Thèse de Doctorat
de l'École Normale Supérieure de Cachan

présentée par
Monsieur Sébastien Bardin

pour obtenir le grade de

Docteur
de l'École Normale Supérieure de Cachan

(Spécialité : Informatique)

Vers un Model Checking avec Accélération Plate des Systèmes Hétérogènes

Thèse soutenue à Cachan le 20 octobre 2005 devant le jury composé de :

Yassine Lakhnech	Professeur des Universités	Président du jury
Ahmed Bouajjani	Professeur des Universités	Rapporteur
Jean-François Raskin	Professeur associé	Rapporteur
Grégoire Sutre	Chargé de Recherche	Examinateur
Alain Finkel	Professeur des Universités	Directeur de thèse
Laure Petrucci	Professeur des Universités	Co-directrice de thèse

Thèse préparée au sein du Laboratoire Spécification et Vérification.

Introduction

Les *systèmes réactifs* sont des systèmes autonomes, logiciels ou matériels, qui interagissent avec leur environnement ; ils sont de plus en plus répandus : processeurs embarqués dans les avions et les voitures, protocoles de communication sur internet ou encore programmes multi-threads.

Un système réactif est le plus souvent distribué, c'est-à-dire qu'il est constitué d'un ensemble de composants communicant entre eux et avec l'environnement. Le comportement de chaque composant est communément représenté par un automate fini de contrôle muni d'un nombre fini de variables. La figure 1 représente un protocole classique de transmission de données via des canaux non fiables. Le protocole est constitué de deux composants : l'émetteur et le récepteur, qui communiquent par deux canaux. Les composants peuvent lire des messages (noté ``?a'') et envoyer des messages (noté ``!a''). Les messages possibles sont : ``A0'', ``A1'', ``0'', ``1''.

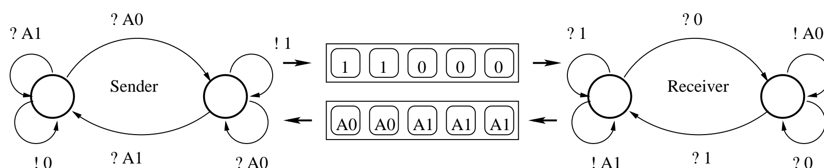


FIG. 1 – Le protocole du bit alterné

La vérification des systèmes réactifs est une tâche ardue à cause entre autres des entrelacements des comportements des différents composants. Il existe deux grandes approches de la vérification formelle des systèmes réactifs : la preuve et le model checking. La preuve consiste à exprimer le système et sa propriété sous forme de formules logiques ϕ_s et ϕ_p et à démontrer $\phi_s \vdash \phi_p$. Cette technique s'applique à de nombreuses catégories de problèmes, mais bien souvent c'est l'utilisateur qui prouve la correction du système et le logiciel vérifie la validité de la preuve. Le model checking [CGP99] repose sur

des techniques plus automatiques d'exploration systématique des comportements possibles du système. C'est cette approche que nous suivons dans ce travail.

Model-checking

Nous considérons que les systèmes sont donnés par un automate fini de contrôle muni de variables, comme celui de la figure 1. Une *configuration* du système est la donnée d'un état de l'automate de contrôle (appelé par la suite *location*) et d'une valeur pour chaque variable du système. Les transitions du système définissent une relation d'accessibilité en un coup entre les configurations. Ainsi on note $c_1 \xrightarrow{t} c_2$ si à partir de la configuration c_1 on atteint la configuration c_2 en utilisant la transition t. Une configuration c est accessible s'il existe une suite de transitions $t_1, \ldots, t_n$ et une configuration initiale c_0 telles que $c_0 \xrightarrow{t_1} \ldots \xrightarrow{t_n} c$. L'ensemble des configurations accessibles s'appelle l'ensemble d'accessibilité.

Les propriétés que l'on désire vérifier sur les systèmes réactifs sont typiquement l'équivalence comportementale à un système de référence (par exemple les équivalences de traces ou la bisimulation), des propriétés dynamiques exprimées dans une logique temporelle ou plus simplement des propriétés d'accessibilité. Ces dernières sont le plus souvent exprimées en termes d'ensembles de *mauvaises* configurations à éviter. Dans tout le document nous nous concentrons sur les propriétés d'accessibilité.

On distingue les systèmes dont l'ensemble d'accessibilité est fini (*systèmes finis*) des systèmes dont l'ensemble d'accessibilité est infini (*systèmes infinis*).

Systèmes finis. Dans le cas fini, les propriétés "raisonnables" sont décidables. Une propriété d'accessibilité P se décide en énumérant toutes les configurations accessibles du système et en vérifiant que chacune satisfait P. Ceci peut être fait par un calcul itératif comme illustré à la figure 2.

Cette approche se heurte à l'explosion combinatoire du nombre de configurations effectivement accessibles. Cependant, des techniques comme les ordres partiels ou la représentation symbolique des configurations [JEK$^+$90] ont permis au model checking de passer à l'échelle. Dans l'approche symbolique on ne travaille plus directement sur les configurations mais sur une représentation symbolique, des arbres binaires de décision [Bry92] dans ce cas. Cette méthode est aussi à la base du model checking de systèmes infinis.

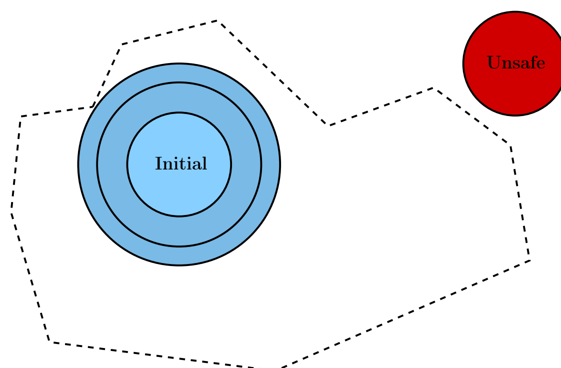


FIG. 2 – Vérification itérative d'une propriété d'accessibilité

Systèmes infinis. Un système est infini si les variables qu'il manipule sont non bornées. De nombreux systèmes sont naturellement infinis car ils manipulent des piles (récursion), des files (communication), des horloges (prise en compte du temps) ou des compteurs. Le model checking de systèmes infinis se heurte rapidement à un obstacle de taille : l'énumération des configurations ne termine pas sur de tels systèmes.

Évidement, les systèmes réellement mis en œuvre sont toujours finis : il y a toujours une borne sur la capacité des canaux, une précision limitée des horloges, une mémoire limitée. Cependant, soit le nombre de configurations est tellement élevé que, malgré tout, l'énumération ne fonctionne pas[1] ; soit il est difficile de déterminer une borne (par exemple la capacité des canaux de communication du réseau internet) ; soit on veut vérifier le système pour n'importe quelle borne (vérification paramétrée).

Deux approches ont été développées pour étendre le model checking aux systèmes infinis. L'ensemble d'accessibilité peut parfois être calculé modulo une relation d'équivalence sur les configurations pour se ramener à l'étude d'un système fini. Cette approche a notamment été mise en œuvre avec succès dans les automates temporisés [AD94]. On peut également adapter les méthodes de model checking symbolique en manipulant cette fois des représentations symboliques d'ensembles de configurations infinis. Par exemple pour un système à files comme celui de la figure 1, on peut représenter des ensembles de contenus de file par une expression régulière sur l'alpahabet $\{\texttt{A0},\texttt{A1},\texttt{0},\texttt{1}\}$. Ainsi $\texttt{A0}^*$ représentera tous les contenus de files contenant

[1] Il suffit de deux entiers codés sur 32 bits pour avoir potentiellement 2^{64} configurations du système.

uniquement une suite de messages `A0`.

Model-checking symbolique. Les représentations symboliques les plus utilisées sont basées sur des langages réguliers : elles sont expressives et les structures de données basées sur les automates offrent des algorithmes bien connus et performants pour les opérations ensemblistes de base, comme l'union ou le calcul de successeurs/prédécesseurs, ainsi que le test d'inclusion. Avec ces opérations il devient possible de lancer une procédure itérative de calcul des configurations accessibles (voir par exemple [KMM+01]). Ces techniques permettent de travailler sur des modèles tels que des systèmes à pile [BEF+00], des systèmes à files [BH99, ACBJ04], des systèmes à compteurs [WB98, ISD+02, BFLP03], et beaucoup d'autres familles de systèmes. Ces systèmes sont très expressifs et souvent les problèmes de vérification associés sont indécidables. De plus, dans la pratique, une procédure itérative de calcul symbolique de point fixe calquée sur celle du model checking fini a très peu de chances de terminer.

Accélération

Pour améliorer la convergence de la procédure, des "*techniques d'accélération*" ont été développées. Ces techniques permettent de calculer des sous ensembles de l'ensemble d'accessibilité qui ne sont pas uniformément bornés. On peut procéder par exemple en remplaçant une boucle de contrôle "`x :=x+1 ; y :=y-1`" par sa clôture transitive "`k :=random_int() ; x :=x+k ; y :=y-k`".

On trouve des prémisses de l'accélération dans la construction de l'arbre de couverture de réseaux de Petri par Karp et Miller en 1968, étendue par Finkel aux systèmes de transitions bien structurés [Fin87]. Le premier article sur l'accélération est probablement [BW94], dans lequel Boigelot et Wolper étudient l'effet de l'itération de fonctions affines.

Actuellement il existe de nombreuses techniques d'accélération pour des familles différentes de systèmes : systèmes à compteurs [WB98, FL02], systèmes à files parfaites [BGWW97, BH99], systèmes à files lossy [ACBJ04], systèmes hybrides [AAB00, BHJ03]. Certaines ont été implantées [ABS01, BFLP03, Las] et des études de cas prometteuses ont été menées à bien [AAB99, AAB00, ABS01, BFLP03, ACBJ04, BFL04]. Ces résultats d'accélération consistent souvent en un théorème énonçant que la clôture transitive d'une action, ou d'une séquence d'actions, peut être effectivement calculée. La difficulté de ces résultats réside en général dans l'identification précise des condi-

tions sur la transition et sur l'ensemble initial de configurations qui assurent l'effectivité du calcul.

Cependant, la façon d'utiliser de tels résultats n'est pas claire. Il y a deux questions principales : comment choisir les cycles à accélérer, et comment injecter le résultat dans un calcul plus classique de point fixe. Dans certains outils, par exemple LASH [Las], c'est l'utilisateur qui doit indiquer à l'outil quelles boucles accélérer ; dans d'autres comme TReX [ABS01], une heuristique parmi d'autres est implantée par défaut.

Alternatives à l'accélération

Sous-classes décidables. Il existe des classes de systèmes infinis pour lesquelles certaines propriétés d'accessibilité sont décidables. Parmi les plus connues on peut citer les réseaux de Petri [May81], les systèmes à compteurs reversal-bornés [Iba78], les automates temporisés [AD94], les systèmes à files lossy [CFP96] et les systèmes à pile [BEM97]. Cependant les algorithmes proposés sont bien souvent inefficaces en pratique et on leur préfère des procédures symboliques plus efficaces. Par exemple l'algorithme d'accessibilité des réseaux de Petri n'a pas été implanté, la procédure de référence pour les automates temporisés est un calcul itératif approché[2] et bien que le calcul en arrière des systèmes à files lossy garantisse la terminaison, c'est le calcul en avant qui est implanté.

Approximations. Certaines approches relâchent le calcul exact et se permettent de calculer une surapproximation de l'ensemble d'accessibilité. Les calculs de successeurs en sont simplifiés et sont souvent moins coûteux que les accélérations, cependant il se peut que la surapproximation soit trop grossière pour vérifier la propriété. Parfois les approximations visent à assurer la terminaison du calcul, mais ce n'est pas toujours le cas. On peut citer les méthodes de widening de l'interprétation abstraite [CC77, CH78, ACH+95], ou encore les méthodes basées sur le paradigme *"abstract-check and refine"*, où l'abstraction est affinée quand un faux contre-exemple est découvert. Citons par exemple [BB04, GRV04].

[2]Dans ce cas la complexité théorique de la procédure approchée est même supérieure à la complexité de l'algorithme d'accessibilité.

Contribution de la thèse

Nous nous intéressons dans cette thèse au développement des méthodes d'accélération. Notre contribution porte sur les aspects théoriques : cadre unifié de l'accélération, composition d'accélérations ; algorithmiques : des structures de données et des algorithmes spécifiques aux compteurs et aux pointeurs ; et expérimentaux : implantation de l'outil FAST et études de cas.

Fondements de l'accélération. Les techniques d'accélération existantes ne reposent pas sur des bases théoriques communes, même si elles partagent des idées semblables. De plus, l'utilisation effective des algorithmes d'accélération dans une procédure de calcul de point fixe n'a été abordée que dans la thèse de Leroux [Ler03b] dans le cadre des systèmes à compteurs. Nous définissons un cadre théorique de l'accélération englobant la plupart des techniques connues. Ce cadre permet de comparer les différentes accélérations existantes et fournit des repères pour le développement de nouvelles techniques. Nous avons aussi contribué à combler le fossé entre les théorèmes d'accélération plate d'une part, et leur utilisation effective à l'intérieur d'une procédure de calcul de point fixe d'autre part. La notion d'applatissement permet de cerner exactement les systèmes dont l'ensemble d'accessibilité est calculable par accélération plate. Nous proposons différents algorithmes complets pour le calcul d'accessibilité des systèmes applatissables, et des optimisations indépendantes du type de donnés considéré. Une amélioration importante est la diminution du nombre de cycles utiles au calcul des configurations accessibles par des méthodes de *réduction*. Nous définissons deux réductions génériques : la réduction par conjugaison et la réduction par commutation. Ces résultats ont été publiés dans [BFLS05].

Composition d'accélérations. Les systèmes hétérogènes manipulent des variables de différents types, par exemple des files et des compteurs. Une difficulté dans l'analyse de ces systèmes est que bien souvent des techniques symboliques existent pour chaque type de données, mais aucune technique n'est disponible pour le système complet. On se retrouve alors confronté à deux choix insatisfaisants : soit on vérifie des abstractions (de projection) du système, type de données par type de données, mais ces abstractions risquent d'être trop grossières ; soit on développe complètement de nouveaux résultats ad hoc. Nous proposons une voie médiane : la composition des représentations symboliques et des algorithmes d'accélérations. La difficulté principale est qu'en général le produit cartésien d'accélérations n'est pas une accélération. Tout d'abord nous identifions la classes des systèmes faiblement

hétérogènes, dans laquelle le domaine des variables *et* les transitions sont des produits cartésiens. Ensuite nous définissons une classe d'algorithmes d'accélération stable par composition sur des systèmes faiblement hétérogènes. La composition est effective. Ces résultats ont été publiés dans [BF04].

Systèmes à compteurs. Les systèmes à compteurs ont été parmi les premiers à bénéficier de résultats d'accélération [BW94, FO97a]. Une représentation symbolique usuelle est à base d'automates binaires [BC96, WB00, Ler03a] et les algorithmes d'accélération sont ceux de [FL02, Boi03]. Nous nous sommes concentrés sur l'application effective de l'accélération à ces systèmes. Nous proposons de nouveaux algorithmes d'accélération pour les translations convexes et les translations positives. Ces algorithmes sont respectivement quadratique et linéaire dans la garde de la fonction alors que l'algorithme de Finkel et Leroux [FL02] pour des fonctions plus générales est exponentiel dans la garde. Nous prouvons aussi que la réduction par union [FL02] permet de calculer l'ensemble d'accessibilité de systèmes non applatissables. Ces résultats ont été publiés avec Finkel et Leroux dans [BFL04].

L'outil Fast. Les résultats sur les compteurs ont été implantés dans l'outil de vérification Fast [BFL+, Fas]. L'outil fournit un moteur d'analyse puissant qui calcule effectivement l'ensemble d'accessibilité de nombreux systèmes, ce qui surpasse les outils concurrents. La technologie au cœur de Fast est aussi très souple et peut facilement être adaptée à d'autres types de calculs comme les ensembles co-accessibles ou les ensembles de couverture. Enfin, de nombreuses études de cas ont été réalisées avec Fast. Nous décrivons plus particulièrement la vérification paramétrée du protocole de reprise sur panne TTP et du protocole de communication CES. Fast a été présenté dans [BFLP03].

Systèmes à pointeurs. Enfin, nous étudions la vérification de programmes manipulant de la mémoire dynamique. Nous définissons le modèle des systèmes à pointeurs que nous munissons d'une sémantique précise en termes de graphes mémoire. Nous proposons un cadre symbolique adapté à la vérification des systèmes à pointeurs : les états mémoire symboliques. Enfin nous définissons une abstraction, conservative vis à vis des propriétés de fuite mémoire et violation mémoire, qui permet de manipuler des structures avec de meilleures propriétés algorithmiques que les états mémoire symboliques. Ces résultats ont été publiés dans [BFN04].

Organisation du document

Le chapitre 2 présente le cadre de l'accélération plate. La composition d'accélérations est étudiée au chapitre 3. Le cadre de l'accélération plate est ensuite instancié aux systèmes à compteurs au chapitre 4. L'outil FAST est présenté au chapitre 5. Enfin dans le chapitre 6 nous proposons un cadre symbolique pour la vérification des systèmes à pointeurs.

Table des matières

Introduction **3**

1 Avant-propos **17**

1.1 Ensembles, relations, fonctions, ordres 17
1.1.1 Ensembles 17
1.1.2 Relations 17
1.1.3 Fonctions 18
1.1.4 Ordres et ensembles clos 18
1.2 Ensembles de nombres 18
1.2.1 Nombres, vecteurs et matrices 18
1.2.2 Ensembles Presburger-définissables 19
1.2.3 Polyèdres convexes 20
1.2.4 Ensembles clos de $\mathbb{N}^m$ 20
1.3 Langages 20
1.3.1 Mots 20
1.3.2 Langages 21
1.3.3 Sous-classes de langages réguliers 21

I Théorie de l'accélération **23**

2 Cadre de l'accélération **25**

2.1 Introduction 25
2.1.1 Cadre de l'accélération plate 25
2.1.2 Heuristique pour l'accélération plate 26
2.1.3 Applications 27
2.2 Systèmes et interprétations 27
2.2.1 Définitions 27
2.2.2 Sémantique 28
2.2.3 Propriétés de sûreté 29
2.2.4 Familles de systèmes 30

2.3 Cadre symbolique . . . 31
2.3.1 Définitions . . . 31
2.3.2 Limites de l'approche symbolique . . . 33
2.3.3 Procédure symbolique standard . . . 34
2.4 Accélération . . . 36
2.4.1 Niveaux d'accélération . . . 36
2.4.2 Exemples . . . 37
2.4.3 L'accélération plate comme meilleur compromis . . . 38
2.5 Accélération plate . . . 39
2.5.1 Expressions régulières linéaires restreintes . . . 39
2.5.2 Systèmes plats . . . 39
2.5.3 Applatissement de systèmes non plats . . . 40
2.5.4 Complétude de l'applatissement . . . 43
2.5.5 Le point sur les systèmes applatissables . . . 45
2.6 Procédure pour les systèmes applatissables . . . 45
2.6.1 Première procédure . . . 46
2.6.2 Raffinement . . . 47
2.6.3 Réduction du nombre de séquences utiles . . . 48
2.7 Conclusion . . . 50

3 Composition d'accélérations 51
3.1 Introduction . . . 51
3.1.1 Contexte . . . 51
3.1.2 Composition d'accélérations . . . 52
3.1.3 Approches existantes . . . 53
3.2 Systèmes faiblement hétérogènes . . . 53
3.3 Composition de cadres symboliques . . . 55
3.3.1 Produit cartésien de cadres symboliques . . . 55
3.3.2 Produit cartésien d'accélérations plates . . . 56
3.4 Composition synchronisée d'accélérations . . . 57
3.4.1 P-Cadre symbolique . . . 57
3.4.2 P-Accélérations . . . 58
3.4.3 Produits synchronisés . . . 59
3.4.4 Extensions . . . 62
3.4.5 P-cadres symboliques existants . . . 62
3.5 Conclusion . . . 63

II Vérification de systèmes à compteurs 65

4 Accélération de systèmes à compteurs 67
4.1 Introduction 67
4.1.1 Contexte 67
4.1.2 Accélération plate de systèmes à compteurs 68
4.1.3 Implantation efficace de l'accélération plate 69
4.2 Systèmes à compteurs affines 70
4.2.1 Systèmes à compteurs 70
4.2.2 Systèmes à compteurs affines 71
4.2.3 Monoïde d'un système à compteurs affine 71
4.2.4 Sur les systèmes à compteurs affines 72
4.3 Cadre symbolique : automates binaires 72
4.3.1 Automates binaires 73
4.3.2 Cadre symbolique 74
4.3.3 Complexité de la construction 75
4.3.4 Ensembles reconnaissables par automates binaires . . . 75
4.4 Accélération plate 76
4.4.1 Accélération plate pour fonctions Presburger-affines . . 76
4.4.2 Accélération plate convexe 78
4.4.3 Accélération plate de translations positives 81
4.4.4 Expérimentations 82
4.5 Réduction pour les systèmes à compteurs 84
4.5.1 Une réduction dédiée aux compteurs 84
4.5.2 Accélération de boucles imbriquées 85
4.5.3 Réduction de la longueur des cycles à considérer 86
4.5.4 Expérimentations 87
4.6 Conclusion 88

