ces formalismes en BDDs.

Codage d'un système de transitions étiquetées

Un système de transitions étiquetées $S = (Q, A, T, init)$ peut être considéré comme un Réseau de Petri $(\mathcal{Q}, \mathcal{U}_\prime, \mathcal{T}, \mathcal{G}, \mathcal{V})$ où $\mathcal{U}_\prime$ ne contient pas de sous unités, $\mathcal{Q} = Q$, $\mathcal{T} = T$, $\mathcal{G} = A$ et $\mathcal{V} = \emptyset$.

Le codage d'un système de transitions étiquetées se fait alors directement avec les méthodes de codage définies au chapitre 3.

Codage d'un système de processus communicants

Le modèle système de processus communicants peut être aisément traduit en un Réseau de

Outil	Buts	Formalismes d'entrée	Méthodes de composition
ALDÉBARAN symbolique	Comparaison (BDD) GMM (BDD)	STE système de processus communicants	entrelacée simultanée mixte
MAGEL	GMM (BDD) Analyse (polyèdre)	Réseau de Petri interprété	entrelacée

Table 8.1: Différences d'implémentation

Petri; chaque composante du système est représentée par une unité, les différents niveaux de composition parallèle étant reflétés par une hiérarchie entre les unités.

Dans la version actuelle d'ALDÉBARAN, le traitement d'un tel système se fait différemment. le codage d'un système de processus communicants $(S_i = (Q_i, A_i, T_i, init_i), \mathcal{F})i = 1, \ldots, n$ passe par la mise à plat de son expression de composition $\mathcal{F}$; le formalisme obtenu correspond alors à un *produit synchronisé de systèmes de transitions étiquetées* [Arn92], c'est à dire un ensemble de systèmes de transitions étiquetées dont la mise en parallèle est décrite par des *vecteurs de synchronisation* :

Définition 8.3-1 (Vecteurs de synchronisation)
Un vecteur de synchronisation v *est un élément de* $\prod_{i \in I} A_i \cup \{\kappa\}$ *où* κ *représente l'inaction. La* $i^{ème}$ *composante de* v *indique l'action que* S_i *doit effectuer, l'ensemble des actions indiquées par* v *devant être effectuées simultanément.* ■

Dans le cas de notre traduction, chaque vecteur de synchronisation v est tel que $\exists a \in A, \forall v_i \in v, v_i \in \{\kappa, a\}$, i.e. toutes les composantes actives effectuent la même action. Les actions globales du système construites à l'aide d'un vecteur de synchronisation portent alors l'étiquette de l'action a de ce vecteur.

Un vecteur de synchronisation v tel que seul un élément de v est différent de κ est *asynchrone*. Les autres transitions sont des transitions *synchrones*.

Un produit synchronisé de systèmes de transitions étiquetées correspond en quelque sorte à un Réseau de Petri $(\mathcal{Q}, \mathcal{U}_\prime, \mathcal{T}, \mathcal{G}, \mathcal{V})$ où seule l'unité $\mathcal{U}_\prime$ possède des sous unités. Chaque transition asynchrone correspond à une transition interne à une unité; nous savons donc en construire une représentation symbolique 3. Chaque transition synchrone du système correspond à un *ensemble* de transitions d'un Réseau de Petri équivalent. En effet, un vecteur de synchronisation v va synchroniser ensemble toutes les transitions étiquetées a des composantes i telles que $v_i = a$.

8.3.6 Bilan

L'ensemble des implémentations effectivement réalisées est indiqué dans la table suivante 8.1.

Ces différences d'implémentation ont essentiellement une origine historique, puisque la partie ALDÉBARAN symbolique a débuté avant le développement de MAGEL. En particulier, les

modes de composition simultané et mixte(voir chapitre 3) ne sont opérationnels que dans ALDÉBARAN.

Un point important que nous n'avons pas étudié lors de cette intégration dans ALDÉBARAN est l'adaptation d'un mécanisme de construction de diagnostics lorsque deux systèmes que l'on compare ne sont pas identiques. L'adaptation d'un tel système est un but prioritaire :

- le diagnostic est un outil indispensable à la compréhension, lorsque deux systèmes ne sont pas déclarés équivalents par l'outil. En particulier, il doit fournir des informations sur les causes de l'erreur, mais aussi sur sa localisation, de préférence dans le programme source
- l'utilisation de méthodes symboliques permet d'envisager la construction de diagnostic basés sur des *ensembles d'états*, et plus seulement sur des séquences d'exécution simples menant à des états invalides. En particulier, l'expression du diagnostic en termes de prédicats sur les états est bien adapté à l'explication de la non validité d'une spécification logique (donnée par des formules d'une logique temporelle).

Nous allons maintenant détailler les performances de l'outil MAGEL et de la partie symbolique d'ALDÉBARAN que nous avons réalisée.

Nous séparons cette étude de performances entre les algorithmes basés sur les BDDs et les algorithmes basés sur les polyèdres.

8.4 Performances avec les BDDs

Nous présentons certains des protocoles pour lesquels nous avons utilisé nos outils à base de BDDs. Pour chaque exemple, nous donnons certaines de ses caractéristiques de taille du modèle et de taille du modèle symbolique.

Certains de ces exemples ont été traités avec MAGEL et d'autres avec ALDÉBARAN symbolique (certains avec les deux). Les exemples pouvant être traités avec l'implémentation actuelle de MAGEL sont décrits par des programmes LOTOS ne comportant que des variables booléennes. Dans ce cas, nous utilisons le compilateur CÆSAR pour produire le Réseau de Petri qui sera lu par MAGEL.

Si un programme LOTOS contient des variables de type autres que booléen, et n'est pas facilement traduisible en un programme de cette forme, alors nous pouvons tenter de générer un système de processus communicants au format ALDÉBARAN, par décomposition du programme LOTOS et génération séparée de chacune des composantes. Ce travail peut être réalisé automatiquement par un outil intégré dans la boîte à outils CÆSAR-ALDÉBARAN. Cet outil effectue l'analyse d'un programme LOTOS, et extrait un système de processus communicants au format ALDÉBARAN. Le traitement du protocole sous cette forme se fait alors avec les algorithmes de la partie symbolique d'ALDÉBARAN.

La comparaison de performances entre ces deux alternatives est souvent difficile, car l'utilisation des systèmes de processus communicants donne l'avantage de pouvoir minimiser

à priori chaque composante du système pour la relation d'équivalence utilisée pour la vérification. Par conséquent, il est fréquent que pour un même programme LOTOS, le modèle global (hors minimisation) généré à partir d'un Réseau de Petri soit plus gros que le modèle global généré à partir des processus communicants déjà minimisés.

Tous les chiffres donnés dans les tableaux ci-après ont été obtenus sur une station de travail SPARC station 10 avec 128 Mo de mémoire.

8.4.1 Scheduler de Milner

Le premier exemple que nous considérons est un protocole plutôt artificiel, mais qui sert assez souvent de "benchmark" aux outils de vérification. Il s'agit d'un protocole d'ordonnancement de processus. Un nombre n de processus, appelés *cyclers* sont organisés en anneau. A chaque processus est assigné une tâche particulière. Chaque processus exécute sa tâche, puis passe le contrôle au processus qui le suit dans l'anneau. Nous pouvons considérer ce protocole comme la version sans panne du protocole de jeton circulant présenté plus loin.

Ce protocole présente certaines caractéristiques :

- comme il est utilisé comme exemple par de nombreux outils, il permet une comparaison correcte des performances.
- il est simple de le faire grossir : il suffit d'ajouter autant de *cyclers* que nécessaire. De plus, faire grossir le système global ne fait pas changer la taille de chaque composante.

Cet exemple est aussi une bonne illustration de l'intérêt de la composition simultanée et de la composition mixte, comme nous le verrons plus loin.

Enfin, la description de ce protocole peut se faire uniquement par du contrôle. Par conséquent, nous avons pu le traiter à la fois avec MAGEL et ALDÉBARAN symbolique. Nous donnons les résultats en taille et en temps d'exécution pour les deux cas de figure.

Nous nous intéressons à la génération du modèle minimal pour la bisimulation de branchement ou la bisimulation $\tau^* a$ (le modèle minimal coïncide pour ces deux bisimulations, et les temps d'exécution sont similaires). Le modèle minimal pour la bisimulation forte correspond en fait au modèle global moins une transition. Par conséquent, les performances de la génération de modèle minimal avec les BDDs ne permettent pas de dépasser des exemples de plus de 12 processus pour cette bisimulation.

L'utilisation de ce protocole comme benchmark se fait habituellement sous deux versions :

- les actions début de tâche (a_i) et fin de tâche (b_i) sont toutes visibles :
  Dans ce cas, le modèle minimal pour par exemple la bisimulation de branchement a une taille de l'ordre de 2/3 de celle du modèle global.
- les actions fin de tâche sont cachées :
  dans ce cas, le modèle minimal pour la bisimulation de branchement est proportionnel

au nombre de *cyclers*. Cette version permet de vérifier l'enchaînement correct des tâches.

Comme les performances de la génération de modèle minimal sont liées à la taille du modèle minimal généré, les performances obtenus avec la première version sont peu intéressantes. Par conséquent, nous présentons les performances de notre implémentation uniquement pour la deuxième version.

Dans les tables suivantes, Sc_n correspond à une version du Scheduler avec n processus. Nous donnons en fonction de n la taille du modèle global (hors minimisation) en nombre d'états (N) et de transitions (M). La colonne suivante indique le nombre de variables de l'ensemble support $\vec{x}$ utilisé pour le codage en BDDs, ainsi que la taille du BDDAcc (en nombre de nœuds du graphe de décision) représentant l'ensemble des états accessibles. Le codage de cet exemple est strictement le même pour MAGEL et ALDÉBARAN, en effet l'automate de chaque processus est déjà minimal pour les relations d'équivalence considérées. Par conséquent, les nombres du tableau suivant sont identiques pour les deux outils.

Nom	N	M	Variables	Acc
Sc_{10}	15361	84481	21	77
Sc_{100}	$\sim 10^{32}$	$\sim 10^{34}$	301	401
Sc_{200}	$\sim 10^{62}$	$\sim 10^{64}$	601	1597
Sc_{420}	$\sim 10^{129}$	$\sim 10^{132}$	1261	3357

Paramètres de taille en fonction du nombre de processus

La taille du Réseau de Petri ou du système de processus communicants correspondant à un Scheduler de n processus est :

Réseau de Petri :
Chaque unité correspondant à un processus fait 5 places et 5 transitions. Avec l'unité racine et l'unité de démarrage du protocole, nous avons un réseau de $n + 2$ unités, $5 * n + 2$ places, $5 * n + 2$ transitions.

système de processus communicants :
L'automate de chaque processus comporte 5 états et 6 transitions.

Pour chaque cas de cet exemple, nous donnons les performances pour MAGEL et la composition entrelacée, et ALDÉBARAN avec chaque mode de composition employé. Les performances sont données par la taille du BDD de la relation de transition globale ($\widehat{T}$) et le temps d'exécution (Temps, en secondes) de l'algorithme de minimisation. Ce temps comprend le chargement du modèle, la construction du modèle symbolique, le calcul de ses états accessibles et la génération et écriture sur disque du modèle minimal. Dans le cas d'ALDÉBARAN symbolique, nous indiquons en plus les différentes performances suivant le mode de composition employé.

Enfin, toute case de ce tableau contenant un "-" correspond à des exécutions interrompues par une pénurie de mémoire, ou par l'utilisateur après un temps excédant 10 h.

Nom	MAGEL	ALDÉBARAN					
	$T_{\Diamond}$	$T_{\Diamond}$		$T^{\Downarrow}_{\Diamond}$		$T_{\Diamond\!\!\!\!\!\!\downarrow}$	
	$\widehat{T_{\Diamond}}$	Temps	$\widehat{T_{\Diamond}}$	Temps	$\widehat{T^{\Downarrow}_{\Diamond}}$	Temps	$\widehat{T_{\Diamond\!\!\!\!\!\!\downarrow}}$	Temps
Sc_{10}	426	1.6	426	1	536	1	786	1
Sc_{100}	4476	1911	4476	660	5846	600	9561	380
Sc_{200}	8976	-	8976	-	11746	6403	19311	2994
Sc_{420}	-	-	-	-	-	-	40761	27642

Performances de MAGEL et ALDÉBARAN

Ce tableau montre les possibilités de représentation et de calcul offertes par les BDDs, puisque nous avons pu travailler sur un modèle ayant jusqu'à 10^{129} états. Il faut néanmoins noter que ce protocole est très particulier, et en général ne permet pas de caractériser réellement le comportement d'un algorithme de manière objective. Il nous permet ici plutôt de caractériser les limites de la représentation symbolique BDD, plutôt que l'intérêt de l'algorithme de génération de modèle minimal.

D'autre part, il faut noter que son utilisation en tant que benchmark nous a conduit à traiter cet exemple de manière particulière, du moins avec ALDÉBARAN; en effet, cet exemple se prête bien à une minimisation de ces composantes *avant* l'application d'algorithmes de vérification sur l'ensemble du système. En particulier, si nous voulons appliquer l'algorithme de génération de modèle minimal pour la bisimulation de branchement, nous pouvons alors minimiser chaque composante du système pour cette bisimulation. Chaque composante est alors réduite à un système de transitions étiquetées de 3 états et 3 transitions. Mais la réduction la plus spectaculaire se situe au niveau du système global, puisque sa partie accessible est alors de $2n+1$ états et $2n+1$ transitions (n étant le nombre de cyclers), ce qui en fait un système somme toute aisé à vérifier pour des valeurs importantes de n, la taille du modèle global étant linéaire en fonction du nombre de composantes. Pour rester comparable aux résultats présentés par d'autres outils, nous n'avons pas utilisé cette optimisation pour l'élaboration de ces tableaux.

8.4.2 Protocole de jeton circulant sur un anneau

Ce protocole correspond à un algorithme classique permettant à un réseau de processus de partager une ressource critique. Cet algorithme, proposé dans [Lan77] repose sur une connection des processus selon une structure d'anneau. Cet anneau définit un ordre sur les processus. Une marque spéciale, appelée *jeton*, circule sur cet anneau; le processus ayant en sa possession ce jeton a le droit d'accéder à la section critique. Si le jeton est unique sur l'anneau, alors l'exclusion mutuelle est vérifiée.

Une caractéristique importante de cet algorithme est sa résistance aux pannes. Il permet de traiter deux types de pannes, qui se caractérisent toutes les deux par la perte du jeton :

Erreur de transmission
Lors du passage du jeton entre deux processus, une erreur de transmission se produit et le jeton est perdu.