5 Fast et la vérification du TTP 89
5.1 Introduction 89
5.1.1 Architecture 90
5.1.2 Outils similaires 90
5.2 Moteur de calcul 91
5.2.1 Architecture logicielle 91
5.2.2 Adaptation de l'heuristique access3 91
5.2.3 Autres choix techniques 92
5.3 Entrées-sorties 93
5.3.1 Le système 93
5.3.2 La stratégie 93
5.3.3 Interface graphique 94

5.4 Expérimentations 94
5.4.1 À propos des tests 96
5.4.2 Le jeu de tests 96
5.4.3 Résultats 97
5.4.4 Validation de l'heuristique 98
5.5 Comparaison à d'autres outils 99
5.5.1 Les différents outils 99
5.5.2 Comparaison calcul en avant exact 101
5.5.3 Co-accessibilité et couverture 102
5.5.4 Commentaires 104
5.6 Vérification du protocole TTP 105
5.6.1 Présentation du protocole TTP 105
5.6.2 Modélisation 106
5.6.3 Vérification automatique pour une défaillance 108
5.6.4 Vérification pour deux défaillances 110
5.6.5 Résultats 113
5.6.6 Vérification avec ALV, LASH et TReX 113
5.7 Vérification du protocole CES 114
5.7.1 Le protocole CES 114
5.7.2 Simulation par système à compteurs 116
5.7.3 Correction de la modélisation des files 117
5.7.4 Vérification avec FAST 117
5.7.5 Résultats 118
5.7.6 Vérification avec ALV et LASH 118
5.8 Conclusion 119

III Vérification de systèmes à pointeurs 121

6 Cadre des systèmes à pointeurs 123
6.1 Introduction 123
6.1.1 Contexte 123
6.1.2 Cadre des systèmes à pointeurs 125
6.1.3 Approches existantes 126
6.2 Systèmes à pointeurs 127
6.2.1 Domaine d'interprétation : graphes mémoire 127
6.2.2 Actions des systèmes à pointeurs 128
6.2.3 Sémantique opérationnelle 129
6.2.4 Famille des systèmes à pointeurs 133
6.2.5 Propriétés des systèmes à pointeurs 133
6.3 Cadre symbolique : états mémoire symboliques 135

6.3.1 États mémoire symboliques 135
6.3.2 Isomorphisme d'états mémoire symboliques 138
6.3.3 Graphes mémoires minimaux 139
6.3.4 Forme atomique minimale 140
6.3.5 Forme minimale 141
6.3.6 Minimisation effective 141
6.3.7 Union, intersection, complément 143
6.3.8 Inclusion et test du vide 144
6.3.9 Successeurs symboliques 144
6.3.10 Propriétés 147
6.4 Abstraction de systèmes à pointeurs 148
6.4.1 Graphes mémoire abstraits 148
6.4.2 États mémoire symboliques abstraits 149
6.4.3 Graphes mémoire étanches 151
6.4.4 Graphes mémoire canoniques 152
6.4.5 Complément d'états mémoire symboliques abstraits . . 153
6.4.6 Résumé sur les états mémoire symboliques 154
6.5 Conclusion 155

Conclusion **157**

Bibliographie **160**

Chapitre 1

Avant-propos

1.1 Ensembles, relations, fonctions, ordres

1.1.1 Ensembles

Pour deux ensembles E et F, on note $E \cup F, E \cap F, E \backslash F$ et $E \times F$ respectivement l'union, l'intersection, la différence et le produit cartésien de E et F. L'ensemble $E^i, i > 0$, est défini inductivement par $E^1 = E$ et $E^{n+1} = E \times E^n$. On note $E \subseteq F$ si E est un sous-ensemble de F. L'ensemble vide est désigné par $\emptyset$. L'ensemble des parties de E sécrit 2^E, et l'ensemble des parties finies de E s'écrit $\mathcal{P}_f(E)$. Le cardinal d'un ensemble fini X est noté $|X|$.

1.1.2 Relations

Une *relation* $\mathcal{R}$ sur $E \times F$ est un ensemble $\mathcal{R} \subseteq E \times F$. On note $x \, \mathcal{R} \, x'$ quand $(x, x') \in \mathcal{R}$. La *relation inverse* de $\mathcal{R}$, notée $\mathcal{R}^{-1} \subseteq F \times E$, est définie par $(x', x) \in \mathcal{R}^{-1}$ si et seulement si $(x, x') \in \mathcal{R}$. L'image de $x \in E$ par $\mathcal{R}$ est un ensemble $\mathcal{R}(x) \subseteq F$ défini par $\mathcal{R}(x) = \{x' \in F \mid x\mathcal{R}x'\}$. Nous étendons la définition à $X \subseteq E$ par $\mathcal{R}(X) = \{x' \in F \mid \exists x \in X, x\mathcal{R}x'\}$. Étant données deux relations $\mathcal{R}_1 \subseteq E \times F$ et $\mathcal{R}_2 \subseteq F \times G$, la composition de $\mathcal{R}_1$ et $\mathcal{R}_2$, notée $\mathcal{R}_1 \bullet \mathcal{R}_2$, est la relation sur $E \times G$ définie par $x \, (\mathcal{R}_1 \bullet \mathcal{R}_2) \, x''$ si et seulement si il existe $x' \in F$ tel que $x \, \mathcal{R}_1 \, x'$ et $x' \, \mathcal{R}_2 \, x''$.

Une relation binaire sur E est une relation sur $E \times E$. On dira plus simplement une relation sur E. La relation identité Id_E sur E se définit par $Id_E = \{(x, x) \mid x \in E\}$. Une relation $\mathcal{R}$ sur E est *réflexive* si $Id_E \subseteq \mathcal{R}$; *transitive* si $x\mathcal{R}x'$ et $x'\mathcal{R}x''$ implique $x\mathcal{R}x''$ pour tout x, x', x'' ; *antisymétrique* si $x\mathcal{R}x'$ et $x'\mathcal{R}x$ implique $x = x'$ pour tout x, x'. La relation $\mathcal{R}^i$ est une

relation binaire définie inductivement par $\mathcal{R}^0 = Id_E$ et $\mathcal{R}^{i+1} = \mathcal{R} \bullet \mathcal{R}^i$. La *clôture* réflexive et transitive de $\mathcal{R}$, notée $\mathcal{R}^*$, est définie par $\mathcal{R}^* = \bigcup_{i \geq 0} \mathcal{R}^i$.

1.1.3 Fonctions

Une *fonction* f est une relation sur $E \times F$ telle que pour tout $x \in E$, il existe au plus un $x' \in F$ tel que $(x, x') \in f$. On note $f(x) = x'$. Une fonction est totale si pour tout $x \in E$, il existe $x' \in F$ tel que $f(x) = x'$. Sinon la fonction est dite partielle. Dans la suite les fonctions seront présumées totales, les fonctions partielles seront explicitement précisées.

Nous notons $f : E \to F$ une fonction sur $E \times F$. Si les ensembles auxquels appartiennent x et x' sont clairs, nous notons $f : x \mapsto x'$ la fonction définie par $f(x) = x'$, ou simplement $x \mapsto x'$ en suivant la notation fonctionnelle.

1.1.4 Ordres et ensembles clos

Un *pré-ordre partiel* $\preceq$ sur E est une relation réflexive et transitive sur E. Un *ordre partiel* sur E est un pré-ordre antisymétrique. Un *ordre total* $\preceq$ sur E est un ordre partiel sur E tel que pour tout $x, x' \in E$, on a soit $x \preceq x'$ soit $x' \preceq x$.

Un pré-ordre $\preceq$ sur E est un *beau pré-ordre* si de toute suite infinie (x_n) d'éléments $x_n \in E$, on peut toujours extraire une suite infinie (x_{n_i}) croissante pour $\preceq$, c'est-à-dire que pour tout $n_i, x_{n_i} \preceq x_{n_i+1}$. La définition s'étend aux beaux ordres.

Soit F muni d'un pré-ordre $\preceq$. Alors $G \subseteq F$ est *clos par le bas* si pour tout $x \in G$ et $y \in F$, $y \preceq x$ implique $y \in G$. On définit symétriquement les ensembles *clos par le haut* : pour tout $x \in G$ et $y \in F$, $y \succeq x$ implique $y \in G$.

1.2 Ensembles de nombres

1.2.1 Nombres, vecteurs et matrices

Nous désignons par $\mathbb{N}, \mathbb{Z}, \mathbb{R}$ et $\mathbb{R}_+$ l'ensemble des entiers naturels, l'ensemble des entiers relatifs, l'ensemble des nombres réels et l'ensemble des nombres réels positifs. L'ordre classique sur les nombres est noté $\leq$. C'est un ordre total. L'ordre $\leq$ sur $\mathbb{N}$ est un bel ordre total.

$\mathbb{N}^m$ (respectivement $\mathbb{Z}^m$ et $\mathbb{R}^m$) désignent les ensembles de vecteurs d'entiers naturels (respectivement d'entiers relatifs et de nombres réels) à m composantes. $\mathcal{M}_m(D)$ est l'ensemble des matrices carrées de dimension m à coefficients dans D, où D vaut $\mathbb{N}, \mathbb{Z}$ ou $\mathbb{R}$. Nous définissons l'ordre $\leq$ sur les vecteurs de nombres de manière standard : $(u_1, \ldots, u_m) \leq (v_1, \ldots, v_m)$ si pour toute composante i, $u_i \leq v_i$. La relation $\leq$ est un bel ordre (partiel) sur $\mathbb{N}^k$ et un ordre partiel sur $\mathbb{Z}^k$ et $\mathbb{R}^k$.

L'ensemble $\mathbb{N}$ est parfois étendu avec l'élément ∞ représentant intuitivement le point à l'infini. Nous notons $\mathbb{N}_\infty$ l'ensemble $\mathbb{N} \cup \{\infty\}$. Les opérations sur ∞ sont définies par : pour tout $n \in \mathbb{N}, n < \infty$ et pour tout $n \in \mathbb{N}_\infty, n + \infty = \infty$. Dans ce cas $\leq$ est toujours un bel ordre total sur $\mathbb{N}_\infty$ et un bel ordre partiel sur $\mathbb{N}_\infty^m$.

Soit $X \subseteq \mathbb{N}^m$. Un élément $x \in X$ est minimal s'il n'existe pas de $x' \in X \backslash \{x\}$ tel que $x' \leq x$. Les sous-ensembles de $\mathbb{N}^m$ vérifient la propriété suivante.

Lemme 1.2.1. Soit un ensemble $X \subseteq \mathbb{N}^m$, alors X a un ensemble fini d'éléments minimaux.

Démonstration. On raisonne par l'absurde en supposant que X a un ensemble infini d'éléments minimaux. On note $(m_i)_{i \in \mathbb{N}}$ une suite infinie formée avec les éléments minimaux de X. Comme $\leq$ est un bel ordre partiel sur $\mathbb{N}^m$, on extrait de $(m_i)_{i \in \mathbb{N}}$ une sous-suite croissante $(m_{i_k})_{k \in \mathbb{N}}$ de X. On a donc $m_{i_1} \leq m_{i_2}$, ce qui est une contradiction avec m_{i_2} est un élément minimal de X. □

1.2.2 Ensembles Presburger-définissables

La *logique de Presburger* est le fragment de l'arithmétique privée de la multiplication et de la division. On sait depuis longtemps que la satisfaisabilité d'une formule de Presburger est décidable. Soit un ensemble fini V de variables libres. L'ensemble des *formules de Presburger* ϕ sur V est définie par la grammaire suivante, où t est un terme et ϕ est une formule :

$$t ::= 0|1|v \in V|t - t|t + t$$
$$\phi ::= t \leq t|\neg\phi|\phi \vee \phi|\exists y; \phi|vrai$$

Cette logique est interprétée sur les vecteurs de nombres entiers ou réels. Dans ce qui suit, D désigne $\mathbb{N}, \mathbb{Z}$ ou $\mathbb{R}$. Pour une formule ϕ sur $|V|$ variables

libres, on note $[\![\phi]\!] \subseteq D^m$ l'ensemble des solutions de ϕ.

On dit qu'un ensemble $X \subseteq \mathbb{N}^m$ (resp. $\mathbb{Z}^m, \mathbb{R}^m$) est *Presburger-définissable* s'il existe une formule de Presburger à m variables libres telle que $X = [\![\phi]\!]$. Les ensembles Presburger-définissables sont clos entre autres par union, intersection et complément. Les ensembles Presburger-définissables correspondent aux ensembles semi-linéaires. On pourra consulter [BHMV94] pour plus de détails sur la logique de Presburger.

1.2.3 Polyèdres convexes

Un *polyèdre convexe* est une intersection finie de contraintes de la forme $\Sigma a_i v_i \# c$ où $\# \in \{<, >, \leq, \geq\}$, $a_i, v_i \in D^m$. Les polyèdres convexes sont donc définissables par un fragment de la logique de Presburger. Un polyèdre convexe P sur D^m vérifie : pour tout $x, x' \in P$, pour tout $t, t' \in \mathbb{R}$ tel que $t + t' = 1$, $tx + t'x' \in D^m$ implique $tx + t'x' \in P$. Intuitivement, on peut joindre deux points de P en ligne droite sans sortir de P.

1.2.4 Ensembles clos de $\mathbb{N}^m$

Nous définissons les deux restrictions suivantes de la logique de Presburger :

La *logique Clos par le Bas (CB) [BM99]* est le fragment de la logique de Presburger dans lequel les termes sont de la forme $t ::= x \leq c$, avec x une variable libre et $c \in \mathbb{N}^m_\infty$, et $\phi ::= t|\phi \wedge \phi|\phi \vee \phi|vrai|faux$.

La *logique Clos par le Haut (CH) [BM99]* est le fragment de la logique de Presburger dans lequel les termes sont de la forme $t ::= x \geq c$, avec x une variable libre et $c \in \mathbb{N}^m_\infty$, et $\phi ::= t|\phi \wedge \phi|\phi \vee \phi|vrai|faux$.

Les ensembles CB-définissables sont exactement les ensembles clos par le bas de $\mathbb{N}^m$ et les ensembles CH-définissables sont exactement les ensembles clos par le haut de $\mathbb{N}^m$ [BM99].

1.3 Langages

1.3.1 Mots

Un alphabet Σ est un ensemble fini quelconque. Une lettre $a \in \Sigma$ est un élément de Σ. Un *mot* w sur Σ est une séquence finie $w = a_1 \dots a_n$ de lettres de Σ. Le *mot vide* est noté ε. La *taille du mot* $w = a_1 \dots a_n$, notée $|w|$, vaut

$|w| = n$. Par convention $|\varepsilon| = 0$.

On note Σ^* l'ensemble de tous les mots sur Σ (y compris ε), Σ^k l'ensemble de tous les mots sur Σ de longueur k et $\Sigma^{\leq k}$ l'ensemble de tous les mots sur Σ de longueur inférieure ou égale à k.

On note $w_1.w_2$ la *concaténation des mots* w_1 et w_2. Pour tout $w \in \Sigma^*$, $w.\varepsilon = \varepsilon.w = w$. La relation $\leq$ sur Σ^* désigne l'ordre lexicographique usuel. L'ordre $\leq$ est un bel ordre (partiel) sur Σ^*.

1.3.2 Langages

Un *langage* $\mathcal{L}$ sur Σ est un ensemble $\mathcal{L} \subseteq \Sigma^*$. Soit $a \in \Sigma$. L'opération de concaténation s'étend aux langages par $w \in \mathcal{L}_1.\mathcal{L}_2$ si $w = w_1.w_2$ pour $w_1 \in \mathcal{L}_1$ et $w_2 \in \mathcal{L}_2$. On définit alors inductivement $\mathcal{L}^i$ par $\mathcal{L}^0 = \{\varepsilon\}$ et $\mathcal{L}^{i+1} = \mathcal{L}.\mathcal{L}^i$. Le langage $\mathcal{L}^*$ désigne $\mathcal{L}^* = \bigcup_{i\geq 0} \mathcal{L}^i$.

La classe des *langages réguliers* sur Σ est la plus petite classe de langages sur Σ contenant $\emptyset, \{\varepsilon\}$ ainsi que tous les $\{a\}$ pour $a \in \Sigma$ et close par union, concaténation et *.

Un langage régulier peut être représenté par une expression régulière $reg ::= \emptyset|\varepsilon|a \in \Sigma|reg.reg|reg + reg|reg^*$ (théorème de Kleene).

1.3.3 Sous-classes de langages réguliers

Les langages réguliers sont souvent utilisés pour leurs bonnes propriétés de clôture. Des sous-classes sont parfois préférées, soit pour garantir d'autres propriétés de clôture soit pour des raisons algorithmiques. Voici quelques classes citées dans la thèse.

Les *expressions régulières simples* ou sre [ACBJ04] sont des unions finies d'expressions de la forme $\Sigma_1(a_1 + \varepsilon) \dots \Sigma_n(a_n + \varepsilon)$ avec $a_i \in \Sigma \cup \{\varepsilon\}$ et $\Sigma_i \subseteq \Sigma$. Les sre définissent exactement la classe des langages réguliers clos par le bas et apparaissent naturellement dans l'étude de canaux de communication à pertes.

Les *expressions régulières semi-linéaires* ou slre [Yu97, FPS03] sont des unions finies d'expressions de la forme $u_1.w_1^* \dots u_n.w_n^*$ avec $u_i, w_i \in \Sigma^*$. Les slre sont exactement la classe des langages à densité polynomiale [Yu97]. Ils apparaissent dans l'étude de l'accélération plate.

Les *schémas de contraintes alphabétiques* ou apc [BMT01] sont des unions finies d'expressions de la forme $\Sigma_1 a_1 \ldots \Sigma_n a_n$ avec $a_i \in \Sigma$ et $\Sigma_i \subseteq \Sigma$. Les apc définissent des langages clos par semi-commutation et sont utilisés par exemple pour des systèmes distribués sur un anneau.

Première partie

Théorie de l'accélération

Chapitre 2

Cadre de l'accélération

2.1 Introduction

Nous nous concentrons dans ce chapitre sur deux problèmes clés de l'accélération : le manque d'unification des techniques et des outils d'une part, et le fossé entre les théorèmes d'accélération et leur utilisation dans des procédures de calcul de point fixe d'autre part.

2.1.1 Cadre de l'accélération plate

Nous proposons le *premier cadre théorique* pour le model checking avec accélération. Ce cadre s'articule autour de quatre concepts clés. Les deux premiers sont classiques et reprennent des définitions usuelles, les deux autres sont originaux.

Le *système* (définition 2.2.3) est l'abstraction mathématique modélisant les programmes à analyser. Nos systèmes sont composés d'une structure de contrôle et d'un nombre fini de variables, dont les valeurs sont modifiées par le franchissement des transitions.

Le *cadre symbolique* (définition 2.3.1) permet de manipuler des ensembles infinis de configurations via une représentation symbolique et des opérations de base.

L'*algorithme d'accélération* (définition 2.4.1) calcule tous les successeurs d'une configuration donnée par les séquences de transitions appartenant à un langage $\mathcal{L}$ sur l'alphabet T des transitions du système.

L'*heuristique de recherche* permet de relier l'algorithme d'accélération au calcul effectif de l'ensemble des configurations accessibles.

Nous distinguons trois niveaux d'accélération selon le type de langage $\mathcal{L}$ considéré : *l'accélération de boucle* si $\mathcal{L}$ est de la forme t^* avec $t \in T$, *l'accélération plate* si $\mathcal{L}$ est de la forme w^* avec $w \in T^*$ et *l'accélération globale* si $\mathcal{L}$ est une expression régulière quelconque sur T. Ces niveaux rendent compte de la plupart des résultats d'accélération sur des systèmes particuliers. On peut citer par exemple le cas des systèmes à files [BH99, ACBJ04] et des systèmes à compteurs [BW94, WB98, FL02]. La section 2.4.2 fait un bilan des nombreuses techniques d'accélération conformes à notre cadre.

D'un côté l'accélération de boucle apparaît insuffisante en pratique pour assurer la convergence et beaucoup de résultats d'accélération de boucle peuvent s'étendre à l'accélération plate. D'un autre côté, l'accélération globale assure systématiquement la convergence, et ne peut donc être obtenue pour des classes Turing-complètes. Or c'est précisément la vérification de telles classes qui nous intéresse. L'accélération plate apparaît donc comme le bon compromis.