Panne d'un processus

Le site supportant un processus donné tombe en panne. La transmission du jeton sera alors interrompue au niveau de ce processus.

Si le jeton est perdu, il faut le regénérer. Le problème est alors de choisir *un et un seul* processus parmi les processus encore actifs qui se chargera de cette regénération. Ce choix est effectué par un algorithme d'élection, qui assure que un et un seul processus sera choisi pour régénérer le jeton.

De la même manière que le protocole de Milner, cet exemple est facile à faire grossir en ajoutant des processus dans l'anneau. Néanmoins, chaque composante est plus complexe que dans l'exemple précédent, et nous verrons que pour un nombre limité de processus, le modèle du protocole devient rapidement énorme. D'autre part, la spécification de ce protocole doit intégrer une modélisation de l'anneau, et en particulier des numéros de station sur cet anneau. Cette numérotation peut se faire à l'aide d'une variable entière, limitée à un intervalle déterminé par le nombre de stations. A partir d'une telle spécification, il est facile d'obtenir une spécification uniquement basée sur des variables booléennes. Par conséquent, comme dans le cas du scheduler, nous pouvons utiliser MAGEL ou ALDÉBARAN pour la génération de modèle minimal.

Contrairement à l'exemple précédent, nous allons minimiser pour la bisimulation forte les composantes du système de processus communicants correspondant à ce protocole. Cette minimisation permet d'obtenir des améliorations de performances considérables, comme le montre les différences de taille du modèle global, quand il est généré à partir du Réseau de Petri ou à partir du système de processus communicants.

Nom	RdP				SdPC			
	N	M	Variables	$\widehat{Acc}$	N	M	Variables	$\widehat{Acc}$
Jet_2	21964	62977	25	1067	680	3460	14	182
Jet_3	$3.5\ 10^6$	??	43	18576	5040	20976	24	358
Jet_4	-	-	-	-	241056	$1.2\ 10^6$	34	669
Jet_5	-	-	-	-	$3.5\ 10^6$	$1.1\ 10^8$	43	988
Jet_6	-	-	-	-	$4.7\ 10^7$	$2.7\ 10^9$	52	1295

Paramètres de taille en fonction du nombre de processus

Ce tableau montre les différences de taille importantes obtenues par minimisation préalable des composantes du système. Un point important est que dès que nous considérons la version 3 processus de ce protocole, le modèle généré directement par CÆSAR est de taille prohibitive ; une approche soit par décomposition, soit par génération de modèle minimal est nécessaire.

Les bonnes performances d'ALDÉBARAN sur cet exemple sont en partie à mettre sur le compte d'une optimisation qui semble anodine, mais qui permet des améliorations de performances intéressantes : il s'agit du rapprochement (du point de vue de l'ordre des variables) des variables BDDs de deux processus se synchronisant(voir chapitre 4). Cette optimisation n'est pas (actuellement) intégrée à MAGEL, mais seulement à ALDÉBARAN.

On retrouve ici la grande importance d'un bon choix de l'ordre des variables BDDs pour obtenir de bonnes performances.

Le tableau de performances suivant correspond à la minimisation pour la bisimulation de branchement. Cette fois ci, même si le modèle minimal obtenu coïncide avec celui obtenu par la bisimulation τ^*a, les performances pour la τ^*a sont très nettement moins bonnes. Par conséquent, la composition mixte ou simultanée n'est appliquée que pour le calcul des états accessibles. Nous indiquons néanmoins la taille de la relation de transition correspondant à ces modes de composition.

Nom	MAGEL		ALDÉBARAN					
	$T_{\Diamond}$							
	$\widehat{T_{\Diamond}}$	Temps	$\widehat{T_{\Diamond}}$	Temps	$\widehat{T^{\Vert}_{\Diamond}}$	Temps	$\widehat{T_{\Diamond}}$	Temps
Jet_2			329	0.4	367	0.4	606	0.3
Jet_3		2170	831	21	941	19	2366	20
Jet_4	-	-	1503	220	1680	270	5118	270
Jet_5	-	-	2375	2460	2629	2900	9199	1930
Jet_6	-	-	3345	27420	3659	29500	14149	8740

Performances de MAGEL**et** ALDÉBARAN

Comme pour l'exemple précédent, l'application de la composition simultanée permet d'améliorer les performances, surtout quand le nombre d'itérations devient important. La composition mixte apporte peu d'améliorations, voire même des diminutions de performances. Mais quel que soit le mode de composition, l'utilisation des BDDs permet de produire dans un temps raisonnable un modèle minimal pour un système de plusieurs dizaines de millions d états.

8.4.3 Bus VME

Ce protocole décrit le contrôle d'un bus de type VME (*Versa Module Eurocard*). Nous utilisons comme base de travail une spécification LOTOS de ce protocole écrite par H. Garavel. Le bus VME est un bus asynchrone permettant la communication de processeurs, de mémoires et de périphériques fonctionnant à des vitesses différentes. L'accès au bus est contrôlé par une unité de gestion spécialisée. Chaque périphérique peut faire une demande de réservation du bus à cette unité. L'accès au bus peut aussi se faire sur interruptions, qui sont gérées par une autre unité spécialisée. Les demandes de réservation comme les interruptions sont classées suivant des niveaux de priorité fixés statiquement.

Comme dans le cas précédent, ce protocole est difficile à traiter avec la méthode directe : génération du modèle, puis vérification sur le modèle. En effet, nous n'avons pas pu générer complètement le graphe avec CÆSAR (nous avons arrêté la génération à 17 millions de transitions). De plus, ce protocole utilise des structures de données complexes, interdisant l'utilisation directe de MAGEL sur le Réseau de Petri. Par conséquent, seule l'approche de traduction de la description LOTOS en système de processus communicants permet d'obtenir des résultats.

Pour la construction de ce système, plusieurs niveaux de décomposition sont possibles, car la composition parallèle des différents processus est hiérarchisée suivant un arbre de pro-

fondeur 3. Pour chaque décomposition utilisée, nous avons minimisé les composantes pour la bisimulation forte. Si nous voulons produire le modèle minimal de tout le système pour par exemple la bisimulation de branchement, il est alors possible de minimiser les composantes pour cette même bisimulation. Mais les gains obtenus sont faibles, car les composantes de ce système contiennent peu de τ (leurs actions seront cachées au fur et à mesure de leur composition avec d'autres composantes).

Nous comparons les performances des algorithmes en considérant 3 niveaux de décomposition différentes, qui correspondent respectivement à prendre les nœuds de profondeur 1, 2 et 3 de l'arbre de décomposition comme processus de base.

Profondeur	système de processus communicants					Modèle	
de	Processus	Taille min		Taille Max			
décomposition	N_{proc}	n_1	m_1	n_2	m_2	N	M
1	4	10	17	8110	39715	32187	126618
2	11	8	9	625	3500	48939	200738
3	17	8	9	528	2840	49368	195775

Paramètres de taille en fonction du niveau de décomposition

Les performances du tableau suivant correspondent à la génération de modèle minimal pour la bisimulation de branchement, en ne considérant visibles que trois actions qui correspondent au niveau haut et au niveau bas d'un signal de demande d'accès au bus, et d'un signal d'autorisation d'accès au bus. Le modèle minimal résultant correspond au cycle d'enchaînement de ses actions, donc un modèle de 3 états et 3 transitions.

	Variables	$\widehat{Acc}$	$\widehat{T_\diamond}$	Temps (s)
1	36	15004	99603	953
2	54	17152	16627	383
3	69	46661	11101	726

Performances d'ALDÉBARAN

Le protocole du bus VME constitue donc en quelque sorte un contre exemple des performances jusque là bonnes des BDDs, puisque ALDÉBARAN met plusieurs minutes à minimiser des graphes de taille somme toute faible. Ces performances relativement moyennes sont essentiellement dûes à une explosion de la taille des BDDs manipulés. Cette explosion est manifestement dûe plus à la structure de ce protocole qu'à la modélisation que nous en avons faite. En particulier, nous voyons que pour le niveau de décomposition 1, nous avons un petit nombre de processus de taille relativement importante. Dans ce cas, ce sont les BDDs de la relation de transition qui ont tendance à exploser. Si la décomposition est poussée plus loin, la relation de transition se modélise mieux. Par contre, le nombre de variables des ensembles support augmente et ce sont les BDDs représentant des ensembles d'états (en particulier les états accessibles) qui explosent en taille. L'explosion de la taille des BDDs de cet exemple est essentiellement dûe à un facteur : les processus sont fortement synchronisés entre eux, ce qui crée beaucoup de dépendances entre les variables supports des BDDs. De plus, cette forte synchronisation a une conséquence supplémentaire : la réorganisation de l'ordre des processus pour tenter d'optimiser la taille des BDDs produit peu de résultat, car aucun ordre n'est clairement supérieur à un autre.

Il faut néanmoins noter que ces performances sont probablement plus caractéristiques du comportement des BDDs pour la modélisation de protocoles "réels", comme l'ont confirmé des expériences faites sur d'autres protocoles.

8.5 Discussion

Cette panoplie d'exemples de caractéristiques assez variées que ce soit en taille ou en complexité du protocole nous a permis de dégager les points forts et les points faibles de l'utilisation des BDDs pour la représentation de modèles.

Pour toutes les applications que nous avons réalisées avec les BDDs, les performances sont liées à l'existence de synchronisations plus ou moins fortes entre les processus. En effet, ces synchronisations imposent la mise en relation de variables des ensembles support des BDDs du modèle symbolique et par conséquent un accroissement de leur taille.

Les exemples les plus frappants sont d'un côté le Scheduler de Milner, qui est *régulier*(tous les cyclers sont isomorphes) et faiblement synchronisé (chaque cycler n'est synchronisé qu'avec ses voisins), d'où les performances particulièrement bonnes. A l'autre extrême, nous trouvons le bus VME, qui est constitué de beaucoup de processus différents, et comprend surtout une forte synchronisation, puisque tous les processus sont synchronisés sur le gestionnaire d'accès au bus pour au moins l'action de $RESET$ du bus.

Obtenir de bonnes performances dans le cas de faible synchronisation est en général très intéressant; en effet, la composition de processus asynchrones est une des sources majeures de l'explosion des états dans les modèles construits par entrelacement des actions. Les BDDs permettent donc de traiter une catégorie de protocoles posant habituellement des problèmes à des méthodes énumératives plus classiques. Il serait intéressant de comparer les résultats obtenus avec les BDDs avec des méthodes comme les ordres partiels, qui permettent un traitement particulier de l'entrelacement des actions pour éviter de traiter tous les états du modèle.

Un point important bien exploité dans le cas d'un modèle symbolique est l'utilisation de modèles représentant le parallélisme *implicitement* (Réseau de Petri et système de processus communicants) et non pas *explicitement* (système de transitions étiquetées). Ceci donne d'une part la possibilité d'appliquer certaines méthodes de réduction à priori sur chaque composante d'un modèle composé. En pratique, cette possibilité s'avère extrêmement efficace pour réduire la taille globale du système. D'autre part, la complexité de la représentation du parallélisme est reporté au niveau du modèle symbolique, qui s'accommode en général beaucoup mieux de l'explosion du nombre d'états et de transitions que cela provoque.

Enfin, les expériences ont montré qu'à modèle équivalent, il est toujours plus efficace de construire la représentation symbolique à partir d'un système de processus communicants qu'à partir du système de transitions étiquetées. En effet, la représentation du contrôle dans le système de transitions étiquetées a perdu toute notion de structure du programme de départ. Ceci correspond en fait à essayer de traduire un programme de C à Pascal en passant par la version du programme compilée en assembleur. Toute notion de structure

a disparue, et le code correspondant à la traduction sera beaucoup plus volumineux que le code obtenu par une traduction directe du programme.

Génération de Modèle Minimal

Dans le cas plus particulier de la génération de modèle minimal, des critères supplémentaires entrent en jeu. En particulier, la relation de bisimulation choisie (et donc la taille du modèle généré) a une grande importance. Comme les performances de l'algorithme dépendent plus de la taille du modèle minimal généré que de la taille du modèle de départ, nous obtenons de meilleurs résultats avec des bisimulations "faibles" qu'avec la bisimulation forte. Ceci est à comparer avec des méthodes de minimisation énumératives, où la minimisation pour la bisimulation forte est souvent plus performante que pour d'autres bisimulations plus faibles, pour lesquelles il est souvent nécessaire de transformer la relation de transition par des calculs coûteux de fermeture transitive.

Par conséquent, la génération de modèle minimal avec les BDDs constituent un bon complément des algorithmes énumératifs de minimisation, comme celui intégré dans ALDÉBARAN.

Les BDDs par rapport à l'énumératif

Les différentes expérimentations faites avec les BDDs montrent que cette méthode de représentation doit être considérée comme *complémentaire* des méthodes énumératives "classiques", en particulier la vérification à la volée. Les différences entre ces deux classes de méthodes peuvent être vues selon le type de systèmes à vérifier ou le résultat attendu :

Systèmes à vérifier :
L'utilisation des BDDs a permis de montrer leur efficacité pour des systèmes faiblement synchronisés. Ce genre de système est habituellement sujet au problème de l'explosion des états ; les méthodes énumératives (hors méthodes d'ordre partiel) sont donc difficiles à mettre en œuvre. Par contre, les méthodes énumératives s'avèrent en général plus efficaces pour des systèmes plus synchronisés.

Résultat attendu :
Les méthodes symboliques, et les BDDs en particulier permettent de raisonner de manière plus globale, sur des ensembles d'états. Les méthodes énumératives permettent de raisonner plus facilement de manière locale, au niveau de l'état. Cette différence s'illustre bien lors de l'utilisation ppour la vérification de protocoles. Les méthodes énumératives comme la vérification à la volée sont très intéressantes lors de la phase de mise au point d'un protocole ; les erreurs sont nombreuses, un algorithme de vérification à la volée tombe alors rapidement sur une erreur, permet sa correction par la fourniture d'un diagnostic, et le processus continue. Cette phase de mise au point constitue la partie principale (en temps) du processus de vérification d'un programme.

Les méthodes symboliques interviennent plus efficacement en phase finale de mise au

point. Le protocole est presque correct, les erreurs se situent beaucoup plus loin dans l'exécution, ou s'expriment de manière globale. Les méthodes énumératives peuvent alors être un peu en bout de course, les méthodes symboliques peuvent permettre d'effectuer cette validation finale.