2.1.2 Heuristique pour l'accélération plate

Nous développons ici de nouveaux concepts pour l'étude algorithmique de l'accélération plate.

Systèmes applatissables. Les notions *d'applatissement* (définition 2.5.2) et de *système applatissable* (définition 2.5.3) font le lien entre d'un côté les résultats d'accélération plate et de l'autre le calcul effectif de l'ensemble d'accessibilité. Nous montrons que l'ensemble d'accessibilité est calculable par accélération plate si et seulement si le système est applatissable (théorème 2.5.4). C'est la première caractérisation, complète, des systèmes calculables par accélération plate.

La majeure partie des systèmes analysés par des outils comme ALV, FAST, LASH ou TREX sont applatissables mais pas plats, ce qui souligne la pertinence de ce concept. De plus de nombreux résultats de calcul effectif d'ensembles d'accessibilité s'avèrent être des résultats d'applatissement et de nombreuses sous-classes standards sont applatissables, par exemple les VASS à deux compteurs [LS04], les automates temporisés [CJ99], les machines à compteurs reversal bornées, les VASS lossy et d'autres sous classes de VASS [LS05].

Procédures. La procédure ACCESS2 (procédure 2, section 2.6) est complète dans le sens où elle termine si et seulement si elle est appliquée à un

système applatissable (et pas seulement plat). C'est le premier résultat de complétude pour le model checking symbolique avec accélération.

La procédure ACCESS2 est générique et peut être instanciée de différentes manières. Nous proposons une instanciation ACCESS3 (procédure 3, section 2.6.2) visant à trouver efficacement un applatissement d'un système non plat, sans perdre la complétude. Il apparaît qu'un point clé de ACCESS3 est la *réduction* du nombre de circuits que la procédure doît considérer. Nous généralisons la notion de réduction introduite dans [FL02] et nous définissons deux nouvelles réductions indépendantes du domaine de données considéré.

2.1.3 Applications

Le cadre de l'accélération fournit une grille de lecture pour comparer les outils et méthodes entre eux. Notre cadre fournit également une liste de repères pour le développement d'un nouvel outil basé sur l'accélération. Les définitions, algorithmes et méthodes exposés dans ce chapitre seront repris tout au long de la thèse, en les spécialisant à des systèmes particuliers.

2.2 Systèmes et interprétations

Nous ne travaillons pas sur des programmes ou machines réelles, mais sur des abstractions que nous appelerons *systèmes*. Intuitivement un système est composé d'une structure de contrôle et d'un nombre fini de variables. Nous définissons les systèmes et leurs sémantiques dans cette section.

2.2.1 Définitions

Tout d'abord nous définissons l'aspect syntaxique des systèmes. En plus de la structure de contrôle, chaque système est caractérisé par des formules étiquetant les arcs. Ces formules obéissent à des règles syntaxiques et forment un ensemble récursif Φ.

Définition 2.2.1 (Système non interprété)**.** Un *système non interprété* S est un tuple $\mathsf{S} = (Q, \Phi, T)$, où Q est un ensemble fini de *locations*, Φ est un ensemble récursif de formules appelées *actions*, $T \subseteq Q \times \Phi \times Q$ est un ensemble fini de *transitions*.

Soit un *système non interprété* $\mathsf{S} = (Q, \Phi, T)$, les fonctions *source*, *cible* et *action* $\alpha : T \to Q$, $\beta : T \to Q$ et $l : T \to \Phi$ sont définies comme suit : pour une transition $t = (q, \sigma, q') \in T$, $\alpha(t) = q, \beta(t) = q', l(t) = \sigma$.

Nous voulons définir maintenant la sémantique opérationelle d'un système. Pour cela nous devons lui associer une *interprétation*, c'est-à-dire affecter un domaine D aux variables du système et une relation sur D à chaque formule.

Définition 2.2.2 (Interprétation)**.** Étant donné un ensemble récursif Φ de formules et un ensemble arbitraire D, une *interprétation* I de Φ sur D est un triplet $I = (\Phi, D, [\![\cdot]\!])$ tel que $[\![\cdot]\!] : \Phi \to 2^{D\times D}$ associe à chaque formule $\varphi \in \Phi$ une relation sur D.

Définition 2.2.3 (Système)**.** Un *système interprété* S (noté un système) est une paire (S, I) composée d'un système non interprété $\mathsf{S} = (Q, \Phi, T)$ et d'une interprétation $I = (\Phi, D, [\![\cdot]\!])$ de Φ, dénoté $S = (Q, \Phi, T, D, [\![\cdot]\!])$.

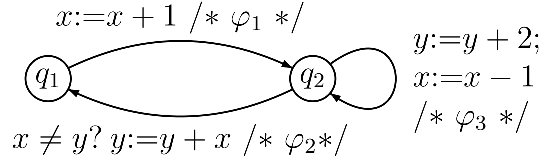


FIG. 2.1 – S_0, un premier système non interprété.

La figure 2.1 donne une représentation graphique d'un système non interprété S_0. L'ensemble des actions Σ n'est pas précisé dans l'exemple, mais le lecteur peut remarquer qu'il contient des affectations gardées par des expressions booléennes. $\varphi_1, \varphi_2, \varphi_3$ sont les trois actions effectivement utilisées dans S_0. Si l'on se donne comme domaine d'interprétation D les couples d'entiers $\mathbb{Z}^{\{x,y\}}$ (ou $\mathbb{Z}^2$), c'est-à-dire que l'on décide que x et y sont à valeurs dans $\mathbb{Z}$, une interprétation possible pour ces actions est la relation sur les couples d'entiers directement associée. Par exemple $[\![\varphi_2]\!] = \{((x, y), (x', y')) \mid x \neq y \wedge y' = y + x \wedge x' = x\}$. Ceci transforme S_0 en un système interprété S_0.

2.2.2 Sémantique

L'ensemble des *configurations* $\mathcal{C}_S$ de S est $Q \times D$. La sémantique d'une transition $t \in T$ est donnée par la relation $\xrightarrow{t}$ sur $\mathcal{C}_S$ definie par : $(q, d) \xrightarrow{t} (q', d')$ si $q = \alpha(t), q' = \beta(t)$ et $(d, d') \in [\![l(t)]\!]$. Cette définition peut être étendue à l'ensemble T^* des séquences de transitions. Alors $\xrightarrow{\varepsilon} = Id_{\mathcal{C}_S}$ et $\xrightarrow{t\cdot\pi} = \xrightarrow{t} \bullet \xrightarrow{\pi}$. Nous étendons aussi $\to$ aux *langages* $\mathfrak{L} \subseteq T^*$ par $\xrightarrow{\mathfrak{L}} = \bigcup_{\pi\in\mathfrak{L}} \xrightarrow{\pi}$. La fonction $[\![\cdot]\!]$ peut être étendue de la même façon à tout langage $\mathfrak{L} \subseteq \Phi^*$.

On étudie souvent le comportement d'un système S par rapport à un ensemble initial de configurations. On définit ainsi les *systèmes initialisés.*

Définition 2.2.4 (Système initialisé)**.** Un *système initialisé* avec X_0 est une paire (S, X_0) où $S = (Q, \Phi, T, D, [\![\cdot]\!])$ est un système sur I et $X_0 \subseteq \mathcal{C}_S$.

Remarque 2.2.1. Dans la suite nous désignons par système à la fois un système S et un système initialisé (S, X_0).

2.2.3 Propriétés de sûreté

Pour tout $X \subseteq \mathcal{C}_S$ et $\mathfrak{L} \subseteq T^*$, l'ensemble $\mathsf{post}_S(\mathfrak{L}, X)$ des configurations accessibles depuis X en utilisant des séquences de transitions dans $\mathfrak{L}$ est défini par $\mathsf{post}_S(\mathfrak{L}, X) = \{x' \in \mathcal{C}_S \mid \exists x \in X;\ (x, x') \in \xrightarrow{\mathfrak{L}}\}$. Nous nous concentrons sur deux ensembles en particulier. L'ensemble $\mathsf{post}_S(T, X)$ de toutes les configurations atteignables en un coup depuis X est noté $\mathsf{post}_S(X)$. L'ensemble $\mathsf{post}_S(T^*, X)$ de toutes les configurations atteignables depuis X, appelé *l'ensemble d'accessibilité de* X, est noté $\mathsf{post}^*_S(X)$.

Étant donné un ensemble de configurations initiales X_0, vérifier une propriété d'accessibilité P peut se faire comme suit : (1) calculer $\mathsf{post}^*_S(X_0)$, puis (2) décider si $\mathsf{post}^*_S(X_0) \subseteq P$. Nous nous concentrons dans cette thèse sur le calcul de l'ensemble d'accessibilité, qui est le problème central. Cet ensemble $\mathsf{post}^*_S(X_0)$ n'est pas récursif en général. Aussi nous ne pouvons espérer mieux que des procédures de calcul correctes, sans garantie théorique de terminaison mais efficaces en pratique.

Calcul en arrière. Une approche symétrique est de calculer “en arrière” l'ensemble des états co-accessibles $\mathsf{pre}^*_S(P)$ vérifiant la propriété P, et de décider si $X_0 \subseteq \mathsf{pre}^*_S(P)$. Au niveau d'abstraction de ce chapitre, les deux approches sont symétriques. Aussi nous ne traitons que le calcul en avant, et l'adaptation au calcul en arrière est directe.

Calcul de la relation d'accessibilité. Une troisième approche est de calculer *la relation d'accessibilité* $\xrightarrow{T^*}$. Il suffit ensuite de calculer l'ensemble d'accessibilité par la formule $X_0 \xrightarrow{T^*} \mathsf{post}^*(X_0)$. Le cadre que nous développons ici peut être étendu dans cette direction. Cependant cela nécessite des notations supplémentaires, aussi ce n'est pas traité dans cette thèse.

Remarque 2.2.2. Quand il n'y a pas d'ambiguité, le système S est omis dans les notations.

2.2.4 Familles de systèmes

On s'intéresse souvent non pas un système particulier mais à un ensemble de systèmes partageant une même interprétation. Nous introduisons à cet effet la notion de *famille de systèmes.*

Définition 2.2.5 (Famille de systèmes)**.** Étant donnée une interprétation $I = (\Phi, D, [\![\cdot]\!])$, *la famille de systèmes de* I, notée $\mathcal{F}(I)$, est la classe de tous les systèmes $S = (Q, \Phi, T, D, [\![\cdot]\!])$ avec I comme interprétation.

La définition 2.2.5 permet de retrouver un grand nombre de modèles classiquement étudiés.

Les machines de Minsky s'obtiennent en définissant le domaine d'interprétation $D = \mathbb{N}^{Var}$, où $Var = \{x_1, x_2, \ldots\}$ est un ensemble fini de variables, et Φ est l'ensemble des incrémentations "$x_i{:=}x_i + 1$", des décrémentations gardées "$x_i > 0?\, x_i{:=}x_i - 1$" et des tests à zéro "$x_i = 0?$". L'interprétation standard sur les entiers est associée.

Les systèmes à compteurs [BGP97, LS04] s'obtiennent en considérant le même domaine $D = \mathbb{N}^{Var}$ ou une variante $D = \mathbb{Z}^{Var}$, avec des relations définissables dans l'arithmétique de Presburger. De nombreuses restrictions existent, par exemple les systèmes à compteurs affines où les actions sont des transformations affines avec une garde Presburger-définissable [FL02, WB98] ; les machines à compteurs reversal bornées [ISD+02] où un compteur ne peut passer du mode incrément au mode décrément qu'un nombre borné de fois à l'avance ; les VASS et les réseaux de Petri dans lesquels les compteurs ne peuvent être testés à 0.

Les systèmes à pile pour lesquels le domaine est $D = \Gamma^*$, où Γ est l'alphabet de pile. Les actions ajoutent ou enlèvent des lettres sur le sommet de la pile.

Les systèmes à files [BZ83] avec un domaine $D = (\Gamma^*)^C$ où C est un ensemble fini de canaux de communication fifo[1], et Γ est un alphabet de file. Les actions ajoutent et enlèvent des messages selon une politique fifo. Les systèmes à files lossy sont une variante dans laquelle les canaux peuvent perdre spontanément des messages.

Les automates temporisés [AD94] considèrent le domaine $D = \mathbb{R}_+^{Var}$. Ici les actions sont gardées par des (in)égalités linéaires simples qui ne peuvent que remettre à 0 certaines variables (horloges). D'autres actions, implicites dans la présentation standard, modélisent l'écoulement du temps.

[1] *First-In First-Out*, pour premier arrivé, premier sorti.

Les systèmes hybrides [ACH+95] ont pour domaine $D = \mathbb{R}_+^{Var}$, les actions sont des fonctions affines avec des gardes convexes et les variables peuvent évoluer spontanément. Leur évolution n'est pas uniforme mais contrainte par la location courante.

La plupart des ces systèmes peuvent simuler une machine de Turing. On dit qu'ils sont Turing-complets. Évidement aucune propriété intéressante n'est décidable sur de tels systèmes. C'est le cas des machines de Minsky à au moins deux compteurs, des systèmes à compteurs (affines) à au moins deux compteurs, des systèmes à files et des systèmes hybrides. À l'inverse, les VASS, les machines reversal bornées, les systèmes à pile, les systèmes à files lossy et les automates temporisés ne sont pas Turing-complets car l'accessibilité (entre autres) est décidable dans ces modèles [May81, Iba78, BEM97, CFP96, AD94].

2.3 Cadre symbolique

2.3.1 Définitions

En pratique les procédures de model checking manipulent des représentations symboliques (appelées ici *régions*) représentant des ensembles (possiblement infinis) de configurations. La définition ci-dessous suit directement des idées exprimées par exemple dans [BJNT00, CC77, KMM+01].

Définition 2.3.1 (Cadre symbolique)**.** Un *cadre symbolique* est un tuple $(\Phi, D, [\![\cdot]\!]_1, L, [\![\cdot]\!]_2)$ où $I = (\Phi, D, [\![\cdot]\!]_1)$ est une interprétation, L est un ensemble de formules appelées *régions* et $[\![\cdot]\!]_2 : L \to 2^D$ est une fonction de *concrétisation des régions* tels qu'il existe une relation décidable $\sqsubseteq$ et des fonctions récursives $\sqcup$, $\textsc{post}$ vérifiant

1. il existe un élément $\bot \in L$ tel que $[\![\bot]\!]_2 = \emptyset$.
2. $\sqsubseteq \subseteq L \times L$ vérifie $\forall \mathtt{x}_1, \mathtt{x}_2 \in L$, $\mathtt{x}_1 \sqsubseteq \mathtt{x}_2$ si et seulement si $[\![\mathtt{x}_1]\!]_2 \subseteq [\![\mathtt{x}_2]\!]_2$.
3. $\sqcup : L \times L \to L$ vérifie $\forall \mathtt{x}_1, \mathtt{x}_2 \in L$, $[\![\mathtt{x}_1 \sqcup \mathtt{x}_2]\!]_2 = [\![\mathtt{x}_1]\!]_2 \cup [\![\mathtt{x}_2]\!]_2$.
4. $\textsc{post} : \Phi \times L \to L$ vérifie $\forall a \in \Phi, \forall \mathtt{x} \in L$, $[\![\textsc{post}(a, \mathtt{x})]\!]_2 = [\![a]\!]_1 ([\![\mathtt{x}]\!]_2)$.

Notation. Pour une interprétation $I = (\Phi, D, [\![\cdot]\!]_1)$ et un ensemble de régions L, la fonction de concrétisation $[\![\cdot]\!]_2$ est habituellement connue. Aussi nous désignerons par $[\![\cdot]\!]$ à la fois $[\![\cdot]\!]_1$ et $[\![\cdot]\!]_2$ et nous notons les cadres symboliques par $SF = (I, L)$. Dans la suite de ce chapitre, nous fixons un cadre symbolique arbitraire $SF = (I, L)$. Quand nous considérons un système S, si rien n'est spécifié nous supposerons que $S \in \mathcal{F}(I)$.

Remarque 2.3.1. Dans certains travaux, le cadre symbolique peut être affaibli. *Une inclusion faible* assure seulement : $\mathtt{x}_1 \sqsubseteq \mathtt{x}_2$ implique que $[\![\mathtt{x}_1]\!] \subseteq [\![\mathtt{x}_2]\!]$, mais pas l'inverse. Ainsi il se peut que l'on ne détecte pas une inclusion des concrétisations. Une *union faible* vérifie $[\![\mathtt{x}_1]\!] \cup [\![\mathtt{x}_2]\!] \subseteq [\![\mathtt{x}_1 \sqcup \mathtt{x}_2]\!]$, typiquement le widening en interprétation abstraite [CC77, ACH+95].

Exemples. Voici une liste de cadres symboliques bien connus, destinés aux familles de systèmes listées à la section 2.2.4.

Les langages réguliers sont utilisés pour représenter les ensembles de configurations de systèmes à pile [BEF+00], de protocoles distribués sur un anneau de taille arbitraire [KMM+01] et d'automates communicants [Pac87]. Des sous-classes de langages réguliers sont parfois préférées pour des raisons d'efficacité : langages clos par la relation sous-mot pour les systèmes à files lossy (sre [ACBJ04]) ou clos par semi-commutations (apc [BMT01]).

Les matrices de différences de bornes (dbm [AD94]) sont une représentation canonique des sous-ensembles convexes de $\mathbb{R}_+^{Var}$ définis par les contraintes diagonales et orthogonales simples des automates temporisés. Les matrices de différences de bornes paramétrées (pdbm [AAB00]) permettent de considérer des opérations et des ensembles plus complexes, mais comme la représentation utilise des formules d'arithmétique, l'inclusion est indécidable.

Les arbres de couverture partagés (cst [DRV04]) sont une représentation compacte des sous-ensembles de $\mathbb{N}^{Var}$ clos par le haut. Ces ensembles apparaissent naturellement dans l'analyse en arrière des protocoles broadcast [EN98, EFM99] et de plusieurs extensions monotones des réseaux de Petri.

Les (unions finies de) polyèdres convexes [ACH+95] sont des conjonctions d'inégalités linéaires définissant des sous-ensembles de $\mathbb{R}_+^{Var}$, et adaptés à l'analyse des systèmes hybrides.

Les diagrammes de décision numériques (ndd [BGP97, FL02]) sont des automates reconnaissant les ensembles semi-linéaires de $\mathbb{Z}^{Var}$ et sont utilisés dans l'analyse des systèmes à compteurs.

Les automates de vecteurs réels (rva [BBR97]) sont des automates de Büchi faibles reconnaissant des sous-ensembles de $\mathbb{R}_+^{Var}$ et sont utilisés dans l'analyse des systèmes hybrides linéaires.

Extension aux configurations. Soit un système S ayant un ensemble de locations Q et $X \subseteq \mathcal{C}_S$. L'ensemble $\mathsf{post}^*(X)$ est de la forme : $\bigcup_{q \in Q} \{q\} \times D_q$,

où les D_q sont inclus dans D. En faisant l'hypothèse d'un ordre implicite sur les locations $q_1, \ldots, q_{|Q|}$, nous travaillons sur des tuples de régions à valeurs dans $L^{|Q|}$. La fonction $[\![\cdot]\!]$ est étendue à $L^{|Q|}$ par $[\![(\mathtt{x}_1, \ldots, \mathtt{x}_{|Q|})]\!] = \bigcup_{i \leq |Q|} \{q_i\} \times [\![\mathtt{x}_i]\!]$. Toutes les opérations symboliques s'étendent à ces tuples. L'inclusion et l'union se font composante par composante.

L'opération POST est déjà étendue aux transitions par POST : $T \times L^{|Q|} \to L^{|Q|}$ où : $\text{POST}((q_i, a, q_j), (\mathtt{x}_1, \ldots, \mathtt{x}_{|Q|}))$ vaut $(\mathtt{x}'_1, \ldots, \mathtt{x}'_{|Q|})$ tel que $\mathtt{x}'_p = \perp$ si $p \neq j$, et $\mathtt{x}'_j = \text{POST}(a, \mathtt{x}_i)$. Puis POST est étendu aux séquences de transitions par induction sur la séquence. Nous définissons enfin POST : $L^{|Q|} \to L^{|Q|}$ par $\text{POST}((\mathtt{x}_1, \ldots, \mathtt{x}_{|Q|})) = \bigsqcup_{t \in T} \text{POST}(t, (\mathtt{x}_1, \ldots, \mathtt{x}_{|Q|}))$.

2.3.2 Limites de l'approche symbolique

Un sous-ensemble de configurations $X \subseteq \mathcal{C}_S$ est *L-définissable* s'il existe un tuple de régions $\mathtt{x} \in L^{|Q|}$ tel que $[\![\mathtt{x}]\!] = X$. Clairement, le calcul symbolique de $\mathsf{post}^*(X)$ est possible seulement si $\mathsf{post}^*(X)$ est L-définissable. Le problème "$\mathsf{post}^*([\![\mathtt{x}]\!])$ est-il L-définissable ?" est indécidable, même en se restreignant à des systèmes à deux compteurs avec des formules de Presburger.