8.6 Performances avec les polyèdres : MAGEL

Nous discutons à présent des performances de notre implémentation centrée autour de l'utilisation des polyèdres. Les performances de notre système d'analyse dépendent d'une part d'une bonne implémentation des opérateurs entre polyèdres. Ces performances sont essentiellement liées à l'utilisation des deux formes duales de représentation des polyèdres. Deux points sont alors à prendre en compte :

le calcul d'un représentation à partir de l'autre :
Comme chaque opérateur entre polyèdres utilisent de préférence l'une ou l'autre des deux représentations, il est important de disposer d'un algorithme efficace de passage d'une forme à l'autre. Nous avons utilisé pour cela une implémentation optimisée par Hervé Leverge [LeV92] de l'algorithme de Chernikova [Che68].

la mémorisation de ces représentations :
Nous pouvons choisir de ne garder en mémoire qu'une des deux formes, par exemple le système de contraintes, et de recalculer la deuxième chaque fois que c'est nécessaire. L'autre solution consiste à conserver en mémoire chacune des deux représentations qui a déjà été calculée, et donc de garder éventuellement des informations redondantes sur les polyèdres. Dans ce compromis entre temps d'exécution et espace mémoire, nous avons choisi de privilégier le temps d'exécution et de conserver en mémoire les deux représentations du polyèdre.

Les performances de l'analyse dépendent d'autre part de facteurs liés au système à analyser. En particulier, nous distinguons les paramètres suivants :

le nombre de variables :
Le nombre de contraintes ou d'éléments générateurs d'un polyèdre est largement dépendant du nombre de variables sur lesquelles sont définis les polyèdres. Il est donc intéressant d'essayer de diminuer au maximum l'ensemble des variables apparaissant dans le système. N'eanmoins, il est difficile d'agir sur ce paramètre sans modifier profondèmenet le système.

le système d'équations : Un autre facteur déterminant pour les performances est la taille et la complexité du système d'équations à résoudre, c'est-à-dire la taille et la complexité du graphe de dépendance. Nous distinguons de plus le graphe de dépendance *généré* et le graphe de dépendance *exploré*. Les performances de l'algorithme de décomposition du graphe dépendront de la taille du graphe généré, alors que les performances de l'analyse dépendent essentiellement de la taille et de la structure du graphe exploré.

Pour illustrer le comportement de notre système d'analyse en fonction de ces différents paramètres, nous reprenons les deux exemples présentés au chapitre 7. Nous avons déclinés ces exemples en plusieurs versions : par ajout de lecteur et rédacteur dans le cas du premier exemple ; par ajout de processus dans le cas du moniteur de tâches.

Exemple	Réseau de Petri		Graphe généré		Graphe exploré		Nombre de variables	temps
	places	transitions	sommets	arcs	sommets	arcs		
3 lecteurs 2 rédacteurs	20	19	320	608	166	250	3	1.3s
4 lecteurs 3 rédacteurs	25	24	2156	4748	754	1257	3	15s
5 lecteurs 3 rédacteurs	27	26	4364	10412	1454	2677	3	42s
Moniteur 2 tâches	17	27	27	51	24	48	8	4.4s
Moniteur 3 tâches	26	46	46	91	43	88	15	25.9s

Performances de l'analyse

Ce tableau fait bien apparaître les deux critères qui influent sur les performances de l'analyse, c'est-à-dire la taille du graphe de dépendance (en particulier la partie explorée de ce graphe) et le nombre de variables présentes dans le réseau. Ce tableau montre aussi que même pour des systèmes de taille importante, les temps d'analyse restent bons. L'expérience montre que la limitation principale actuelle de notre implémentation vient essentiellement de problèmes de débordement arithmétique ou d'explosions de la taille des systèmes de contraintes. Les débordements arithmétiques surviennent lors de la minimisation de systèmes générateurs dont certains éléments contiennent des coefficients rationnels non entiers.

Conclusion

Ce travail se situe dans le cadre de la vérification formelle de systèmes distribués. Cette vérification est basée sur la construction d'un modèle du système à vérifier ; ce modèle est habituellement donné sous la forme d'un *système de transitions étiquetées*. La taille importante de ces modèles est un facteur limitatif majeur de l'applicabilité de ces méthodes ; ce problème est connue sous le nom de *problème d'explosion des états*.

L'objectif de ce travail était l'étude et la mise en œuvre de méthodes permettant de remédier au problème de l'explosion des états. Pour cela, nous avons considéré différentes solutions, qui sont unies par une caractéristique commune : l'utilisation de méthodes *symboliques* de représentation du modèle. Les solutions que nous avons étudiées sont les suivantes :

un modèle :
Nous avons proposé de travailler sur un modèle permettant la représentation du parallélisme de manière compacte et la représentation des variables en compréhension et non plus en extension. Ce modèle est un *Réseau de Petri interprété*, dont les transitions sont étendues par des gardes et des affectations définies sur les variables du programme. Cette catégorie de modèle est fréquemment utilisée comme forme intermédiaire de compilateurs, comme le compilateur CÆSAR. C'est par simulation de ce réseau que CÆSAR génère un système de transitions étiquetées ; le modèle réseau se situe en *amont* des causes principales de l'explosion des états.

un algorithme :
Nous avons étudié et mis en œuvre un algorithme qui s'attaque directement au problème de l'explosion des états, puisqu'il permet de *réduire* un modèle *pendant* sa génération. Cet algorithme est connue sous le nom de génération de modèle minimal. Nous avons modifié la version existante de l'algorithme pour y intégrer certaines optimisations. Certaines de ces optimisations sont directement liées à l'utilisation de représentations symboliques. Nous avons d'autre part défini les structures de données nécessaires pour une mise en œuvre efficace et adapté l'algorithme à différentes relations d'équivalence de bisimulation.

une première représentation symbolique :
Nous avons étudié une méthode de représentation symbolique déjà largement utilisée dans le monde de la vérification ; il s'agit des Diagrammes de Décision Binaires (BDDs), qui permettent la représentation et manipulation efficace de fonctions booléennes.

Nous avons défini la construction d'un *modèle symbolique* basé sur cette représentation, à partir de notre modèle Réseau de Petri.. Nous avons d'autre part adapté certains algorithmes standards de vérification *comportementale* pour une utilisation avec les BDDs. En particulier, nous avons mis en œuvre l'algorithme de génération de modèle minimal pour la réduction et la comparaison de modèles. Nous avons aussi adapté un algorithme de comparaison basé sur la construction d'un *produit synchrone* des modèles. Ceci permet d'étendre la gamme des relations de comparaison à certaines relations de préordre quand un des deux systèmes à comparer est déterministe.

une deuxième représentation symbolique :
nous avons étudié une deuxième méthode de représentation symbolique adéquate pour la représentation de variables numériques. Cette représentation est basée sur des systèmes d'inégalités linéaires, et est connue sous le nom de *polyèdres convexes*.

Pour résoudre les problèmes particuliers posés par les polyèdres pour le calcul de points fixes, nous avons utilisé des résultats déjà établis dans le cadre de l'*interprétation abstraite* [CC77] et utilisés pour l'analyse de programmes impératifs avec les polyèdres [CH78]. Ces résultats permettent en particulier d'approcher le calcul de points fixes éventuellement infinis, par l'introduction d'un opérateur d'*élargissement* dans la séquence de calcul.

Nous avons adapté ces résultats pour une analyse sémantique sur un modèle *semi symbolique*, construit à partir du modèle Réseau de Petri. Ce modèle semi symbolique est un *automate interprété*, où le contrôle est représenté en extension et les données symboliquement.