Théorème 2.3.1. *Étant donnés le cadre symbolique des systèmes à deux compteurs associé aux formules de Presburger, un système à deux compteurs S et $\mathtt{x}_0 \in L^{|Q|}$, alors le problème de savoir si $\mathsf{post}^*([\![\mathtt{x}_0]\!])$ est L-définissable est indécidable.*

Démonstration. Nous réduisons le problème de l'accessibilité de location dans les systèmes à quatre compteurs, connu pour être indécidable. Le problème est le suivant. Nous considérons un système S_0 avec quatre variables variables x, y, y_o, z à valeurs dans $\mathbb{N}$, un ensemble fini de locations Q, et une configuration initiale $(q_0, c_0) \in Q \times \mathbb{N}^4$. Nous voulons savoir s'il existe une exécution de S_0 sur l'entrée (q_0, c_0) telle que q est atteint durant l'exécution. Supposons que pour tout (S', q', c') nous pouvons décider si $\mathsf{post}^*_{S'}((q', c'))$ est représentable par une formule de Presburger (Presburger-définissable). Nous rappelons ici que l'arithmétique de Presburger ne contient pas la multiplication de variables (typiquement $z = x \times y$). Nous réduisons le problème d'accessibilité de location comme suit. Nous transformons S_0 en S_1 en ajoutant un nombre fini de nouvelles locations Q_1 et nouvelles transitions sur Q_1, commençant sur $q_1 \in Q_1$, encodant la multiplication de x par y et assignant le résultat à z dans la location $q_z \in Q_1$ (la variable y_0 est utilisée pour garder la valeur de y durant l'opération). Puis nous ajoutons les transitions suivantes. Une transition $(q_0, \text{“}x := 0, y := 0, z := 0'', q_1)$, les transitions $(q_1, \text{“}y := y + 1\text{“}, q_1)$, $(q_1, \text{“}y := y + 1\text{“}, q_1)$, les transitions $(q, \text{“}x := x + 1\text{“}, q)$, $(q, \text{“}x := x - 1\text{“}, q)$, $(q, \text{“}y := y + 1\text{“}, q)$, $(q, \text{“}y := y - 1\text{“}, q)$ $(q, \text{“}z := z + 1\text{“}, q)$, $(q, \text{“}z := z - 1\text{“}, q)$ et pour tout $q'' \in Q_0 \cup Q_1$ une transition

$(q, \text{“}x := x, y := y, z := z\text{“}, q'')$. Il est alors facile de vérifier que $\mathsf{post}^*_{S_1}((q_0, c_0))$ est Presburger-définissable (et vaut alors $(Q_0 \cup Q_1) \times \mathbb{N}^4$) si et seulement si q est accessible (sinon la projection de l'ensemble d'accessiblité sur q_z est $\{(x, y, z) | z = x \times y\}$, ce qui n'est pas Presburger-définissable). □

Nous nous intéressons non pas à la définissabilité de l'ensemble $\mathsf{post}^*(X)$ mais à son calcul effectif.

Définition 2.3.2 (L-définissabilité effective)**.** Une fonction $f : 2^{\mathcal{C}_S} \to 2^{\mathcal{C}_S}$ est *effectivement L-définissable* s'il existe une fonction récursive $g_f : L^{|Q|} \to L^{|Q|}$ telle que $\forall \mathtt{x} \in L^{|Q|}, f([\![\mathtt{x}]\!]) = [\![g_f(\mathtt{x})]\!]$.

La proposition suivante montre que la L-définissabilité de $\mathsf{post}^*(X)$ est une condition nécessaire mais pas suffisante à son calcul effectif. Ce résultat découle directement de [BM99] et [May03].

Proposition 2.3.1. Nous considérons la famille des systèmes à files lossy et le cadre symbolique des langages réguliers clos par le bas (sre). Pour tout $\mathtt{x} \in \mathsf{sre}^{|Q|}$, l'ensemble $\mathsf{post}^*([\![\mathtt{x}]\!])$ est sre-définissable. Cependant la fonction $f : X \mapsto \mathsf{post}^*(X)$ n'est pas effectivement sre-définissable.

Démonstration. Dans [BM99] les auteurs montrent que $\mathsf{post}^*(X)$ n'est pas effectivement calculable pour des machines de Minsky à pertes (c'est-à-dire dont les compteurs peuvent être spontanément décrémentés). Dans [May03], l'auteur propose une simulation des machines de Minsky à pertes par des systèmes à files lossy. Cette simulation conserve l'ensemble des configurations accessibles. La combinaison des deux résultats permet de conclure. □

2.3.3 Procédure symbolique standard

Nous décrivons ici une première procédure de calcul symbolique des configurations accessibles en avant. La procédure ACCESS1 (procédure 1) est basée sur un calcul itératif direct du point fixe : la procédure collecte les successeurs avec POST et $\sqcup$, et teste si le point fixe est atteint avec $\sqsubseteq$. Cette procédure est par exemple utilisée dans l'outil ALV [Alv]. Nous considérons dans la suite que cette procédure est la procédure symbolique de référence, que nous voulons améliorer.

Nous pouvons déjà remarquer que cette procédure est *partiellement correcte*, c'est-à-dire que lorsqu'elle termine elle retourne bien une représentation de $\mathsf{post}^*([\![\mathtt{x}_0]\!])$.

Procédure ACCESS1($\mathtt{x}_0$)
entrée : $\mathtt{x}_0 \in L^{|Q|}$

1: $\mathtt{x} \leftarrow \mathtt{x}_0$
2: **while** POST($\mathtt{x}$) $\not\sqsubseteq \mathtt{x}$ **do**
3: $\quad \mathtt{x} \leftarrow$ POST($\mathtt{x}$) $\sqcup \mathtt{x}$
4: **end while**
5: retourner $\mathtt{x}$

Procédure 1: ACCESS1, procédure standard de model-checking symbolique

Théorème 2.3.2 (Correction partielle). *Si* ACCESS1($\mathtt{x}_0$) *termine, alors* $[\![\text{ACCESS1}(\mathtt{x}_0)]\!] = \mathsf{post}^*([\![\mathtt{x}0]\!])$.

Démonstration. Si la procédure termine, ACCESS1($\mathtt{x}_0$) est un point fixe de POST. Donc $[\![\text{ACCESS1}(\mathtt{x}_0)]\!]$ est un point fixe de post. De plus, à chaque itération de la procédure, $[\![\mathtt{x}]\!] \subseteq \mathsf{post}^*([\![\mathtt{x}_0]\!])$. Ceci assure que $[\![\text{ACCESS1}(\mathtt{x}_0)]\!]$ est le plus petit point fixe de post, c'est-à-dire $\mathsf{post}^*([\![\mathtt{x}0]\!])$. □

Nous voulons caractériser exactement la classe des systèmes pour lesquels la procédure ACCESS1 termine. Nous introduisons la notion de système L-uniformément borné.

Définition 2.3.3 (L-uniformément borné). Un système $(S, \mathtt{x})$ est *L-uniformément borné* si il existe $n_\mathtt{x} \in \mathbb{N}$ tel que pour tout $c_1 \in Q \times [\![\mathtt{x}]\!]$ et $c_2 \in Q \times D$, si $c_2 \in \mathsf{post}^*(\{c_1\})$ alors $c_2 \in \bigcup_{i \leq n_\mathtt{x}} \mathsf{post}^i(\{c_1\})$.

Le théorème suivant montre que ACCESS1 termine exactement sur la classe des systèmes L-uniformément bornés.

Théorème 2.3.3 (Terminaison). ACCESS1 *termine sur* $\mathtt{x}_0 \in L^{|Q|}$ *si et seulement si le système* $(S, \mathtt{x}_0)$ *est L-uniformément borné.*

Démonstration. Nous supposons que $(S, \mathtt{x}_0)$ est L-uniformément borné. Alors il existe $n_{\mathtt{x}_0}$ tel que $\mathsf{post}^*([\![\mathtt{x}_0]\!]) = \bigcup_{i \leq n_{\mathtt{x}_0}} \mathsf{post}^i([\![\mathtt{x}_0]\!])$. Donc ACCESS1($\mathtt{x}_0$) termine après au plus $n_{\mathtt{x}_0}$ itérations. Nous supposons maintenant que ACCESS1 termine sur l'entrée $\mathtt{x}_0$. Alors le point fixe est atteint après un nombre n_i d'itérations de la boucle **while**. n_i est la constante cherchée. □

Nous dirons qu'un système S est *L-uniformément borné* si pour tout $\mathtt{x} \in L^{|Q|}$, $(S, \mathtt{x})$ est L-uniformément borné. En adaptant la preuve du théorème 2.3.3, on peut montrer que ACCESS1 termine pour tout $\mathtt{x} \in L^{|Q|}$ si et seulement si le système S est L-uniformément borné.

Remarque 2.3.2. Si l'inclusion symbolique $\sqsubseteq$ ou l'union symbolique $\sqcup$ sont faibles, le théorème 2.3.3 ne s'applique pas.

Remarque 2.3.3. Les *systèmes de transitions bien structurés* [FPS03] avec des ensembles clos par le haut sont L-uniformément bornés *en arrière*. Cette classe contient notamment les réseaux de Petri et les systèmes à files lossy. Cependant, de nombreux systèmes ne sont pas L-uniformément bornés, par exemple le protocole TTP étudié au chapitre 5.

2.4 Accélération

Nous formalisons maintenant la notion d'accélération. Soit $\mathfrak{L}$ un langage sur l'alphabet des transitions T, et $X \subseteq \mathcal{C}_S$. Une accélération selon $\mathfrak{L}$ à partir de X calcule $X' \subseteq \mathcal{C}_S$ tel que $X \xrightarrow{\mathfrak{L}} X'$.

2.4.1 Niveaux d'accélération

Selon les techniques, le langage $\mathfrak{L}$ peut être plus ou moins complexe. Nous identifions trois niveaux d'accélération, selon que $\mathfrak{L} = t^*, t \in T^*$, ou $\mathfrak{L} = w^*, w \in T^*$ ou $\mathfrak{L}$ est un langage régulier sur T^*. Ces trois niveaux incluent la majeure partie des résultats d'accélération développés depuis [BW94].

Définition 2.4.1 (Accélération)**.** Un cadre symbolique SF admet

1. une *accélération de boucle* s'il existe une fonction récursive POST_STAR : $\Phi \times L \to L$ telle que $\forall a \in \Phi$, $\forall \mathtt{x} \in L$, $[\![\text{POST_STAR}(a, \mathtt{x})]\!] = [\![a^*]\!]\,([\![\mathtt{x}]\!])$;
2. une *accélération plate* s'il existe une fonction récursive POST_STAR : $\Phi^* \times L \to L$ telle que that $\forall \pi \in \Phi^*$, $\forall \mathtt{x} \in L$, $[\![\text{POST_STAR}(\pi, \mathtt{x})]\!] = [\![\pi^*]\!]\,([\![\mathtt{x}]\!])$;
3. une *accélération globale* s'il existe une fonction récursive POST_STAR : RegExp(Φ) $\times L \to L$ telle que pour toute expression régulière $\mathtt{e}$ sur Φ et pour tout $\mathtt{x} \in L$, $[\![\text{POST_STAR}(\mathtt{e}, \mathtt{x})]\!] = [\![\mathtt{e}]\!]\,([\![\mathtt{x}]\!])$.

Par exemple, dans la figure 2.1 page 28, l'accélération de boucle concerne seulement φ_3, et revient à calculer $X' = [\![\varphi_3]\!]^*\,(X) = \{(x', y') \in \mathbb{Z}^2 | \exists (x, y) \in X; \exists k \in \mathbb{N}; x' = x - k \wedge y' = y + 2 \cdot k\}$. L'accélération plate doît permettre de calculer entre autres $[\![(\varphi_1 \cdot \varphi_2)^*]\!]\,(X)$, $[\![(\varphi_1 \cdot \varphi_3 \cdot \varphi_2)^*]\!]\,(X)$, $[\![(\varphi_1 \cdot \varphi_3 \cdot \varphi_3 \cdot \varphi_2)^*]\!]\,(X)$, $[\![(\varphi_3 \cdot \varphi_2 \cdot \varphi_1)^*]\!]\,(X)$. L'accélération globale demande de calculer des entrelacements plus complexes de transitions tels que des boucles imbriquées comme $[\![(\varphi_1 \cdot \varphi_3^* \cdot \varphi_2)^*]\!]\,(X)$ (configurations (q_1, X') accessibles à partir de (q_1, X)).

Extensions à $L^{|Q|}$. Dans chaque cas, POST_STAR peut être étendu aux tuples de régions $L^{|Q|}$ pour gérer des transitions. Nous détaillons l'extension pour l'accélération plate. Nous étendons tout d'abord POST_STAR en POST_STAR : $T^* \times Q \times L \to Q \times L$. Soit une séquence de transitions $(q_1, \varphi_1, q_2) \cdot (q_3, \varphi_2, q_4) \cdot (q_5, \varphi_3, q_6)$. Si la séquence n'est pas une séquence valide du graphe de contrôle (c'est-à-dire si $q_2 \neq q_3$ ou $q_4 \neq q_5$), alors la relation associée est réduite à l'identité. Sinon, la séquence est valide et la séquence de transitions est équivalente à $(q_1, \varphi_1 \cdot \varphi_2 \cdot \varphi_3, q_6)$. Si cette séquence n'est pas un cycle (c'est-à-dire si $q_1 \neq q_6$), l'accélération revient à itérer la séquence simplement une fois (et à faire l'union des anciennes configurations). Nous utilisons les opérateurs POST et $\sqcup$. Enfin si la séquence est un cycle $\pi = (q_1, \varphi_1 \cdot \varphi_2 \cdot \varphi_3, q_1)$, alors POST_STAR$(\pi, (q, \mathtt{x})) = (q, \mathtt{x})$ si $q \neq q_1$, et $(q_1,$ POST_STAR$(\varphi_1 \cdot \varphi_2 \cdot \varphi_3, \mathtt{x}))$ sinon. Finalement POST_STAR est étendu en POST_STAR : $T^* \times L^{|Q|} \to L^{|Q|}$ de la même manière que POST à la section 2.3.1.

L'extension est la même pour l'accélération de boucle. Dans le cas de l'accélération globale, pour chaque couple de locations (q, q'), il faut considérer l'intersection du langage régulier e avec le langage régulier des séquences de transitions de q à q'.

Liens entre les niveaux d'accélération. Les niveaux d'accélération sont stricts, c'est-à-dire qu'il existe des cadres symboliques admettant une accélération de boucle mais pas d'accélération plate, et d'autres admettant une accélération plate mais pas d'accélération globale. Par exemple le cadre des machines de Minsky avec les intervalles de $\mathbb{N}$ admet une accélération de boucle mais pas d'accélération plate, car les intervalles ne sont pas clos par accélération d'une séquence d'instructions d'une machine de Minsky. En effet, considérons l'intervalle restreint au singleton $\{0\}$, et la séquence $x' := x + 1; x' := x + 1$. Alors l'ensemble d'accessibilité $2.\mathbb{N}$ n'est pas un intervalle.

2.4.2 Exemples

Les approches suivantes se conforment au cadre de l'accélération.

Accélération de boucle. Tous les cadres symboliques définis à partir des machines de Minsky et munis de formules d'intervalles ou de formules de Presburger, admettent une accélération de boucle.

Accélération plate. Les systèmes à compteurs (à monoïde fini) munis de formules de Presburger admettent une accélération plate [FL02, théorème 2]. D'autres exemples sont les systèmes à files avec les cqdd [BH99,

théorème 5.1], les systèmes à files *sans comptage* avec des slre [FPS03, théorème 5.2] ou qdd[BGWW97, théorème 6], les systèmes à files lossy avec des sre [ACBJ04, corollaire 6.5].

Accélération globale. Les systèmes à compteurs reversal bornés [ISD+02], les VASS à deux compteurs [LS04], les lossy VASS et d'autres sous-classes des VASS avec des formules de Presburger [LS05], les systèmes à pile avec des langages réguliers et les systèmes de réécriture semi-commutatifs avec des languages apc [BMT01] admettent une accélération globale.

Autres approches similaires. Les approches suivantes sont reliées de près à l'accélération puisqu'elles visent à calculer exactement l'ensemble d'accessibilité, même si elles ne rentrent pas dans le cadre défini ci-dessus. En effet, les méthodes utilisées ne dépendent pas vraiment de l'action φ mais essaient plutôt de détecter un incrément δ tel que $\mathtt{x}' = \mathtt{x} + \delta$. Si un tel δ est découvert, il est itéré et en général un test permet de vérifier que le résultat obtenu ne contient que des configurations accessibles. [BHJ03] étudie l'itération de transitions de systèmes hybrides linéaires sous certaines conditions. Le cadre symbolique sous-jacent est les rva. L'itération d'une transition termine, mais ne suffit pas pour assurer le calcul de l'ensemble d'accessibilité. [AAB00] étudie l'itération de transitions sur des systèmes temporisés à compteurs restreints. Le cadre symbolique sous-jacent est les pdbm. Le test de correction de l'itération est indécidable.

2.4.3 L'accélération plate comme meilleur compromis

L'accélération de boucles est assez facile à obtenir, cependant elle est souvent trop restreinte pour faire converger le calcul de l'ensemble des états accessibles. Une accélération plate est plus difficile à obtenir, et demande notamment de bonnes propriétés de clôture des ensembles L-définissables et des contructions plus complexes pour POST_STAR. Cependant la convergence est obtenue plus souvent. L'accélération globale est une propriété très forte qui assure le calcul de l'ensemble d'accessibilité pour toute région initiale. Bien sûr, cela implique qu'on ne peut espérer trouver d'accélération globale pour des systèmes Turing-complets.

Cette dernière remarque va à l'encontre de notre but, c'est-à-dire la vérification de propriétés de sûreté pour des systèmes infinis, puisque la plupart des systèmes listés à la section 2.2.4 sont Turing-complets. Aussi l'accélération plate apparaît comme le meilleur compromis pour nos objectifs. Nous nous concentrons donc sur l'accélération plate dans la suite.

2.5 Accélération plate

Nous voulons tout d'abord caractériser la classe des systèmes pour lesquels l'accélération plate suffit à assurer le calcul de l'ensemble des configurations accessibles.

2.5.1 Expressions régulières linéaires restreintes

L'accélération plate permet de calculer les configurations accessibles par des langages de chemins plus généraux que l'itération d'une séquence.

Soit un alphabet Σ, nous définissons *les expressions linéaires régulières restreintes* (rlre) sur Σ comme les expressions régulières ρ de la forme $u_1^* \dots u_n^*$, où $u_i \in \Sigma^*$. Cette notion est très proche des expressions régulières semi-linéaires étudiées dans [Fri00, FPS03]. L'accélération plate permet de calculer les configurations accessibles par des langages de chemins définis par des rlre sur T.

Proposition 2.5.1. Soit un système S admettant une accélération plate, alors pour tout rlre ρ sur T, la fonction $f : X \mapsto \mathsf{post}(\rho, X)$ est effectivement L-définissable.

Démonstration. On raisonne par induction sur ρ. Si $\rho = \varepsilon$ alors $[\![x]\!] = \mathsf{post}(\varepsilon, [\![x]\!])$ et la propriété est satisfaite. Si $\rho = u^* \cdot \rho_1$ où $u \in \Phi^*$, on applique l'hypothèse d'induction à $\mathsf{post}(\rho_1, [\![\text{POST_STAR}(u, x)]\!])$. □

2.5.2 Systèmes plats

Les systèmes plats, c'est-à-dire sans boucles imbriquées dans la structure de contrôle, sont une première classe intuitive de systèmes pour lesquels l'accélération plate suffit au calcul de l'ensemble d'accessibilité. Nous appelons *cycle élémentaire* sur T une séquence de transitions $(q_{e_1}, \varphi_1, q_{s_1}) \dots (q_{e_n}, \varphi_n, q_{e_1})$ telle que : $\forall i < n, q_{s_i} = q_{e_{i+1}}$ et tous les q_{e_i} et q_{e_j}, $i \neq j$, sont distincts. Un cycle élémentaire est donc une séquence de transitions bouclant sur une location q_{e_1}, telle qu'une location est visitée au plus une fois sauf q_{e_1} qui est visitée deux fois.

Définition 2.5.1 (Système plat [CJ98, Fri00, FPS03]). Un système non interprété $S = (Q, \Phi, T)$ est *plat* si pour toute location $q \in Q$, il existe au plus un cycle élémentaire contenant q. Un système $S = (\mathsf{S}, D, [\![\cdot]\!])$ est plat si S est plat.