Nous avons en particulier appliqué un algorithme d'analyse en avant pour déterminer une *approximation supérieure* des états du système. Par la même analyse, nous pouvons déterminer des résultats plus locaux par rapport au contrôle du programme, et déterminer des conditions de franchissement de certaines transitions, des bornes sur des variables en certains points du programme, ...

Mise en œuvre

Nous avons intégré l'ensemble de ces méthodes au sein d'un même outil, l'outil MAGEL. Cet outil permet à partir d'un Réseau de Petri d'effectuer la minimisation ou la comparaison du système de transitions étiquetées correspondant à l'aide des algorithmes réalisés avec les BDDs. D'autre part, il permet l'analyse sémantique du programme représenté par un Réseau de Petri, à l'aide des algorithmes d'analyse réalisés avec les polyèdres.

L'ensemble des méthodes basées sur les BDDs ont de plus été intégrées dans l'outil de vérification ALDÉBARAN, afin de fournir un pendant symbolique à l'ensemble des méthodes énumératives de cet outil. Cette intégration a permis de vérifier ou de faciliter la vérification de certains protocoles. En dehors des exemples jouets comme le Scheduler de Milner, pour lequel les BDDs offrent des performances spectaculaires, nous avons pu traiter des exemples de taille respectable, pour lesquels les méthodes classiques ont des performances très

inférieures (protocole du Transit Node, qui n'a pas été présenté ici), ou sont impuissantes (protocole du Jeton Circulant). Néanmoins, d'autres exemples ont permis aussi de déterminer certaines limites des BDDs, comme dans l'exemple du Bus VME, où les BDDs ont des performances inférieures à certaines méthodes énumératives comme la vérification à la volée.

L'utilisation de la partie analyse avec les polyèdres de MAGEL a permis de découvrir des propriétés intéressantes sur des exemples non triviaux, comme l'exemple du moniteur de tâches présenté au chapitre 7. Les limites de cette méthode d'analyse ne sont pas (actuellement) les temps d'exécution, qui sont en général très bons. Ces limites sont liées à la qualité du résultat (approximations trop fortes), et aussi aux méthodes arithmétiques utilisées dans notre implémentation, qui occasionnent certains débordements dans les valeurs des coefficients.

Par conséquent, cette méthode d'analyse est prometteuse, mais nécessite un certain nombre d'améliorations du point de vue théorique et pratique, pour pouvoir étendre la gamme des problèmes traitables.

Perspectives

Un certain nombre de perspectives à court terme sont des extensions des méthodes que nous avons déjà mises en œuvre :

Diagramme de Décision Multivalués :
une technique que nous avons décrite dans ce document, mais que nous n'avons pas encore mis en œuvre consiste en la représentation de variables entières bornées par des méthodes similaires aux BDDs. Ce genre d'application parait particulièrement adapté au cas des protocoles, puisque ceux-ci utilisent très souvent des entiers bornés, de domaine relativement restreint. Il existe déjà des techniques permettant ce genre de représentation, basés sur des Diagrammes de Décision Multivalués (MDDs). Un exemple particulier de bibliothèque a été réalisé par Timothy Kam [Kam92] , et fait l'objet d'une utilisation intensive dans le cadre d'outils de vérification. L'idée de base de cette bibliothèque est la mise en œuvre des MDDs par codage en BDDs. A chaque variable entière est alors associée un ensemble support de variables booléennes permettant son codage en BDDs. D'un point de vue pratique, nous envisageons de mettre en œuvre une technique similaire, en créant une interface MDD au dessus de notre bibliothèque BDD. La réalisation de cette interface pourrait largement s'inspirer des travaux de T.Kam.

D'un point de vue théorique, nous nous intéressons à une autre approche de mise en œuvre des MDDs, par le développement d'une théorie permettant de construire des Diagrammes de Décision n-aires. Des travaux dans ce sens existent déjà pour des Diagrammes de Décision Ternaires. La difficulté est de réussir à retrouver les caractéristiques qui font l'efficacité des BDDs, à savoir la définition d'une forme normale compacte et la détermination d'algorithmes efficaces pour les opérations sur ces représentations.

analyse en arrière :
l'analyse sémantique nous permet de déterminer certaines propriétés invariantes sur les variables du programme. Dans MAGEL, cette analyse repose entièrement sur une analyse *en avant* du programme. Certains systèmes d'analyse sémantique combinent cette analyse en avant avec une analyse *en arrière*. L'analyse en arrière permet de répondre à des questions d'accessibilité d'ensembles d'états et d'autre part d'affiner les résultats de l'analyse en avant. Cette combinaison d'analyse avant et arrière ne doit pas poser de problèmes difficiles d'adaptation à notre système et permettrait d'améliorer les résultats obtenus avec MAGEL.

diagnostic :
Un manque manifeste des outils que nous avons réalisés concerne la production de *diagnostics* quand la comparaison de deux systèmes indiquent qu'ils ne sont pas en relation l'un avec l'autre. L'utilisation de méthodes symboliques permet d'envisager la production de diagnostics généraux; en effet, la construction de diagnostics se fait alors directement en termes de prédicats sur les états, et en terme de graphes plutôt que de séquences. Certaines expérimentations ont montré l'intérêt de cette approche; lors du calcul des états puits d'un modèle avec les BDDs, nous avons implémenté un calcul des séquences minimales permettant à partir de l'état initial d'accéder aux états puits. La construction du graphe complet de toutes ces séquences s'avèrent du même ordre de coût que le calcul d'*une* séquence particulière.

analyse de systèmes temporisés :
une application récente de l'analyse approchée avec les polyèdres concerne l'analyse de systèmes temporisés [Hal93] , et en particulier à l'analyse de systèmes hybrides linéaires [HPR94]. Ce genre d'applications devient particulièrement intéressant dans notre contexte, avec l'introduction éventuelle du temps dans le langage LOTOS. Une simple extension de l'outil MAGEL tenant compte des résultats de [HPR94] devrait permettre d'intégrer l'analyse de programmes LOTOS temporisés.

Applications des résultats de l'analyse sémantique

Les résultats de l'analyse sémantique vont nous permettre d'améliorer et d'optimiser les différents outils de vérification formelle que nous utilisons.

Un première application concerne le compilateur CÆSAR. Les informations obtenus sur des programmes LOTOS grâce aux méthodes d'analyse de MAGEL pourraient être utilisées par le compilateur CÆSAR pour optimiser la génération de son graphe en diminuant la taille de son vecteur d'état.

Une autre direction pratique serait d'intégrer ces méthodes d'analyse dans le cadre d'un débogueur *abstrait*, dans la même idée que le système exposé dans [Bou92]. Un tel système, qui permet de répondre à des questions telles que "Quelles sont les valeurs possibles de la variable x en tel point de communication?", "Quelles sont les conditions nécessaires pour arriver à tel point du programme avec $x+y \leq 5$?", viendrait compléter avec profit un système

de débogage basé sur un système plus classique de simulation.

Nous pouvons aussi utiliser les résultats de l'analyse sémantique pour améliorer le codage de notre modèle en BDDs, et donc améliorer les performances des algorithmes de vérification que nous avons mis en œuvre. En particulier, nous pouvons obtenir et utiliser avec profit les informations suivantes :

Bornes des variables entières
L'analyse nous permet de déterminer une approximation supérieure du domaine de variation des variables. Cette information permet de rendre possible (si le domaine des variables était inconnu) ou d'optimiser (par diminution du nombre de variables supports nécessaires) le codage des variables entières sous forme de BDD.

Relations entre variables
Une cause fréquente d'explosion de la taille des BDDs est l'existence de relations invariantes entre les variables du programme représenté. Une solution habituelle pour limiter cette explosion est de modifier (quand c'est possible) l'ordre de codage en BDDs des variables pour que des variables en relation soient les plus proches possible. D'autres travaux [HD93] proposent de coder ces relations dans des BDDs particuliers. Ces BDDs, qui sont généralement de petite taille, sont considérés comme étant *implicitement* en conjonction avec les BDDs représentant la relation de transition du système. Les algorithmes de vérification sont alors adaptées pour travailler avec cette conjonction implicite. Ceci permet de diminuer sensiblement la taille de la relation de transitions.

Quelle que soit la méthode utilisée pour traiter l'existence de relations entre variables, il faut pouvoir déterminer ces relations a priori, de préférence de manière automatique. L'analyse sémantique nous permet de déterminer un sous ensemble des relations existantes et peut donc servir à optimiser la représentation à base de BDDs.

Bibliographie

[ACHH93] R. Alur, C. Courcoubetis, T. A. Henzinger et Pei-Hsin Ho. Hybrid automata: an algorithmic approach to the specification and analysis of hybrid systems. Dans *Workshop on Theory of Hybrid Systems*, Lyngby, Denmark, octobre 1993. LNCS 736, Springer Verlag.

[ADAY92] A.J.Hu, D.Dill, A.J.Drexler et C.Han Yang. Higher-level specification and verification with bdds. Dans *Computer Aided Verification*, juillet 1992.

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

[Arn92] A. Arnold. *Systèmes de transitions finis et sémantique des processus communicants.* 1992.

[BCL91] J.R. Burch, E.M. Clarke et D.E. Long. Symbolic model checking with partitioned transition relations. Rapport de recherche CMU-CS-91-195, Carnegie Mellon University, October 1991.

[BCM+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill et L.J. Hwang. Symbolic model checking : 10^{20} states and beyond. Dans *Proceedings 5th Annual Symposium on Logic in Computer Science (LICS 90), Philadelphia USA*, pages 118–129, Los Alamitos, CA, juin 1990. IEEE Computer Society Press.

[BdS92] A. Bouali et R. de Simone. Symbolic bisimulation minimisation. Dans G. Bochmann, éditeur, *Proceedings of the fourth workshop on Computer-Aided Verification (Montreal,Canada)*, june 1992.

[BFH90] A. Bouajjani, J.C. Fernandez et N. Halbwachs. Minimal model generation. Dans R.P. Kurshan et E.M. Clarke, éditeurs, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA).* DIMACS, juin 1990.

[BFH+92] A. Bouajjani, J.-C. Fernandez, N. Halbwachs, P. Raymond et C. Ratel. Minimal state graph generation. *Science of Computer Programming*, 18:247–269, 1992.

[Bil87] J.P. Billon. Perfect normal forms for discrete programs. Rapport de recherche, Bull, septembre 1987.

[BM88] J.P. Billon et J.C. Madre. Original concepts of PRIAM, an industrial tool for efficient formal verification of combinational circuits. Dans G. Milne, éditeur, *IFIP WG 10.2 Int Working Conference on the Fusion of Hardware Design and Verification*, Glasgow, Scotland, juillet 1988. North Holland.

[Bou92] Francois Bourdoncle. *Semantique des languages imperatif d'ordre superieur et interpretation abstraite*. PhD thesis, Ecole Polytechnique, 1992.

[Bou93] A. Bouali. *Etudes et mise en œuvre d'outils de vérification basée sur la bisimulation*. Thèse de doctorat, Université de Paris VII, U.F.R. d'informatique, 1993.

[Bry86] R. E. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, C-35(8), 1986.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary decision diagrams. *ACM Computing Surveys*, 1992.

[BW94] Bernard Boigelot et Pierre Wolper. Symbolic verification with periodic sets. *6th Workshop on Computer-Aided Verification,Stanford*, 1994.

[CBM89] O. Coudert, C. Berthet et J. C. Madre. Verification of synchronous sequential machines based on symbolic execution. Dans *International Workshop on Automatic Verification Methods for Finite State Systems, Grenoble*. LNCS 407, Springer Verlag, 1989.

[CC76] P. Cousot et R. Cousot. Static determination of dynamic properties of programs. Dans *2nd International Symposium on Programming*, 1976.

[CC77] P. Cousot et R. Cousot. Abstract interpretation: a unified lattice model for static analysis of programs by construction or approximation of fixpoints. Dans *4th POPL*, janvier 1977.

[CES83] E. Clarke, E.A. Emerson et A.P. Sistla. Automatic verification of finite state concurrent systems using temporal logic. Dans *10th. Annual Symp. on Principles of Programming Languages*, 1983.

[CGL92] E.M. Clarke, O. Grumberg et D.E. Long. Model checking and abstraction. Dans *Symposium on Principles of Programming Languages (POPL 92)*. ACM, octobre 1992.

[CH78] P. Cousot et N. Halbwachs. Automatic discovery of linear restraints among variables of a program. Dans *5th. Annual Symp. on Principles of Programming Languages*, pages 84–87, 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.

[CM90] O. Coudert et J. C. Madre. A unified framework for the formal verification of sequential circuits. Dans *International Conference on Computer Aided Design (ICCAD), Santa Clara*, 1990.

[CMB90] O. Coudert, J.C. Madre et C. Berthet. Verifying temporal properties of sequential machine without building their state diagram. Dans R.P. Kurshan et E.M. Clarke, éditeurs, *Proceedings of the Workshop on Computer-Aided Verification (Rutgers, USA)*. DIMACS, juin 1990.

[Cou78] P. Cousot. *Méthodes itératives de construction et d'approximations de points fixes d'opérateurs monotones sur un treillis, analyse sémantique des programmes séquentiels*. Thèse d'eťat, Université Scientifique et Médicale de Grenoble, mars 1978.

[Cou81] P. Cousot. *Semantic Foundations of Program Analysis*, chapitre 10. Prentice Hall, Inc., Englewood Cliffs, 1981.

[Cou91] O. Coudert. *SIAM : Une Boîte à Outils pour la Preuve Formelle de Systèmes séquentiels*. Thèse de 3e cycle, Ecole Nationale Supérieure des Télécommunications, octobre 1991.

[EFT91] R. Enders, T. Filkorn et D. Taubner. Generating bdds for symbolic model checking in ccs. Dans K. G. Larsen, éditeur, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, juillet 1991.

[Fer88] J.C. Fernandez. *ALDEBARAN : un système de vérification par réduction de processus communicants*. Thèse de doctorat, Université Joseph Fourier (Grenoble), mai 1988.

[Fer90] J.C. Fernandez. An implementation of an efficient algorithm for bisimulation equivalence. *Science of Computer Programming*, 13(2–3):219–236, mai 1990.

[FJJM92] J.C. Fernandez, C. Jard, T. Jéron et L. Mounier. "on the fly" verification of finite transition systems. *Formal Methods in System Design, Kluwer Academic Publishers*, 1992.

[FKM93] J.C. Fernandez, A. Kerbrat et L. Mounier. Symbolic equivalence checking. Dans C. Courcoubetis, éditeur, *Proceedings of the 5th Workshop on Computer-Aided Verification (Heraklion, Greece)*, 1993.