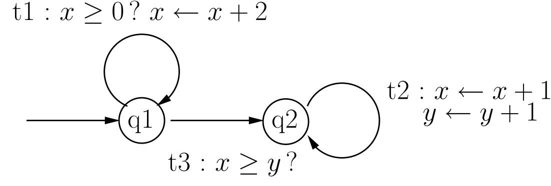


FIG. 2.2: Un système plat

Exemple 2.5.1. Soit le système plat présenté dans la figure 2.2. L'ensemble d'accessibilité peut toujours être calculé en procédant comme suit : d'abord itérer la transition t_1, puis déclencher la transition t_3 et enfin itérer la transition t_2. Par exemple, en partant de la région initiale $(\{x = 0 \wedge y = 0\}, q1)$, les étapes de calcul sont :
$(\{x = 0 \wedge y = 0\}, q1) \xrightarrow{t1^*} (\{\exists k, x = 2k \wedge y = 0\}, q1) \xrightarrow{t3} (\{\exists k, x = 2k \wedge y = 0\}, q2) \xrightarrow{t2^*} (\{\exists \theta, \exists k, x = 2k + \theta \wedge y = \theta\}, q2)$

Proposition 2.5.2. Soit un système plat S admettant une accélération plate, alors $f : X \mapsto \mathsf{post}^*(X)$ est effectivement L-définissable.

Démonstration. Il est clair que pour tout système plat S, il existe une expression régulière semi-linéaire (slre) ρ' sur T telle que pour tout $\mathrm{x} \in L^{|Q|}$, $\mathsf{post}^*([\![\mathrm{x}]\!]) = \mathsf{post}(\rho', [\![\mathrm{x}]\!])$. De plus ρ' est effectivement calculable. La slre ρ' a la forme $\rho' = \Sigma_i u_{i,1} w_{i,1}^* \dots u_{i,n} w_{i,n}^*$, où $u_{i,j}, w_{i,j} \in T^*$. Nous définissons la rlre ρ sur T par $\rho = \Pi_i u_{i,1}^* w_{i,1}^* \dots u_{i,n}^* w_{i,n}^*$. On peut vérifier que $\mathsf{post}(\rho, [\![\mathrm{x}]\!]) = \mathsf{post}(\rho', [\![\mathrm{x}]\!]) = \mathsf{post}^*([\![\mathrm{x}]\!])$. La proposition 2.5.1 permet de conclure. □

2.5.3 Applatissement de systèmes non plats

La plupart des systèmes intéressants ne sont pas plats. Le cœur du problème est donc de s'attaquer à des systèmes non plats, c'est-à-dire avec des boucles imbriquées.

Remarquons tout d'abord que si (1) nous connaissons un *système plat* S' tel que S et S' sont *équivalents pour l'accessibilité*, et (2) nous pouvons calculer $\mathsf{post}_S^*(c)$ à partir de $\mathsf{post}_{S'}^*(c)$, alors $\mathsf{post}_S^*(c)$ est calculable. Une façon de s'assurer des conditions (1) et (2) ci-dessus est de considérer des *applatissements* de S. La notion d'applatissement est une extension de la notion de dépliage, où l'on permet d'avoir un cycle élémentaire par location.

Définition 2.5.2 (Applatissement)**.** Un système $S' = (Q', \Phi, T', D, [\![\cdot]\!])$ est un applatissement d'un système $S = (Q, \Phi, T, D, [\![\cdot]\!])$ si

- S' est plat, et
- il existe une *fonction de repliage* $z : Q' \to Q$ telle que si $(q_1, w, q_2) \in T'$ alors $(z(q_1), w, z(q_2)) \in T$.

La figure 2.3 montre un système (fictif) S et l'un de ses applatissements. S a deux locations q_1, q_2 tandis que son applatissement a six locations $q'_1, q''_1, q'''_1, q'_2, q''_2, q'''_2$. La fonction z fait correspondre q'_1, q''_1 et q'''_1 à q_1, et q'_2, q''_2 et q'''_2 à q_2.

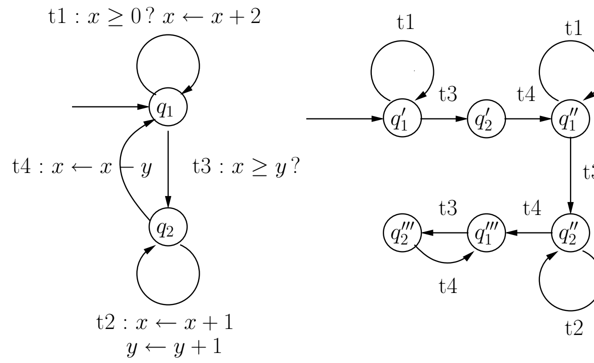


FIG. 2.3: Un système (gauche) et un de ses applatissements (droite)

Soit S un système et S' est l'un de ses applatissements. Nous étendons la fonction z aux configurations de S' par $z((q', x)) = (z(q'), x)$. L'extension de z à $X \subseteq \mathcal{C}_{S'}$ est définie par

$$z(\bigcup_{q' \in Q'} \{q'\} \times D_{q'}) = \bigcup_{q \in Q} \{q\} \times (\bigcup_{q' \in Q', z(q') = q} D_{q'})$$

Nous notons aussi $z^{-1} : \mathcal{C}_S \to \mathcal{C}_{S'}$ la fonction définie par :

$$z^{-1}(\bigcup_{q \in Q} \{q\} \times D_q) = \bigcup_{q' \in Q'} \{q'\} \times D_{z(q')}$$

z^{-1} n'est pas la fonction inverse de z puisque z n'est pas une bijection. Cependant z et z^{-1} vérifient $z^{-1} \bullet z = Id_{\mathcal{C}_S}$. Selon les définitions ci-dessus, z et z^{-1} peuvent être effectivement étendues aux tuples de régions sur $L^{|Q|}$, en $z : L^{|Q'|} \to L^{|Q|}$ et $z^{-1} : L^{|Q|} \to L^{|Q'|}$.

Enfin nous notons naturellement $z : T'^* \to T^*$ la fonction telle que $z(q'_1, \varphi, q'_2) = (z(q'_1), \varphi, z(q'_2))$. Cette fonction est étendue aux langages sur T'. Le résultat suivant provient directement des définitions.

Lemme 2.5.1. Soit un système $S = (Q, \Phi, T, D, [\![\cdot]\!])$ et $S' = (Q', \Phi, T', D, [\![\cdot]\!])$ un de ses applatissements. Alors pour tout langage $\mathcal{L}' \subseteq T'^*$ et pour tout $X \subseteq \mathcal{C}_S$, $z(\mathsf{post}_{S'}(\mathcal{L}', z^{-1}(X))) = \mathsf{post}_S(z(\mathcal{L}'), X))$.

Démonstration. Nous montrons tout d'abord que la propriété est vraie pour une transition $t' \in T'$. Notons $t' = (q'_1, \varphi, q'_2)$ avec $q'_1, q'_2 \in Q'$ et $\varphi \in \Phi$. Nous notons aussi $X = \bigcup_{q \in Q}\{q\} \times D_q$. Conformément à la définition de z^{-1}, $z^{-1}(X) = \bigcup_{q' \in Q'}\{q'\} \times D_{z(q')}$. Nous calculons tout d'abord $z(\mathsf{post}_{S'}(t', z^{-1}(X))$. $\mathsf{post}_{S'}(t', z^{-1}(X)) = \{q'_2\} \times \varphi(D_{z(q'_1)})$. Donc $z(\mathsf{post}_{S'}(t', z^{-1}(X))) = \{z(q'_2)\} \times \varphi(D_{z(q'_1)})$ (E_1). Nous calculons maintenant $\mathsf{post}_S(z(t'), X)$. Par définition, $z(t') = (z(q'_1), \varphi, z(q'_2))$. Donc on a l'égalité $\mathsf{post}_S(z(t'), X) = \{z(q'_2)\} \times \varphi(D_{z(q'_1)})$ (E_2). Les équations E_1 et E_2 prouvent l'égalité cherchée. Le résultat est étendu aux séquences de mots sur T'^* par récurrence, puis aux langages $\mathcal{L}' \subseteq T'^*$ par un raisonnement ensembliste. □

En conséquence pour tout $X \subseteq \mathcal{C}_S$, on a l'inclusion $z(\mathsf{post}^*_{S'}(X')) \subseteq \mathsf{post}^*_S(z(X'))$. Les applatissements intéressants sont ceux où il existe justement un X' tel que $z(\mathsf{post}^*_{S'}(X')) = \mathsf{post}^*_S(z(X'))$, c'est-à-dire ceux à partir desquels on peut calculer l'ensemble d'accessibilité du système non plat.

Définition 2.5.3 (L-applatissable). Soient un système $S = (Q, \Phi, T, D, [\![\cdot]\!])$ et une région $\mathtt{x} \in L^{|Q|}$. Le système $(S, [\![\mathtt{x}]\!])$ est *L-applatissable* s'il existe un applatissement $S' = (Q', \Phi, T', D, [\![\cdot]\!])$ de S et $\mathtt{x}' \in L^{|Q'|}$ tels que $z([\![\mathtt{x}']\!]) = [\![\mathtt{x}]\!]$ et $z(\mathsf{post}^*_{S'}([\![\mathtt{x}']\!])) = \mathsf{post}^*_S(z([\![\mathtt{x}']\!])) = \mathsf{post}^*_S([\![\mathtt{x}]\!])$. Un système S est L-applatissable si pour tout $\mathtt{x} \in L^{|Q|}$, $(S, \mathtt{x})$ est L-applatissable.

Nous montrons que tous les systèmes L-applatissables ont un ensemble d'accessibilité calculable par accélération plate.

Théorème 2.5.1. *Soit S un système admettant une accélération plate, et $\mathtt{x}_0 \in L^{|Q|}$. Si $(S, \mathtt{x}_0)$ est L-applatissable alors $\mathsf{post}^*([\![\mathtt{x}_0]\!])$ est effectivement L-définissable.*

Démonstration. Soit un système S et $\mathtt{x} \in L^{|Q|}$, nous énumérons tous les $(S', z, \mathtt{x}')$ tels que S' soit un applatissement de S (la fonction de mappage est z), et $\mathtt{x}' \in L^{|Q'|}$ tel que $[\![z(\mathtt{x}')]\!] = [\![\mathtt{x}]\!]$ (décidable grâce à $\sqsubseteq$). Puisque S' est plat nous pouvons calculer $\mathtt{x}''$ tel que $[\![\mathtt{x}'']\!] = \mathsf{post}^*_{S'}([\![\mathtt{x}']\!])$ (proposition 2.5.2). Ensuite nous calculons $\mathtt{y} = z(\mathtt{x}'') \in L^{|Q|}$ et testons si $\textsc{post}_S(\mathtt{y}) \sqsubseteq \mathtt{y}$. En cas de succès, $[\![\mathtt{y}]\!]$ est le point fixe de S. Puisque S est L-applatissable , un tel $\mathtt{y}$ existe et finira par être trouvé. La procédure est effective. □

Corollaire 2.5.1. Soit S un système admettant une accélération plate. Si S est L-applatissable alors $f : X \mapsto \mathsf{post}^*(X)$ est effectivement L-définissable.

2.5.4 Complétude de l'applatissement

Nous venons de montrer que l'ensemble d'accessibilité d'un système applatissable peut être calculé par accélération plate. Nous montrons maintenant le sens inverse : si l'on peut calculer l'ensemble d'accessibilité d'un système en utilisant seulement l'accélération plate, alors le système est applatissable.

Applatissabilité et langages réguliers restreints. La caractérisation précédente des systèmes applatissables oblige à manipuler des applatissements de systèmes. Nous donnons tout d'abord une nouvelle caractérisation d'applatissable, à base de rlre. En effet la proposition 2.5.2 montre que pour un système plat, l'ensemble d'accessibilité est effectivement L-définissable en utilisant la fonction d'accélération plate POST_STAR sur une rlre sur T. Le théorème 2.5.1 énonce que pour un système applatissable S, l'ensemble d'accessibilité est calculable en utilisant cette fois un applatissement de S. Le théorème suivant unifie les deux techniques en mettant en évidence le lien entre système applatissable et rlre sur T.

Théorème 2.5.2. *Soit un système $S = (Q, \Phi, T, D, [\![\cdot]\!])$ et $\mathrm{x}_0 \in L^{|Q|}$. (S, x_0) est L-applatissable si et seulement si il existe une rlre ρ sur T tel que* $\mathsf{post}^*([\![\mathrm{x}_0]\!]) = \mathsf{post}(\rho, [\![\mathrm{x}_0]\!])$.

Démonstration. Soit $\mathrm{x} \in L^{|Q|}$, s'il existe une rlre ρ_x telle que $\mathsf{post}_S^*([\![\mathrm{x}]\!]) = \mathsf{post}_S(\rho_\mathrm{x}, [\![\mathrm{x}]\!])$, nous en déduisons un applatissement S'_x de S Intuitivement le système non interprété sous-jacent S'_x est l'automate reconnaissant le langage $\rho_\mathrm{x} \subseteq T^*$.

Prouvons maintenant le sens opposé. Nous supposons que S est L-applatissable . Par définition il existe un système plat S', une fonction d'applatissage z et $\mathrm{x}' \in L^{|Q|}$ tels que : $z([\![\mathrm{x}']\!]) = [\![\mathrm{x}]\!]$ et $z(\mathsf{post}_{S'}^*([\![\mathrm{x}']\!])) = \mathsf{post}_S^*(\mathrm{x})$. De plus S', z, x' peuvent être construits effectivement (voir la preuve du théorème 2.5.1). On peut vérifier que si un tel x' existe, alors $z^{-1}(\mathrm{x}) \in L^{|Q'|}$ convient également et $z(z^{-1}([\![\mathrm{x}]\!])) = [\![\mathrm{x}]\!]$. Comme S' est plat, en utilisant la preuve de la proposition 2.5.2, nous déduisons l'existence effective d'une rlre ρ' sur T' tel que $\mathsf{post}_{S'}^*([\![z^{-1}(\mathrm{x})]\!]) = \mathsf{post}_{S'}(\rho', [\![z^{-1}(\mathrm{x})]\!])$. Soit $\rho = z(\rho')$. On vérifie que ρ est bien une rlre sur T car chaque transition d'un applatissement est une transition du système original, et la propriété s'étend aux séquences de transitions et aux langages. D'après le lemme 2.5.1, et comme $[\![z^{-1}(\mathrm{x})]\!] = z^{-1}([\![\mathrm{x}]\!])$, nous déduisons l'égalité $z(\mathsf{post}_{S'}(\rho', [\![z^{-1}(\mathrm{x})]\!])) = z(\mathsf{post}_{S'}(\rho', z^{-1}([\![\mathrm{x}]\!])) = \mathsf{post}_S(z(\rho'), [\![\mathrm{x}]\!])$. En utilisant les égalités précédentes, il vient que $\mathsf{post}_S^*([\![\mathrm{x}]\!]) = z(\mathsf{post}_{S'}^*([\![z^{-1}(\mathrm{x})]\!])) = \mathsf{post}_S(\rho, [\![\mathrm{x}]\!])$. La rlre ρ est bien la rlre cherchée. □

Corollaire 2.5.2. Soit un système $S = (Q, \Phi, T, D, [\![\cdot]\!])$. S est L-applatissable si et seulement si pour tout $\mathtt{x} \in L^{|Q|}$, il existe une rlre ρ sur T telle que

$\mathsf{post}^*([\![\mathrm{x}]\!]) = \mathsf{post}(\rho, [\![\mathrm{x}]\!])$.

Complétude Nous montrons maintenant la réciproque du théorème 2.5.1, à savoir si un système est calculable avec seulement l'accélération plate alors il est L-applatissable.

Théorème 2.5.3. *Soit S un système admettant une accélération plate et $\mathtt{x} \in L^{|Q|}$. Si $\mathsf{post}^*([\![\mathrm{x}]\!])$ est L-définissable et effectivement calculable (avec $\sqcup, \sqsubseteq, \mathrm{POST}, \mathrm{POST_STAR}$) alors $(S, \mathtt{x})$ est L-applatissable.*

Démonstration. Nous supposons que $\mathsf{post}^*([\![\mathrm{x}]\!])$ est L-définissable et effectivement calculable avec seulement $\sqcup, \sqsubseteq, \mathrm{POST}, \mathrm{POST_STAR}$. Donc le point fixe est atteint après un nombre fini d'applications des opérations $\sqcup, \mathrm{POST}(w_i, \mathtt{x})$ et $\mathrm{POST_STAR}(w_i, \mathtt{x})$, où les $w_i \in T^*$. On en déduit donc naturellement une slre ρ sur T telle que $\mathsf{post}(\rho, [\![\mathrm{x}]\!]) = \mathsf{post}^*([\![\mathrm{x}]\!])$. En utilisant un argument identique à celui de la proposition 2.5.2, nous en déduisons une rlre ρ' telle que $\mathsf{post}(\rho', [\![\mathrm{x}]\!]) = \mathsf{post}^*([\![\mathrm{x}]\!])$. Donc selon le théorème 2.5.2, le système $(S, \mathtt{x})$ est applatissable. □

Corollaire 2.5.3. Soit S un système admettant une accélération plate. Si pour tout $\mathtt{x} \in L^{|Q|}$, $\mathsf{post}^*([\![\mathrm{x}]\!])$ est L-définissable et effectivement calculable (avec $\sqcup, \sqsubseteq, \mathrm{POST}, \mathrm{POST_STAR}$) alors S est L-applatissable.

On peut regrouper les théorèmes 2.5.1, 2.5.2 et 2.5.3 dans le théorème 2.5.4. Ce résultat est la première caractérisation complète des systèmes pour lesquels l'accélération plate suffit à calculer l'ensemble d'accessibilité.

Théorème 2.5.4. *Soit un système S admettant une accélération plate, et $\mathrm{x}_0 \in L^{|Q|}$. Alors les trois propriétés suivantes sont équivalentes :*

1. $\mathsf{post}^*([\![\mathrm{x}_0]\!])$ *est calculable par accélération plate.*
2. (S, x_0) *est L-applatissable.*
3. *Il existe une* rlre ρ *sur T telle que* $\mathsf{post}^*([\![\mathrm{x}_0]\!]) = \mathsf{post}(\rho, [\![\mathrm{x}_0]\!])$.

On a aussi les équivalences exprimées dans le corollaire suivant.

Corollaire 2.5.4. Soit un système S admettant une accélération plate. Alors les trois propriétés suivantes sont équivalentes : (1) pour tout $\mathtt{x} \in L^{|Q|}$, $\mathsf{post}^*([\![\mathrm{x}]\!])$ est calculable par accélération plate ; (2) S est L-applatissable ; (3) pour tout $\mathtt{x} \in L^{|Q|}$, il existe une rlre ρ_{x} sur T telle que $\mathsf{post}^*([\![\mathrm{x}]\!]) = \mathsf{post}(\rho_{\mathrm{x}}, [\![\mathrm{x}]\!])$.

2.5.5 Le point sur les systèmes applatissables

L'accélération plate permet de calculer l'ensemble d'accessibilité des systèmes applatissables. Une question intéressante est alors de savoir si un système est applatissable où non. Malheureusement ce problème est indécidable. Le théorème suivant montre que l'indécidabilité est conservée même en se restreignant à des systèmes à deux compteurs.

Théorème 2.5.5. *Étant donné le cadre symbolique des systèmes à deux compteurs munis de formules de Presburger, savoir si un système S est L-applatissable est indécidable.*

Démonstration. La preuve est essentiellement la même que celle du théorème 2.3.1. On réduit de la même façon le problème de l'accessibilité de location. Remarquons que dans la réduction, q est accessible si et seulement si S_1 est L-applatissable . En effet si q est accessible, l'applatissement consiste à atteindre q, puis boucler sur q de façon à calculer $\mathbb{N}^4$ sur q et enfin utiliser les nouvelles transitions une fois chacune pour propager $\mathbb{N}^4$ sur chaque location $q'' \in Q \cup Q_1$. Si q n'est pas accessible, alors pour tout c_0 $\mathrm{post}^*_{S_1}(c)$ n'est pas Presburger-définissable, et donc S_1 ne peut être L-applatissable. □

Plusieurs familles de systèmes connues pour avoir un ensemble d'accessibilité L-définissable ont été démontrées L-applatissables. C'est le cas des VASS à deux compteurs [LS04], des automates temporisés [CJ99], des machines à compteurs k-reversal bornées, des VASS lossy et d'autres sous-classes de VASS [LS05]. Les systèmes L-uniformément bornés (voir la section 2.3.3) sont aussi tous L-applatissables. Il doit être clair qu'il n'y a pas d'équivalence entre avoir un ensemble d'accessibilité L-définissable et être L-applatissables. Par exemple les systèmes à files lossy.

Le problème suivant est ouvert.

Problème ouvert 2.5.1. Est-ce que les VASS semi-linéaires sont Presburger-applatissables ou non ?

2.6 Procédure pour les systèmes applatissables

La caractérisation précédente donne une construction effective de l'ensemble d'accessibilité pour tout système applatissable, en énumérant les rlre sur T jusqu'à trouver celle qui permet de calculer le point fixe. Cependant cette procédure requiert l'énumération de toutes les rlre sur T. Nous discutons dans cette section d'une implantation efficace en pratique du calcul de l'ensemble d'accessibilité pour les systèmes applatissables.

2.6.1 Première procédure

D'après le théorème 2.5.4, le calcul de point fixe pour les systèmes applatissables se réduit à explorer l'ensemble des rlre sur T. Plutôt que d'énumérer toutes les rlre, on peut construire une séquence croissante de rlre, comme dans la procédure 2.

Équité : nous faisons l'hypothèse que la sous-procédure Choisir sélectionne infiniment souvent chaque $w \in T^*$. C'est-à-dire que si Choisir est appelée infiniment souvent, alors chaque $w \in T^*$ est sélectionné infiniment souvent. Ceci peut être assuré par exemple en énumérant les mots de longueur inférieure à k donné, puis en itérant en incrémentant k.

procédure ACCESS2(x_0)
entrée : $\mathrm{x}_0 \in L^{|Q|}$

1: $\mathrm{x} \leftarrow \mathrm{x}_0$
2: **while** POST(x) $\not\sqsubseteq \mathrm{x}$ **do**
3: Choisir équitablement $w \in T^*$
4: $\mathrm{x} \leftarrow$ POST_STAR(w, x)
5: **end while**
6: retourner x

Procédure 2: ACCESS2, utilisation de l'accélération plate.

Tout d'abord nous montrons que ACCESS2 est partiellement correcte.

Théorème 2.6.1 (Correction partielle)**.** *Quand* ACCESS2 *termine, on a l'égalité* $[\![\text{ACCESS2}(\mathrm{x}_0)]\!] = \mathsf{post}^*([\![\mathrm{x}_0]\!])$.

Démonstration. La correction partielle vient directement des propriétés de POST_STAR et $\sqsubseteq$. □

Nous montrons maintenant que ACCESS2 termine si et seulement si le système est L-applatissable. Nous qualifions ACCESS2 de *complète* dans la mesure où le théorème 2.5.4 montre que l'accélération plate ne permet pas de calculer les ensembles d'accessibilité de systèmes non applatissables.

Théorème 2.6.2 (Complétude)**.** ACCESS2 *termine sur* $\mathrm{x}_0 \in L^{|Q|}$ *si et seulement si* (S, x_0) *est* L*-applatissable.*

Démonstration. Si ACCESS2 termine sur x_0 alors la séquence finie des $w \in T^*$ sélectionnés par Choisir fournit une rlre ρ sur T^* telle que $\mathsf{post}^*([\![\mathrm{x}_0]\!]) = \mathsf{post}(\rho, [\![\mathrm{x}_0]\!])$. Donc (S, x_0) est L-applatissable (théorème 2.5.2).

Supposons maintenant que (S, x_0) est L-applatissable. Il existe une rlre ρ sur T^* telle que post$^*([\![\mathrm{x}_0]\!])$ = post$(\rho, [\![\mathrm{x}_0]\!])$ (théorème 2.5.2). Nous notons $\rho = u_1^* \dots u_n^*$. Puisque Choisir est équitable, la séquence ρ' des w sélectionnés par Choisir finira par être de la forme $\rho' = w_1^* \dots w_m^*$ avec $m \geq n$, où il existe $i_1, \dots, i_n$ tels que $w_{i_1} = u_1, \dots, w_{i_n} = u_n$. Il vient que post$^*([\![\mathrm{x}_0]\!])$ = post$(\rho', [\![\mathrm{x}_0]\!])$, et la procédure s'arrête (et retourne le point fixe). □

Corollaire 2.6.1. ACCESS2 termine sur toute entrée si et seulement si S est L-applatissable.

Remarque 2.6.1. Les résultats précédents ne sont plus vrais si le cadre symbolique SF ne fournit qu'une inclusion faible, ou si POST_STAR n'est pas exacte mais une suraproximation.

2.6.2 Raffinement

Un aspect crucial de la procédure ACCESS2 est d'implanter Choisir de sorte que le point fixe soit atteint rapidement. La procédure 3 est une version raffinée de ACCESS2. Plutôt que de considérer toutes les séquences de transitions possibles dans T^*, nous nous restreignons aux séquences de longueur inférieure ou égale à une borne k. L'ensemble de ces séquences est noté $T^{\leq k}$, et nous appelons k-flattable la procédure ACCESS2 restreinte à $T^{\leq k}$. Si la recherche n'aboutit pas, elle finit par être interrompue, puis k est incrémenté et k-flattable est lancée de nouveau. La procédure Watchdog décide quand interrompre k-applatissable.

Équité : nous supposons qu'au cours d'une exécution infinie de ACCESS3*, la procédure* Watchdog *est appelée infiniment souvent. De plus, entre deux appels de* Watchdog*, chaque $w \in T^{\leq k}$ est sélectionné au moins une fois.*

Sous ces conditions d'équité, la procédure ACCESS3 ci-après reste correcte et complète.

Théorème 2.6.3 (Correction partielle)**.** ACCESS3 *est partiellement correcte : si* ACCESS3 *termine,* $[\![\text{ACCESS3}(\mathrm{x}_0)]\!]$ = post$^*([\![\mathrm{x}_0]\!])$.

Démonstration. La correction partielle vient directementde des définitions de POST_STAR et $\sqsubseteq$. □

Théorème 2.6.4 (Complétude)**.** ACCESS*3 termine sur* $\mathrm{x}_0 \in L^{|Q|}$ *si et seulement si* (S, x_0) *est L-applatissable.*

procédure ACCESS3($\mathtt{x}_0$)
entrée : $\mathtt{x}_0 \in L^{|Q|}$
1: $\mathtt{x} \leftarrow \mathtt{x}_0$; $k \leftarrow 0$
2: $k \leftarrow k+1$
3: **Lancer**
4: **Tant que** POST($\mathtt{x}$) $\not\sqsubseteq$ $\mathtt{x}$ **Faire** /* k-applatissable */
5: Choisir équitablement $w \in T^{\leq k}$
6: $\mathtt{x} \leftarrow$ POST_STAR($w, \mathtt{x}$)
7: **Fin tant que** /* end k-applatissable */
8: **Avec**
9: **Quand** Watchdog stop **goto** 2
10: **retourner** $\mathtt{x}$

Procédure 3: ACCESS3, raffinement de ACCESS2

Démonstration. Les hypothèses d'équité sur Choisir et sur Watchdog, et la réutilisation des calculs de chaque passage dans k-applatissable assurent l'équité de Choisir sur T^*. On adapte alors la preuve du théorème 2.6.2. □

Corollaire 2.6.2. ACCESS3 termine sur toute entrée si et seulement si S est L-applatissable.

Détails techniques. Pour réaliser une implantation, il faut également considérer les points suivants. (1) La *taille* des régions [2] calculées jusque là peut devenir telle que les calculs deviennent impossibles en pratique. (2) Un critère d'arrêt doît être choisi pour Watchdog. Le chapitre 5 décrit les choix effectués pour l'outil FAST sur les systèmes à compteurs. Les idées utilisées sont génériques, par contre leur efficacité dépend certainement du type de données considéré.

2.6.3 Réduction du nombre de séquences utiles

La limite principale de la procédure ACCESS3 est le cardinal de $T^{\leq k}$ exponentiel en k. Nous introduisons la notion de réduction pour compacter le nombre de transitions utiles au calcul de l'espace d'accessibilité.

Définition 2.6.1 (k-Réduction)**.** Étant donnée une interprétation $I = (\Phi, D, [\![\cdot]\!])$, une k-réduction $r : \mathcal{F}(I) \to \mathcal{F}(I)$ fait correspondre à chaque système $S = (Q, \Phi, T, D, [\![\cdot]\!]) \in \mathcal{F}(I)$ un système $S' = (Q, \Phi, T', D, [\![\cdot]\!]) \in \mathcal{F}(I)$ tel que :

[2] Chaque région a ses propres critères de taille. Par exemple pour des formules de Presburger, le nombre de nœuds de l'automate binaire associé est un facteur pertinent.

1. $\forall t' \in T', \xrightarrow{t'} \subseteq \xrightarrow{T^*}$,
2. $\forall w \in T^{\leq k}, \exists \rho \in \mathsf{rlre}(T').\ \xrightarrow{w^*} \subseteq \xrightarrow{\rho}$,
3. $|T'| \leq |T^{\leq k}|$.

On vérifie facilement le résultat suivant, qui justifie l'utilisation des k-réductions pour l'analyse des configurations accessibles.

Lemme 2.6.1. Les conditions 1 et 2 assurent que si S est L-applatissable avec une rlre ρ de taille k, alors S' est L-applatissable et a le même ensemble d'accessibilité.

Il est assez aisé de trouver des k-réductions, par exemple en enlevant les transitions qui ne font rien. En pratique une réduction n'est utile que si elle diminue fortement le nombre de transitions à considérer, et qu'elle a un coût de calcul négligeable. Nous définissons ci-après deux k-réductions. Ces k-réductions ont le double avantage d'être génériques (et donc appliquables pour tout domaine D) et efficaces en pratique.

Définition 2.6.2. Nous définissons les réductions suivantes :

1. La réduction par conjugaison : enlever les séquences de transitions de $T^{\leq k}$ equivalentes à conjugaison près à une séquence déjà listée (par exemple si l'on a $t_1 \cdot t_2 \cdot t_3$, on ne considère pas $t_2 \cdot t_3 \cdot t_1$).
2. La réduction par commutation : si t_1 et t_2 commutent, c'est-à-dire si $\xrightarrow{t_1} \bullet \xrightarrow{t_2} = \xrightarrow{t_2} \bullet \xrightarrow{t_1}$, alors on enlève $t_1 \cdot t_2$ et $t_2 \cdot t_1$ de $T^{\leq k}$.

Proposition 2.6.1. La réduction par conjugaison et la réduction par commutation sont des k-réductions. La réduction par conjugaison vérifie $|T'| = \mathcal{O}\frac{|T^{\leq k}|}{k}$.

Démonstration. *Réduction par conjugaison.* Soient les trois transitions t_1, t_2, t_3, alors $t_2 \cdot t_3 \cdot t_1$ et $t_3 \cdot t_1 \cdot t_2$ sont inutiles au calcul de l'ensemble d'accessibilité. En effet on peut remarquer que $\xrightarrow{(t_2 \cdot t_3 \cdot t_1)^*}$ et $\xrightarrow{(t_3 \cdot t_1 \cdot t_2)^*}$ se calculent facilement à partir de $\xrightarrow{(t_1 \cdot t_2 \cdot t_3)^*}$. Par exemple $\xrightarrow{(t_2 \cdot t_3 \cdot t_1)^*} = Id \cup \xrightarrow{t_2} \bullet \xrightarrow{t_3} \bullet \xrightarrow{(t_1 \cdot t_2 \cdot t_3)^*} \bullet \xrightarrow{t_1}$. Le cardinal $|T'|$ du système réduit vaut : $|T'| = \sum_{i \leq k} \frac{|T^i|}{i}$. Donc $|T'| = \mathcal{O}\frac{|T^k|}{k}$. Il vient que $|T'| = \mathcal{O}\frac{|T^{\leq k}|}{k}$. *Réduction par commutation.* Si t_1 et t_2 satisfont $[\![t_1 \cdot t_2]\!] = [\![t_2 \cdot t_1]\!]$ alors $\xrightarrow{(t_1 \cdot t_2)^*}$ est égal à $\xrightarrow{t_1^*} \bullet \xrightarrow{t_2^*}$, aussi nous pouvons ne pas considérer ni $t_1 \cdot t_2$ ni $t_2 \cdot t_1$. □

En plus de ces techniques génériques, il est souhaitable de développer des réductions adaptées à un cadre symbolique particulier. Le chapitre 4 étudie une réduction spécifique aux systèmes à compteurs affines.

2.7 Conclusion

Tout au long de ce chapitre nous avons développé un cadre pour le model-checking avec accélération. Ce cadre englobe la majeure partie des travaux d'accélération depuis [BW94]. Il donne des bases théoriques communes justifiant des techniques et outils existants.

De plus, alors que les travaux d'accélération antérieurs se sont surtout concentrés sur les théorèmes d'accélération proprement dits, nous avons étudié comment intégrer ces résultats d'accélération dans un calcul de point fixe. Nous avons caractérisé exactement l'ensemble des systèmes pour lesquels l'accélération plate suffit au calcul de l'ensemble d'accessibilité : les systèmes L-applatissables. Une heuristique correcte et complète a été proposée. Enfin les réductions permettent de compacter l'ensemble des transitions du système et donc de simplifier le calcul du point fixe. Nous avons défini deux techniques de réduction (par conjugaison ou par commutation) indépendantes du type de données considéré.

Ces résultats peuvent être utilisés pour comparer et améliorer les outils existants tels que TReX ou Lash, ou bien concevoir de nouveaux outils basés sur de nouveaux résultats d'accélération. Le reste de la thèse montrera comment instancier ce cadre plus particulièrement aux systèmes à compteurs, systèmes à pointeurs et aux systèmes manipulant plusieurs types de données.

Chapitre 3

Composition d'accélérations

3.1 Introduction

Les systèmes hétérogènes sont des systèmes manipulant des variables de différents types, par exemple des compteurs et des files. Nous nous intéressons dans ce chapitre à la composition automatique de cadres symboliques et d'accélérations sur les systèmes hétérogènes.

3.1.1 Contexte

Les programmes écrits en langages de haut niveau sont naturellement hétérogènes. On peut citer aussi les protocoles de transmission de messages sur réseaux non fiables comme le BRP étendu [AAB99]. Ces protocoles manipulent des canaux de transimissions (non fiables), des compteurs car le nombre de renvoies est limité et des horloges pour interrompre une transmission trop longue. La figure 3.1 montre une modélisation hétérogène du BRP (seul le client est représenté).

Bien souvent, la vérification pratique d'un système hétérogène se heurte au problème suivant : il existe des méthodes pour analyser le système projeté sur un type de données particulier, mais aucune méthode existante ne s'applique au système complet.

On procède alors généralement de deux manières : soit on abstrait le comportement du système en vérifiant chaque projection indépendamment, soit on développe une technique de vérification adaptée. La première approche calcule une surapproximation parfois trop grossière et la seconde approche est très coûteuse.

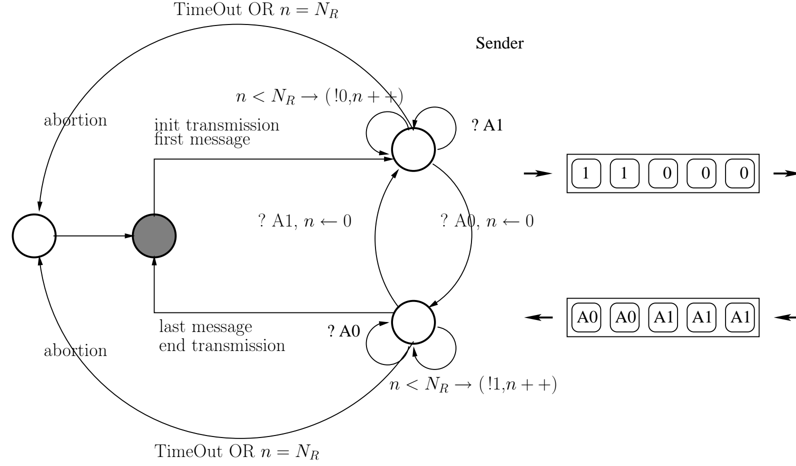


FIG. 3.1: Le protocole BRP étendu (vue du client)

Nous voulons dans ce chapitre construire un cadre symbolique et une accélération adaptés à un système hétérogène à partir des cadres symboliques et accélérations disponibles sur chacun des types de données du système. Notre approche est complètement automatique et est plus précise qu'une vérification par produit cartésien.

3.1.2 Composition d'accélérations

Nous introduisons la notion *d'interprétation faiblement hétérogène* (définition 3.2.2), dans laquelle les relations de transitions sont des produits cartésiens de relations sur chaque type de données. Le produit cartésien de cadres symboliques reste un cadre symbolique pour les systèmes faiblement hétérogènes (théorème 3.3.1). Malheureusement le produit cartésien d'accélérations n'est pas une accélération car il calcule une surapproximation des configurations accessibles (théorème 3.3.2).

Les P-cadres symboliques (définition 3.4.1) et les P-accélérations (définition 3.4.2) sont des classes de cadres symboliques et d'algorithmes accélération plate pour lesquelles on peut définir un produit synchronisé plus précis que le produit cartésien. Ces deux classes ont exactement les bonnes propriétés de composition souhaitées : le produit synchronisé de P-cadres symboliques est un P-cadre symbolique (théorème 3.4.1) et le produit synchronisé de P-accélérations est une P-accélération (théorème 3.4.2).

Enfin le produit synchronisé est effectif et les algorithmes d'accélérations

développés sur les automates binaires pour les compteurs et sur les cqdd pour les files ou les piles [Bou01] s'avèrent être des P-accélérations.

3.1.3 Approches existantes

La bibliothèque CSL [YKTB01] fournit des structures de données génériques pouvant être composées. Cependant comme l'accélération n'est pas prise en compte, la composition des différents types de données est un simple produit cartésien. La bibliothèque contient des booléens codés sous forme de bdd et des entiers codés par des automates binaires ou des formules.

L'outil TReX [ABS01] peut manipuler des systèmes avec files lossy, compteurs et horloges. L'accélération sur les horloges et compteurs est une accélération dédiée. Sur les systèmes à files lossy, horloges et compteurs TReX utilise un produit cartésien d'accélérations.

Dans [BH99] Bouajjani et Habermehl s'intéressent à l'accélération plate sur des systèmes à files FIFO munis du cadre des cqdd. Les notions de variables d'itération et de synchronisation d'accélérations sont présentes, mais mêlées aux algorithmes sur les cqdd. Ici nous clarifions les idées, en séparant ce qui est général de ce qui dépend des cadres symboliques particuliers utilisés, et nous généralisons à des types de données arbitraires.

3.2 Systèmes faiblement hétérogènes

Une interprétation $I = (\Phi, D, [\![\cdot]\!])$ est *hétérogène* si D est un produit cartésien. Nous introduisons dans cette section les interprétations faiblement hétérogènes, pour lesquels les relations $\varphi \in \Phi$ sont aussi des produits cartésiens de relations.

Nous définissons tout d'abord le produit d'interprétations.

Définition 3.2.1 (Produit d'interprétations)**.** Soit deux interprétations I_1 et I_2, notées $I_1 = (\Phi_1, D_1, [\![\cdot]\!]_1)$ et $I_2 = (\Phi_2, D_2, [\![\cdot]\!]_2)$. Le produit de I_1 et I_2, noté $I_1 \times I_2$, est l'interprétation définie par $I_1 \times I_2 = (\Phi_1 \times \Phi_2, D_1 \times D_2, [\![\cdot]\!]_1 \times [\![\cdot]\!]_2)$.

Une interprétation faiblement hétérogène est un produit d'interprétations.

Définition 3.2.2 (Interprétation faiblement hétérogène)**.** Soit $n > 1$ et $D_1, \ldots, D_n$ des ensembles. Une interprétation $\mathcal{H}$ est faiblement hétérogène sur $D_1 \times \ldots \times D_n$ si il existe n interprétations $I_i = (\Phi_i, D_i, [\![\cdot]\!]_i)$ telles que $\mathcal{H} = I_1 \times \ldots \times I_n$.

Notation. Dans la suite nous dirons qu'une interprétation est faiblement hétérogène, sans forcément faire référence aux domaines $D_1, \ldots, D_n$.

Un système est faiblement hétérogène s'il est construit sur une interprétation faiblement hétérogène. Intuitivement, un système faiblement hétérogène manipule différents domaines ayant chacun leurs propres actions. Les actions du système sont alors des produits d'actions sur chaque domaine : si une des actions ne peut être réalisée alors la transition entière est invalidée. Par exemple si D_1 est un ensemble de files et D_2 un ensemble de compteurs, on peut utiliser des actions de la forme "si $(x \geq 0)$ alors : décrémenter x et envoyer un message a dans la file f".

En revanche les systèmes faiblement hétérogènes ne permettent pas d'opérations liant fortement des variables appartenant à deux domaines D_1 et D_2 distincts : par exemple si D_1 est un ensemble de files et D_2 un ensemble de compteurs, on ne peut pas écrire la valeur du compteur x dans une file ; ou si D_1 est un ensemble d'horloges et D_2 un ensemble de compteurs, on ne peut comparer la valeur d'une horloge c à la valeur d'un compteur x.