[FS87] S.J. Friedmann et K.J. Supowit. Finding the optimal variable ordering for binary decision diagrams. Dans *24th Design Automation Conference*, 1987.

[Gar89] H. Garavel. *Compilation et vérification de programmes LOTOS*. Thèse de doctorat, Université Joseph Fourier (Grenoble), novembre 1989.

[GJ79] M.R. Garey et D.S. Johnson. Computer and intractibility: a guide to the theory of np-completeness. *WH Freeman and Co*, 1, 1979.

[Gra91] P. Granger. *Analyses Sémantiques de Congruence.* Thèse de 3e cycle, Ecole Polytechnique, juillet 1991.

[GV90] J.F. Groote et F. Vaandrager. An efficient algorithm for branching bisimulation and stuttering equivalence. CS R9001, Centrum voor Wiskunde en Informatica, Amsterdam, janvier 1990.

[GW93] P. Godefroid et P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. Dans Kluwer Academic Publishers, éditeur, *Formal Methods in System Design*, April 1993.

[Hal79] N. Halbwachs. *Détermination automatique de relations linéaires vérifiées par les variables d'un programme.* Thèse de 3e cycle, Université de Grenoble, mars 1979.

[Hal93] N. Halbwachs. Delay analysis in synchronous programs. Dans *5th Conference on Computer-Aided Verification (Elounda, Greece).* LNCS 697, Springer Verlag, jul 1993.

[Hal94] Nicolas Halbwachs. Polka user manual. Document, Verimag, 1994.

[HD93] A. J. Hu et D. L. Dill. Efficient verification with BDDs using implicitly conjuncted invariants. Dans *Fifth Int. Workshop on Computer Aided Verification, Elounda (Crete)*, juillet 1993.

[HM85] M. Hennessy et R. Milner. Algebraic laws for nondeterminism and concurrency. *Journal of the Association for Computing Machinery*, 34(1):137–161, january 1985.

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

[Hol89] G.J. Holzmann. Algorithms for automated protocol validation. Dans Joseph Sifakis, éditeur, *Proceedings of the 1st International Workshop on Automatic Verification Methods for Finite State Systems (Grenoble, France)*, juin 1989.

[HPR94] N. Halbwachs, Y.E. Proy et P. Raymond. Verification of linear hybrid systems by means of convex approximations. Dans *First International Static Analysis Symposium.* Springer Verlag, septembre 1994.

[JJ91] Claude Jard et Thierry Jéron. Bounded-memory algorithms for verification "on-the-fly". Dans K. G. Larsen, éditeur, *Proceedings of the 3rd Workshop on Computer-Aided Verification (Aalborg, Denmark)*, juillet 1991.

[Jou94] M. Jourdan. *Etude d'un environnement de programmation et de vérification des systèmes réactifs multi-langages et multi-outils.* PhD thesis, Université Joseph Fourier, Grenoble, 1994.

[Kam92] Timothy Kam. Mdd package documentation. Document, University of California, 1992.

[Kar76] M. Karr. Affine relationships among variables of a program. Dans *Acta Informatica*, volume 6, 1976.

[Ker94] A. Kerbrat. Reachable states space analysis of lotos programs. *7th international Conference on Formal Description Techniques for Distributed Systems and Communication Protocols*, 1994.

[Kil73] G. Kildall. A unified approach to global program optimization. Dans *ACM symposium on Principles of Programmins Languages*, 1973.

[KS83] P.C. Kanellakis et S.A. Smolka. Ccs expressions, finite state processes, anf three problems of equivalence. Dans *Second ACM Symposium on Principles of Distributed Computing (PODC), Montreal, Quebec, Canada*, août 1983.

[Kur89] R.P. Kurshan. Analysis of discrete event coordination. LNCS 430. Springer-Verlag, mai 1989.

[Lan77] G. Le Lann. *Distributed systems - Towards a formal approach*, pages 155–160. North Holland, 1977.

[LeV92] H. LeVerge. A note on Chernikova's algorithm. Technical Report 635, IRISA, février 1992.

[Loi94] C. Loiseaux. *Vérification symbolique de systèmes réactifs à l'aide d'abstractions.* PhD thesis, Université Joseph Fourier, Grenoble, février 1994.

[Lon93] D. E. Long. Model checking, abstraction and compositional verification. PhD Thesis, Carnegie Mellon University, juillet 1993.

[LY92] D. Lee et M. Yannakakis. Online minimization of transition systems. Dans *24th ACM Symp. on the Theory of Computing, STOC'92, Vancouver, B.C.*, 1992.

[Mad90] J.C. Madre. *PRIAM, un outil de vérification formelle de circuits intégrés digitaux.* PhD thesis, Ecole Nationale Supérieure des Télécommunications, 1990.

[McM92] K.J. McMillan. *Symbolic Model Checking.* PhD thesis, Carnegie Mellon University, 1992.

[Mil80] R. Milner. A calculus of communicating systems. volume 92 of *LNCS*, Berlin, 1980. Springer Verlag.

[Mou92] L. Mounier. *Vérification de spécifications comportementales : étude et mise en œuvre.* Thèse de doctorat, Université Joseph Fourier (Grenoble), janvier 1992.

[NMV90] R. De Nicola, U. Montanari et F. Vaandrager. Back and forth bisimulations. CS R9021, Centrum voor Wiskunde en Informatica, Amsterdam, mai 1990.

[NV90] R. De Nicola et F. Vaandrager. Three logics for branching bisimulation. Dans *Proceedings 5th Annual Symposium on Logic in Computer Science (LICS 90), Philadelphia USA*, pages 118–129, Los Alamitos, CA, juin 1990. IEEE Computer Society Press.

[Par81] David Park. Concurrency and automata on infinite sequences. Dans Peter Deussen, éditeur, *Theoretical Computer Science*, volume 104 of *Lecture Notes in Computer Science*, pages 167–183, Berlin, mars 1981. Springer Verlag.

[PT87] Robert Paige et Robert E. Tarjan. Three partition refinement algorithms. *SIAM Journal of Computing*, 16(6):973–989, décembre 1987.

[QS83] Jean-Pierre Queille et Joseph Sifakis. Fairness and related properties in transition systems — a temporal logic to deal with fairness. *Acta Informatica*, 19:195–220, 1983.

[Rat92] C. Ratel. *Définition et Réalisation d'un Outil de Vérification Formelle de Programmes LUSTRE: le Système LESAR.* PhD thesis, Université Joseph Fourier, Grenoble, 1992.

[Rod88] Carlos Rodríguez. *Spécification et validation de systèmes en XESAR.* Thèse de doctorat, Institut National Polytechnique de Grenoble, mai 1988.

[Sha38] C. E. Shannon. A symbolic analysis of relay and switching circuits. *Transactions AIEE*, 57:305–316, 1938.

[Tar83] R.E. Tarjan. An improved algorithm for hierarchical clustering using strong components. *Information Processing Letters 17*, 1983.

[Val93] A. Valmari. On-the-fly verification with stubborn sets. Dans C. Courcoubetis, éditeur, *Proceedings of the fifth workshop on Computer-Aided Verificatio n (Elounda, Crete)*. DIMACS, june 1993.

[Ver92] H. Le Verge. *Un environnement de transformations de programmes pour la synthèse d'architectures régulières.* PhD thesis, Université de Rennes, 1992.