Commentaires. Dans la définition 3.2.2, la partition du domaine en D_i ne se fait pas forcément par rapport au type des variables (avec un domaine D_i pour chaque type). La partition du domaine se fonde au contraire sur les opérations possibles entre les variables. Les variables d'un même domaine D_i ont des interactions arbitraires, alors que les variables de domaines différents ont des interactions plus limitées. Ainsi deux ensembles D_i peuvent représenter le même type, par exemple on peut avoir un compteur x dans D_1 et deux compteurs y, z dans D_2. Les variables x et y auront alors des interactions plus limitées que les variables y et z. De même rien n'interdit de regrouper des horloges et des compteurs dans un même D_i.

Cette partition peut paraitre arbitraire puisqu'on pourrait regrouper toutes les variables dans un seul domaine. En fait une telle partition prend tout son sens pour la composition de cadres symboliques. Dans ce cas chaque D_i correspondra à un groupe de variables pour lequel il existe un cadre symbolique adapté. Comme certains cadres symboliques considèrent plusieurs types de données (typiquement horloges et compteurs), la partition par types de données n'était pas pertinente.

Les systèmes faiblement hétérogènes sont courants dans la littérature. Par exemple les systèmes à files et les réseaux de Petri peuvent être vus comme des systèmes faiblement hétérogènes où chaque D_i représente une seule variable (contenu de file pour les systèmes à files, compteur pour les

réseaux de Petri). Les réseaux de Petri temporisés [Mer74] et le protocole BRP étendu [AAB99] sont également des systèmes faiblement hétérogènes. Par contre les systèmes à compteurs ne sont pas faiblement hétérogènes car leurs actions font intervenir par exemple des sommes de variables.

3.3 Composition de cadres symboliques

Nous étudions dans cette section les produits cartésiens de cadres symboliques et d'accélérations plates.

3.3.1 Produit cartésien de cadres symboliques

Nous dirons qu'un cadre symbolique est un *cadre symbolique faible* s'il vérifie toutes les hypothèses d'un cadre symbolique (voir la définition 2.3.1) sauf celle concernant l'inclusion. Cette hypothèse est remplacée dans ce cas par : pour toutes régions $\mathtt{x}, \mathtt{x}'$, si $\mathtt{x} \sqsubseteq \mathtt{x}'$ alors $[\![\mathtt{x}]\!] \subseteq [\![\mathtt{x}']\!]$.

Notation. Tous les cadres symboliques manipulés dans ce chapitre sont faibles. Aussi nous parlerons simplement de cadres symboliques.

Rappelons que pour un ensemble X quelconque, la notation $\mathcal{P}_f(X)$ désigne l'ensemble des parties finies de X. Nous définissons le produit cartésien de cadres symboliques.

Définition 3.3.1 (Produit cartésien de cadres symboliques)**.** Soit deux cadres symboliques $SF_1 = (\Phi_1, D_1, [\![\cdot]\!]_1, L_1)$ et $SF_2 = (\Phi_2, D_2, [\![\cdot]\!]_2, L_2)$. Le produit cartésien de SF_1 et SF_2, noté $SF_1 \times SF_2$, est le cadre symbolique défini par $SF_1 \times SF_2 = (\Phi_1 \times \Phi_2, D_1 \times D_2, [\![\cdot]\!]_1 \times [\![\cdot]\!]_2, \mathcal{P}_f(L_1 \times L_2))$.

Le produit cartésien est muni des opérations suivantes : l'opération $\textsc{post}$ est définie en effectuant les opérations de $\textsc{post}_i$ composante par composante. On note $\textsc{post} = \textsc{post}_1 \times \textsc{post}_2$. L'opération $\sqsubseteq$ est la conjonction des $\sqsubseteq_i$ sur chaque composante. Ces opérations sont ensuite étendues classiquement à l'ensemble $\mathcal{P}_f(L_1 \times L_2)$. Finalement l'union $\sqcup$ est l'union sur $\mathcal{P}_f(L_1 \times L_2)$.

Le théorème 3.3.1 montre que le produit cartésien de cadres symboliques est un cadre symbolique sur l'interprétation faiblement hétérogène sous-jacente.

Théorème 3.3.1. *Soit $\mathcal{H} = I_1 \times \ldots \times I_n$ une interprétation faiblement hétérogène. On suppose que chaque I_i est munie d'un cadre symbolique SF_i. Alors $SF_1 \times \ldots SF_n$ est un cadre symbolique pour $\mathcal{H}$.*

Démonstration. On raisonne sur deux cadres symboliques SF_1 et SF_2. La preuve du cas général s'obtient par récurrence. On note L_1 et L_2 les ensembles de régions de SF_1 et SF_2. Nous raisonnons sur les mots $(w_1, w_2) \in L_1 \times L_2$. L'extension à $\mathcal{P}_f(L_1 \times L_2)$ se fait en distribuant les opérations sur les (w_1, w_2). Les trois propriétés à démontrer sont que pour tout $(w_1, w_2), (w_1, w_2) \in L_1 \times L_2$ et pour tout $\varphi \in \Phi_1 \times \Phi_2$ on a : (1) $[\![(w_1, w_2) \sqcup (w'_1, w'_2)]\!] = [\![(w_1, w_2)]\!] \cup [\![(w'_1, w'_2)]\!]$; (2) si $(w_1, w_2) \sqsubseteq (w'_1, w'_2)$ alors $[\![(w_1, w_2)]\!] \subseteq [\![(w'_1, w'_2)]\!]$ (nous considérons des cadres affaiblis) ; et enfin (3) $[\![\text{POST}((\varphi_1, \varphi_2), (w_1, w_2))]\!] = \varphi_1 \times \varphi_2([\![(w_1, w_2)]\!])$. La propriété (1) découle directement des définitions de $\sqcup$ et $[\![\cdot]\!]$. Montrons la propriété (2). Par définition $(w_1, w_2) \sqsubseteq (w'_1, w'_2)$ implique que l'on ait $w_1 \sqsubseteq w'_1$ et $w_2 \sqsubseteq w'_2$. Comme SF_1 et SF_2 sont des cadres symboliques, on en déduit que $[\![w_1]\!] \subseteq [\![w_1]\!]'$ et $[\![w_2]\!] \subseteq [\![w'_2]\!]$. Donc $[\![w_1]\!] \times [\![w_2]\!] \subseteq [\![w'_1]\!] \times [\![w'_2]\!]$. On en déduit que la propriété (2) est vérifiée. Pour la propriété (3) : $\text{POST}((\varphi_1, \varphi_2), (w_1, w_2))$ est égal par définition à $(\text{POST}_1(\varphi_1, w_1), \text{POST}_2(\varphi_2, w_2))$. Donc en utilisant la définition de $[\![\cdot]\!]$ il vient que $[\![\text{POST}((\varphi_1, \varphi_2), (w_1, w_2))]\!] = [\![\text{POST}_1(\varphi_1, w_1)]\!] \times [\![\text{POST}_2(\varphi_2, w_2)]\!]$. Or SF_1 et SF_2 sont des cadres symboliques donc $[\![\text{POST}_i(\varphi_i, w_i)]\!] = \varphi_i([\![w_i]\!])$. On en déduit que $[\![\text{POST}((\varphi_1, \varphi_2), (w_1, w_2))]\!] = \varphi_1([\![w_1]\!]) \times \varphi_2([\![w_2]\!])$, ce qui est égal à $\varphi_1 \times \varphi_2([\![(w_1, w_2)]\!])$. □

Avec ce résultat nous pouvons lancer un calcul itératif de point fixe en réutilisant simplement les cadres symboliques de chaque type de données. C'est ce que propose la bibliothèque générique CSL [YKTB01] par exemple. Ceci correspond tout à fait à notre but initial et renforce la notion de système faiblement hétérogène comme notion centrale dans l'analyse par composition.

3.3.2 Produit cartésien d'accélérations plates

La prochaine étape consiste à combiner des algorithmes d'accélération plate POST_STAR_i sur des interprétations I_i pour créer un algorithme d'accélération plate POST_STAR sur l'interprétation produit $I_1 \times \ldots \times I_n$.

On se dirige naturellement vers le produit d'accélérations plates. Malheureusement le théorème 3.3.2 montre que le produit cartésien ne conserve pas l'accélération plate.

Théorème 3.3.2. *Soit $\mathcal{H} = I_1 \times \ldots \times I_n$ une interprétation faiblement hétérogène telle que tout I_i est muni d'un cadre symbolique SF_i admettant une accélération plate POST_STAR_i. Alors le produit cartésien des accélérations plates $\text{POST_STAR}_1 \times \ldots \times \text{POST_STAR}_n$ n'est pas une accélération plate pour le cadre symbolique $SF_1 \times \ldots \times SF_n$.*

Démonstration. Soit $\varphi \in \Phi_1 \times \ldots \times \Phi_n$ et $w \in L_1 \times \ldots \times L_n$. On note $\varphi =$

$\varphi_1 \times \ldots \times \varphi_n$ et $w = (w_1, \ldots, w_n)$. Soit E_1 et E_2 les ensembles définis par : E_1 est l'ensemble $\varphi^*([\![w]\!])$, et E_2 est l'ensemble $[\![\times\text{POST_STAR}_i(\varphi, w)]\!]$. Alors par définition $E_1 = \bigcup_{i\geq 0} \varphi^i([\![w]\!])$ et donc $E_1 = \bigcup_{i\geq 0} \varphi_1^i([\![w_1]\!]) \times \ldots \times \varphi_n^i([\![w_n]\!])$. Par définition de $\times\text{POST_STAR}_i$, on a $E_2 = [\![\text{POST_STAR}_1(\varphi_1, w_1)]\!] \times \ldots \times [\![\text{POST_STAR}_n(\varphi_n, w_n)]\!]$. Comme les fonctions POST_STAR_i sont des accélérations il vient que $E_2 = \varphi_1^*([\![w_1]\!]) \times \ldots \times \varphi_n^*([\![w_n]\!])$. On en déduit l'expression suivante pour E_2 : $E_2 = \bigcup \varphi_1^{i_1}([\![w_1]\!]) \times \ldots \times \bigcup \varphi_n^{i_n}([\![w_n]\!])$. Il vient que $E_1 \subseteq E_2$ mais en général l'autre sens n'est pas vrai. Donc $E_1 \neq E_2$, d'où la conclusion. □

La preuve du théorème 3.3.2 permet de montrer que le produit cartésien d'accélération est une surapproximation de l'accélération. Ce produit d'accélérations est utilisée par exemple dans TReX pour vérifier des systèmes avec compteurs, horloges et files lossy.

3.4 Composition synchronisée d'accélérations

Nous définissons dans cette section les Presburger-cadres symboliques (notés P-cadres symboliques) et les Presburger-accélérations (notés P-accélérations) et nous les munissons de produits synchronisés. Nous montrons que le produit synchronisé de P-accélérations demeure une P-accélération.

3.4.1 P-Cadre symbolique

L'idée principale est d'utiliser des cadres symboliques basés sur une représentation L combinant un mot w sur un alphabet fini Σ et une formule de Presburger encodant des contraintes arithmétiques.

Définition 3.4.1 (P-cadre symbolique)**.** Soit I une interprétation. Un P-cadre symbolique pour I est un cadre symbolique $P = (I, \mathcal{P}_f(L))$ tel que :

1. l'ensemble des régions L est défini par un triplet $(\Sigma, \mathcal{L}, \mathcal{V})$ où Σ est un alphabet fini, $\mathcal{L}$ est un langage sur Σ et $\mathcal{V} : \mathcal{L} \to \mathbb{N}$. L'ensemble L est défini comme l'ensemble des couples (w, ϕ) où $w \in \mathcal{L}$ et ϕ est une formule de Presburger sur $\mathbb{N}^{\mathcal{V}(m)}$;
2. La fonction de concrétisation $[\![\cdot]\!]$ vérifie $[\![(w, \phi)]\!] = \bigcup_{v\in[\![\phi]\!]} [\![(w, x = v)]\!]$;
3. la fonction de successeurs POST vérifie : pour tout $\varphi \in \Phi$ et pour tout $w \in \mathcal{L}$ il existe une formule de Presburger ζ sur $\mathbb{N}^{\mathcal{V}(w)+\mathcal{V}(w')}$ telle que pour tout $v \in N^{\mathcal{V}(w)}$ on a l'égalité $[\![\text{POST}(\varphi, (w, v))]\!] = [\![(w', \zeta(v, x'))]\!]$.

Notation. Dans la suite nous écrivons (w, v), où $v \in \mathbb{N}^{\mathcal{V}(m)}$ est une constante, au lieu de $(w, x = v)$.

De la condition 2 on déduit que les régions d'un P-cadre symbolique vérifient $[\![(w, Faux)]\!] = \emptyset$ et $[\![(w, \phi_1 \vee \phi_2)]\!] = [\![(w, \phi_1)]\!] \cup [\![(w, \phi_2)]\!]$. La condition 3 permet de déduire la propriété suivante.

Proposition 3.4.1. Pour tout $w \in \mathfrak{L}$ et $\varphi \in \Phi$, il existe $w' \in \mathfrak{L}$ et une formule de Presburger ζ sur $\mathbb{N}^{\mathcal{V}(w)+\mathcal{V}(w')}$ tels que pour toute formule de Presburger ϕ sur $\mathbb{N}^{\mathcal{V}(w)}$ on a $[\![\text{POST}(\varphi, (w, \phi))]\!] = \left[\!\left[(w', \exists x \in \mathbb{N}^{\mathcal{V}(w)}.\phi(x) \wedge \zeta(x, x'))\right]\!\right]$.

Démonstration. Soient $w \in \mathfrak{L}$ et $\varphi \in \Phi$. Comme SF est un cadre symbolique, l'opération POST vérifie : $[\![\text{POST}(\varphi, (w, \phi))]\!] = \varphi([\![(w, \phi)]\!])$, ce qui est égal à $\bigcup_{v \in [\![\phi]\!]} \varphi([\![(w, v)]\!])$ en utilisant la définition de $[\![\cdot]\!]$ et en distribuant φ. Or comme SF est un cadre symbolique il vient que $\varphi([\![(w, v)]\!]) = [\![\text{POST}(\varphi, (w, v))]\!]$. On en déduit donc que $[\![\text{POST}(\varphi, (w, \phi))]\!] = \bigcup_{v \in [\![\phi]\!]} [\![\text{POST}(\varphi, (w, v))]\!]$. En utilisant l'hypothèse 3 de la définition 3.4.1, on déduit l'existence d'un mot $w' \in \mathfrak{L}$ et d'une formule de Presburger ζ (ne dépendant que de w et φ) tels que $[\![\text{POST}(\varphi, (w, \phi))]\!] = \bigcup_{v \in [\![\phi]\!]} [\![(w', \zeta(v, x'))]\!]$. Or comme SF est un P-cadre symbolique, le terme droit de l'égalité vaut exactement $[\![(w', \exists x.\phi(x) \wedge \zeta(x, x'))]\!]$ (condition 2 de la définition 3.4.1). On a donc bien l'égalité cherchée, et w' et ζ respectent toutes les conditions souhaitées. □

3.4.2 P-Accélérations

Les P-accélérations sont des accélérations sur les P-cadres symboliques qui codent explicitement le *nombre d'itérations de cycle* en ajoutant une variable θ dans la partie contrainte d'une région (w, ϕ).

Définition 3.4.2 (P-accélération)**.** Soit I une interprétation et $P = (I, \mathcal{P}_f(L))$ un P-cadre symbolique. Une P-accélération pour P est une fonction d'accélération plate POST_STAR telle que

1. pour tout $w \in \mathfrak{L}$ et pour tout $\varphi \in \Phi$, il existe $w' \in \mathfrak{L}$ et ζ une formule de Presburger sur $\mathbb{N}^{\mathcal{V}(w)+\mathcal{V}(w')+1}$ tels que pour tout $v \in \mathbb{N}^{\mathcal{V}(w)}$, on a l'égalité $[\![\text{POST_STAR}(\varphi, (w, v))]\!] = [\![(w', \exists \theta \in \mathbb{N}.\zeta(v, \theta, x'))]\!]$;
2. soient $w \in \mathfrak{L}$ et $\varphi \in \Phi$, et $w' \in \mathfrak{L}$ et ζ définis à la condition 1. Alors $\varphi^i([\![(w, v)]\!]) = [\![(w', \exists \theta \in \mathbb{N}.\zeta(v, \theta, x') \wedge \theta = i)]\!]$.

La condition 1 de la définition 3.4.2 précise comment POST_STAR introduit la variable d'itération θ, et la condition 2 assure que θ compte bien le nombre d'itérations. On peut en déduire la propriété suivante.

Proposition 3.4.2. Pour tout $w \in \mathfrak{L}$ et pour tout $\varphi \in \Phi$, il existe w' et ζ une formule de Presburger sur $\mathbb{N}^{\mathcal{V}(w)+\mathcal{V}(w')+1}$ tels que pour toute formule de Presburger ϕ sur $\mathbb{N}^{\mathcal{V}(w)}$ on a :

$$[\![\text{POST_STAR}(\varphi, (w, \phi))]\!] = [\![(w', \exists \theta \in \mathbb{N}.\exists x.\phi(x) \wedge \zeta(x, \theta, x'))]\!].$$

Démonstration. Le raisonnement est analogue à la preuve de la proposition 3.4.1, en utilisant les propriétés de POST_STAR au lieu de celles de POST. □

3.4.3 Produits synchronisés

Nous définissons maintenant le produit synchronisé de P-cadres symboliques. Contrairement au produit cartésien où aucune information n'est partagée entre les différentes composantes, dans le produit synchronisé les composantes partagent la même contrainte ϕ.

Définition 3.4.3 (Produit synchronisé de P-cadres symboliques)**.** Soit P_1, P_2 deux P-cadres symboliques, notés $P_i = (I_i, \mathcal{P}_f(L_i))$, tels que les régions L_i sont définies par les triplets $(\Sigma_i, \mathcal{L}_i, \mathcal{V}_i)$. Le produit synchronisé P_1 et P_2, noté $P_1 \otimes P_2$, est le P-cadre symbolique P tel que les régions L de P sont définies par $(\Sigma_1 \times \Sigma_2, \mathcal{L}_1 \times \mathcal{L}_2, \mathcal{V}_1 \times \mathcal{V}_2)$. C'est à dire que les mots de L sont de la forme (w_1, w_2, ϕ) avec $w_1 \in \mathcal{L}_1, w_2 \in \mathcal{L}_2$ et ϕ une formule de Presburger sur $\mathbb{N}^{\mathcal{V}(w_1)+\mathcal{V}(w_2)}$. Les opérations $[\![\cdot]\!]\,, \sqcup, \sqsubseteq$ et POST sont définies par :

- La concrétisation de (w_1, w_2, Φ) vaut $[\![(w_1, w_2, \phi)]\!] = \bigcup_{(v_1, v_2 \in \phi)} [\![(w_1, v_1)]\!] \times [\![(w_2, v_2)]\!]$.
- L'union $\sqcup$ est l'union finie sur $\mathcal{P}_f(L)$;
- Le successeur POST est défini par comme suit. Pour w_1, w_2 et φ donnés, il existe w_1', w_2', ζ_1 et ζ_2 définissant complètement POST_1 et POST_2 (proposition 3.4.1). Alors $\text{POST}(\varphi, (w_1, w_2, \phi)) = (w_1', w_2', \exists(x_1, x_2).\phi(x_1, x_2) \wedge \zeta_1(x_1, x_1') \wedge \zeta_2(x_2, x_2'))$;
- L'inclusion est définie par $(w_1, w_2, \phi) \sqsubseteq (w_1', w_2', \phi')$ si $w_1 = w_1' \wedge w_2 = w_2' \wedge \phi \Rightarrow \phi'$.

Le produit synchronisé de P-cadres symboliques est bien un P-cadre symbolique.

Théorème 3.4.1. *soit $\mathcal{H} = I_1 \times \ldots \times I_n$ une interprétation faiblement hétérogène telle que pour tout i, il existe un P-cadre symbolique $P_i = (I_i, \mathcal{P}_f(L_i))$. Alors $P_1 \otimes \ldots \otimes P_n$ est un P-cadre symbolique sur $\mathcal{H}$.*

Démonstration. Nous raisonnons sur deux P-cadres symboliques P_1 et P_2. Le cas général s'obtient alors par récurrence sur le nombre de P-cadres symboliques. Nous voulons montrer que $P_1 \otimes P_2$, abrégé en $P_\otimes$, est un P-cadre symbolique. Pour cela nous devons montrer que $P_\otimes$ est un cadre symbolique et qu'il vérifie les conditions de la définition 3.4.1. On peut déjà remarquer que par définition $P_\otimes$ vérifie les conditions de la définition 3.4.1. Il ne reste donc plus qu'à montrer que $P_\otimes$ est un cadre symbolique. Nous raisonnons sur des mots $(w_1, w_2, \phi) \in L$. L'extension à des ensembles finis de mots est

directe. On doit montrer que pour tout $(w_1, w_2, \phi) \in L$ et pour tout $\varphi \in \Phi_1 \times \Phi_2$ on a : (1) $[\![(w_1, w_2, \phi) \sqcup (w'_1, w'_2, \phi')]\!] = [\![(w_1, w_2, \phi)]\!] \cup [\![(w'_1, w'_2, \phi')]\!]$; (2) si $(w_1, w_2, \phi) \sqsubseteq (w'_1, w'_2, \phi')$ alors $[\![(w_1, w_2, \phi)]\!] \subseteq [\![(w'_1, w'_2, \phi')]\!]$ (nous considérons des cadres affaiblis) ; et enfin (3) $[\![\text{POST}((\varphi_1, \varphi_2), (w_1, w_2, \phi))]\!] = \varphi_1 \times \varphi_2([\![(w_1, w_2, \phi)]\!])$. Les propriétés (1) et (2) découlent des définitions de $\sqcup, \sqsubseteq, [\![\cdot]\!]$ et de l'hypothèse que les P_i sont des cadres symboliques. Nous montrons la condition (3). Par définition de $[\![\cdot]\!]$ et en distribuant φ, il vient que $\varphi([\![(w_1, w_2, \phi)]\!]) = \bigcup_{(v_1,v_2)\in[\![\phi]\!]} \varphi_1([\![(w_1, v_1)]\!]) \times \varphi_2([\![(w_2, v_2)]\!])$. Or POST_1 et POST_2 sont des accélérations donc chaque $\varphi_i([\![(w_i, v_i)]\!])$ est égal à $[\![\text{POST}_i(\varphi_i, (w_i, v_i))]\!]$. En utilisant l'hypothèse 1 de la définition 3.4.2, on en déduit que $\varphi([\![(w_1, w_2, \phi)]\!]) = \bigcup_{(v_1,v_2)\in[\![\phi]\!]} [\![(w'_1, \zeta_1(v_1, x'_1))]\!] \times [\![(w'_2, \zeta_2(v_2, x'_2))]\!]$. En utilisant la condition sur la concrétisation d'un P-cadre symbolique le membre de droite de cette égalité est égal à $\bigcup_{(v_1,v_2)\in[\![\phi]\!]}(\bigcup_{v'_1\in[\![\zeta_1(v_1,x'_1)]\!]} [\![(w'_1, v'_1)]\!] \times \bigcup_{v'_2\in[\![\zeta_2(v_2,x'_2)]\!]} [\![(w'_2, v'_2)]\!])$, où encore $\bigcup_{(v'_1,v'_2)\in[\![\exists(v_1,v_2).\phi(v_1,v_2)\wedge\zeta_1(v_1,v'_1)\wedge\zeta_2(v_2,v'_2)]\!]} [\![(w'_1, v'_1)]\!] \times [\![(w'_2, v'_2)]\!]$. Or cette expression est égale par définition de POST et $[\![\cdot]\!]$ à $[\![\text{POST}(\varphi, (w_1, w_2, \phi))]\!]$. On a donc bien l'égalité cherchée. On déduit de ces propriétés que $P_\otimes$ est un P-cadre symbolique. □

Les ensembles de configurations représentables par produit synchronisé contiennent ceux représentables par produit cartésien.

Proposition 3.4.3. Soit deux P-cadres symboliques P_1 et P_2 sur les domaines D_1 et D_2. On note $P_\times$ le cadre symbolique défini par $P_\times = P_1 \times P_2$ et $P_\otimes$ le P-cadre symbolique défini par $P_\otimes = P_1 \otimes P_2$. Alors les ensembles $P_\times$-définissables sont inclus dans les ensembles $P_\otimes$-définissables.

Démonstration. Un ensemble $X \subseteq D_1 \times D_2$ est $P_\times$-définissable si il existe un couple $((w_1, \phi_1), (w_2, \phi_2))$ tel que $X = [\![((w_1, \phi_1), (w_2, \phi_2))]\!] = [\![(w_1, \phi_1)]\!] \times [\![(w_2, \phi_2)]\!]$. Alors X est $P_\otimes$-définissable car il peut s'écrire $X = [\![(w_1, w_2, \phi_1(x_1) \wedge \phi_2(x_2))]\!]$. □

Si on prend le P-cadre symbolique P des formules de Presburger sur 1 compteur[1], alors l'ensemble $\{x = y\}$ est $P_\otimes$-définissable mais pas $P_\times$-définissable. En fait souvent les ensembles $P_\times$-définissables sont strictement inclus dans les ensembles $P_\otimes$-définissables. Ceci nous permet de définir un produit synchronisé d'accélérations capturant exactement la composition d'accélérations.

Lors du calcul d'accélération, le nombre d'itérations est encodé explicitement par une variable θ_j. Le produit synchronisé d'accélérations consiste à faire les accélérations séparément sur chaque domaine puis à synchroniser les variables d'itérations θ_j entre elles.

[1]Il y a un seul w possible, toute l'information est codée dans la formule de Presburger.

Définition 3.4.4 (Produit synchronisé de P-accélérations)**.** Soit P_1 et P_2 deux P-cadres symboliques admettant chacun une P-accélération POST_STAR_i définie par $p_i : (\varphi_i \in \Phi_i, w_i \in \mathcal{L}_i) \mapsto (w_i' \in \mathcal{L}, \zeta_i)$. Alors le produit synchronisé de POST_STAR_1 et POST_STAR_2, noté $\text{POST_STAR}_1 \otimes \text{POST_STAR}_2$, est défini par : $\text{POST_STAR}_1 \otimes \text{POST_STAR}_2(w_1, w_2, \phi) =$
$(w_1', w_2', \exists(x_1, \theta, x_2).\phi(x_1, x_2) \wedge \zeta_1(x_1, \theta, x_1') \wedge \zeta_2(x_2, \theta, x_2'))$.

Nous pouvons enfin montrer notre résultat principal : le produit synchronisé de P-accélérations est une P-accélération.

Théorème 3.4.2. *Soit $\mathcal{H} = I_1 \times \ldots \times I_n$ une interprétation faiblement hétérogène telle que pour tout $i \leq n$, l'interprétation I_i est muni d'un P-cadre symbolique P_i admettant une P-accélération* POST_STAR_i*. Alors le P-cadre symbolique $P_1 \otimes \ldots \otimes P_n$ admet la P-accélération* $\text{POST_STAR}_1 \otimes \ldots \otimes \text{POST_STAR}_n$.

Démonstration. Nous raisonnons pour deux interprétations et deux P-cadres symboliques. Le résultat général s'obtient par récurrence. Nous voulons montrer que le produit $\text{POST_STAR}_1 \otimes \text{POST_STAR}_2$, noté $\text{POST_STAR}_\otimes$, est une P-accélération. Pour cela nous devons montrer que c'est une accélération plate et que les conditions de la définition 3.4.2 sont vérifiée. Ce deuxième point est évident. Nous montrons le premier point. Nous raisonnons sur les mots (w_1, w_2, ϕ), l'extension aux ensembles finis de mots est directe. Soit (w_1, w_2, ϕ) et $\varphi \in \Phi_1 \times \Phi_2$, nous voulons montrer que $[\![\text{POST_STAR}(\varphi, (w_1, w_2, \phi))]\!] = \varphi^*([\![(w_1, w_2, \phi)]\!])$. La preuve suit les mêmes idées que celle du théorème 3.4.1 (pour la fonction POST) excepté à un endroit où la condition 2 de la définition 3.4.2 sera utilisée. En utilisant les définitions de φ^* et de $[\![\cdot]\!]$, on montre que $\varphi^*([\![(w_1, w_2, \phi)]\!]) = \bigcup_{i \in \mathbb{N}, (v_1, v_2) \in [\![\phi]\!]} \varphi_1^i([\![(w_1, v_1)]\!]) \times \varphi_2^i([\![(w_2, v_2)]\!])$. En utilisant la condition 2 de la définition 3.4.2 (P-accélération), on déduit que chaque $\varphi_j^i([\![(w_j, v_j)]\!])$ vaut $\left[\!\!\left[(w_j', \exists \theta_j \in \mathbb{N}.\zeta_j(v_j, \theta_j, x_j') \wedge \theta_j = i)\right]\!\!\right]$. En utilisant la définition des concrétisations on déduit que
$\varphi^*([\![(w_1, w_2, \phi)]\!]) = \bigcup_{i \in \mathbb{N}, (v_1, v_2) \in [\![\phi]\!]} [\![(w_1, \zeta_2(v_2, i, x_2'))]\!] \times [\![(w_2, \zeta_2(v_2, i, x_2'))]\!]$. Si on renomme i en θ le terme de droite se réécrit en $E =$
$\bigcup_{(v_1', v_2') \in [\![\exists(x_1, x_2, \theta).\phi(x_1, x_2) \wedge \zeta_1(v_1, \theta, v_1') \wedge \zeta_2(v_2, \theta, v_2')]\!]} [\![(w_1, \zeta_2(v_2, i, x_2'))]\!] \times [\![(w_2, \zeta_2(v_2, i, x_2'))]\!]$.
En utilisant la définition de $\text{POST_STAR}_\otimes$ et $[\![\cdot]\!]$ on remarque que l'ensemble E est égal à $\left[\!\!\left[\text{POST_STAR}_\otimes(\varphi, (w_1, w_2, \phi))\right]\!\!\right]$. On a donc bien montré l'égalité cherchée $[\![\text{POST_STAR}(\varphi, (w_1, w_2, \phi))]\!] = \varphi^*([\![(w_1, w_2, \phi)]\!])$. D'où la conclusion. □

Selon la proposition 3.4.2 une P-accélération est entièrement caractérisée par la fonction $p : (\varphi, w) \mapsto (w', \zeta)$. Nous dirons qu'une P-accélération est définie de manière effective si la fonction p est donnée et récursive. On déduit de la démonstration du théorème 3.4.2 que si les accélérations sont définies de manière effective, alors le produit synchronisé est calculable de manière effective.

Théorème 3.4.3. *Il existe une fonction récursive qui prend en entrée des P-accélérations* $\text{POST_STAR}_1, \ldots, \text{POST_STAR}_n$ *définies de manières effectives et qui retourne la P-accélération* $\text{POST_STAR}_1 \otimes \ldots \otimes \text{POST_STAR}_n$.

Démonstration. Ce résultat découle directement de la démonstration du théorème 3.4.2. □

3.4.4 Extensions

Une extension possible de nos résultats est de permettre aux opérations $\text{POST}(\varphi, (w, \phi))$ et $\text{POST_STAR}(\varphi, (w, \phi))$ de retourner non pas un seul (w', ϕ') mais un nombre fini de (w'_k, ϕ'_k). Par exemple la condition 1 de la définition 3.4.2 (P-accélération) s'écrit alors : pour tout $w \in \mathcal{L}$ et pour tout $\varphi \in \Phi$ il existe $w'_1, \ldots, w'_k \in \mathcal{L}$ et $\zeta_1, \ldots, \zeta_k$ des formules de Presburger tels que pour tout $v \in \mathbb{N}^{\mathcal{V}(w)}$, on a l'égalité $[\![\text{POST_STAR}(\varphi, (w, v))]\!] = \bigcup_{j \leq k} [\![(w'_j, \exists \theta \in \mathbb{N}. \zeta_j(v, \theta, x'))]\!]$. *Nos résultats sont toujours valides sur cette extension, les preuves sont des adaptations directes en distribuant les calculs sur les différents* w'_i .

Une autre extension est de considérer d'autres logiques que la logique de Presburger. Nos constructions (et les preuves) nécessitent que les formules logiques soient closes par intersection et quantification existentielle, que l'on puisse exprimer l'égalité entre une variable et une constante et enfin que la validité de l'implication soit décidable. On peut donc déjà facilement étendre nos constructions à la logique des automates binaires.

3.4.5 P-cadres symboliques existants

Certains cadres symboliques déjà évoqués au chapitre 2 sont des P-cadres symboliques. Ainsi les automates binaires (compteurs), les rva (horloges), les cqdd (files ou piles) sont des P-cadres symboliques. Les états mémoires symboliques développés au chapitre 6 pour les systèmes à pointeurs sont aussi un P-cadre symbolique.

Par exemple pour les cqdd, les mots w sont des expressions régulières semi-linéaires sur l'alphabet des messages Σ_m. Ce sont donc des mots sur l'alphabet $\Sigma = \Sigma_m \cup \{+\} \cup \{^*\} \cup \{\varepsilon\} \cup \{\emptyset\}$. Les formules associent à chaque * une variable comptant le nombre d'itérations de *. Pour les automates binaires, il n'y a qu'un unique mot w, toutes les opérations se font en modifiant la formule ϕ. En contrepartie les bdd, les qdd, les slre et les sre ne sont pas des

P-cadres symboliques car il n'y a pas de représentation explicite du comptage. Les cpdbm sont presque un P-cadre symbolique, sauf que les contraintes ne sont pas restreintes à la logique de Presburger.

Les accélérations des automates binaires et des cqdd (files ou piles [Bou01]) sont des P-accélérations. Les accélérations sur les rva [BHJ03] et sur les cpdbm [AAB00] utilisent bien une variable d'itération, mais elles ne suivent pas exactement le cadre d'accélération défini au chapitre 2. L'algorithme suggéré au théorème 3.4.2 s'applique encore, mais le résultat n'est plus une accélération plate car on ne sait plus calculer l'accélération de n'importe quel circuit (cela dépend dans ce cas de l'ensemble de configurations).

De ces diverses résultats on peut déduire le théorème suivant.

Théorème 3.4.4. *Il existe un P-cadre symbolique et une P-accélération plate pour les systèmes faiblement hétérogène manipulant des compteurs affines, des files et des piles.*

3.5 Conclusion

Nous avons étudié dans ce chapitre la composition automatique de cadres symboliques et d'algorithmes d'accélération plate. Nous avons tout d'abord introduit la notion clé de système faiblement hétérogène, dans lesquels les domaines *et* les relations manipulées sont des produits cartésiens. Nous avons montré que pour un système faiblement hétérogène, le produit cartésien des cadres symboliques est un cadre symbolique. Malheureusement le produit cartésien des accélérations plates n'est pas une accélération plate.

Nous avons défini la classes des P-cadres symboliques et la classe des P-accélérations. Ces deux classes peuvent être dotées d'un produit synchronisé, avec le résultat suivant : le produit synchronisé de P-accélérations est toujours une P-accélération. Des représentations symboliques comme les automates binaires et les cqdd s'avèrent être des P-cadres symboliques et les accélérations plates correspondantes sont des P-accélérations.

Les P-cadres symboliques sont des représentations dont les opérations ont une grande complexité au pire puisqu'elles obligent à manipuler des formules de Presburger. Cependant elles ont de bonnes propriétés de composition et elles sont particulièrement adaptées à la vérification de propriétés quantitatives. De plus, dans le cas des automates binaires la complexité au pire est

rarement atteinte.

Les résultats ci-dessus sont effectifs, c'est à dire que modulo quelques hypothèses sur les algorithmes de P-accélération on sait calculer effectivement l'accélération synchronisée. On peut donc envisager d'inclure cette construction dans un outil de vérification quantitative de systèmes faiblement hétérogènes, pour construire à la demande des P-cadres symboliques et des algorithmes de P-accélération à partir de bibliothèques pour chaque type de donnée. À l'heure actuelle un tel outil permettrait de manipuler des systèmes à compteurs, files et piles.

Deuxième partie

Vérification de systèmes à compteurs

Chapitre 4

Accélération plate pour systèmes à compteurs

4.1 Introduction

4.1.1 Contexte

Nous considérons dans ce chapitre le cas particulier des systèmes à compteurs. Un *système à compteurs* est une structure de contrôle finie étendue avec m variables entières dont les valeurs sont modifiées par des actions (`b := b+1` par exemple). La figure 4.1 présente un système à compteurs.

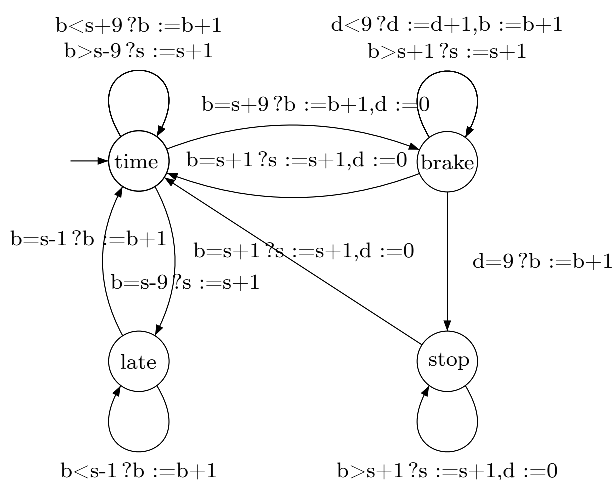


FIG. 4.1: Un système à compteurs.

Ces systèmes sont intéressants pour au moins deux raisons. D'un point

de vue pratique ils permettent de modéliser de nombreux systèmes : protocoles de communication, abstractions de programmes multithreads [Del]. D'un point de vue plus théorique de nombreuses familles de systèmes bien étudiées dans la littérature sont des sous-classes de la famille des systèmes à compteurs, par exemple les VASS avec remise à zéro, arc transfert ou test à zéro [DFS98, FS00], les protocoles broadcast [EN98, EFM99, Del00a, Del00b] ou encore les systèmes à compteurs reversal bornés [ISD+02].

Les systèmes à compteurs sont très expressifs : il suffit de deux compteurs testables à zéro pour simuler une machine de Turing. Bien sûr certaines des sous classes citées précédement ont de meilleures propriétés de décision, puisque l'accessibilité y est décidable. C'est le cas par exemple des VASS [May81] et des systèmes à compteurs reversal bornés [Iba78, ISD+00]. Cependant, comme les algorithmes d'accessibilité ne sont pas génériques, peu ont été implantés.

4.1.2 Accélération plate de systèmes à compteurs

De nombreux travaux considèrent l'accélération des systèmes à compteurs. On peut citer par exemple [Rev90, BW94, FO97a, FO97b, Boi98, BF99, PS00, AAB00, FL02, Boi03]. Ces approchent fournissent des procédures génériques pour le calcul de l'ensemble d'accessibilité de larges sous-classes de systèmes à compteurs. Nous présentons ici les points clés pour analyser les systèmes à compteurs par accélération plate, à savoir le cadre symbolique et l'algorithme d'accélération.

Les systèmes à compteurs affines. Les systèmes à compteurs n'admettent pas d'accélération plate, car le problème de l'accessibilité dans ces systèmes n'est pas décidable (lemme 4.2.1). Nous nous intéressons à la sous-classe des systèmes à compteurs affines où les transitions sont étiquetées par des fonctions affines gardées par une formule de Presburger.

Le cadre symbolique. Les ensembles Presburger-définissables peuvent être représentés symboliquement par des automates [BC96, WB00, Ler03a]. Cette représentation définit un cadre symbolique pour les systèmes à compteurs comme décrit au chapitre précédent.

L'accélération plate. Boigelot et Wolper dans [BW94, Boi98, Boi03] et Finkel et Leroux dans [FL02] ont montré que sous certaines hypothèses algébriques (finitude du monoïde engendré par les matrices des fonctions affines), l'accélération plate est admise. De plus de tels systèmes sont courants : les

VASS et la plupart de leurs extensions [DFS98, FS00], les protocoles broadcast d'Emerson et Namjoshi [EN98, EFM99], et les systèmes à compteurs reversal bornés d'Ibarra [ISD+02] sont des systèmes affines à monoïde fini.

Il existe de nombreux résultats d'accélération plate pour des variantes de systèmes à compteurs affines. [Rev90, FO97a, FO97b, BF99] accélèrent automatiquement des cycles non élémentaires. [PS00] considère le problème de l'accélération en utilisant des formules dans une variante de *WS1S* pour la représentation symbolique des configurations. Parmi les nombreux travaux sur le sujet nous pouvons également citer [BF00] et [BGP99], qui utilisent tous deux l'outil MONA [Mon]. La procédure implantée dans l'outil TREX [AAB00] calcule des itérations de séquences de transitions sur des systèmes à compteurs dont les actions sont des affectations simples $x_i := x_j + c$ gardées par des des contraintes diagonales $x_i - x_j\{\leq, \geq\}c$, où x_i, x_j sont des variables et $c \in \mathbb{N}^m$. Cette procédure n'est pas une accélération au sens strict du chapitre 2. D'un côté, la procédure n'est pas récursive, de l'autre elle permet dans certains cas de calculer des itérations de boucles imbriquées.

4.1.3 Implantation efficace de l'accélération plate

Nous nous concentrons sur une implantation efficace de l'accélération plate pour les systèmes à compteurs. Pour cela nous étudions deux problèmes clés : l'algorithme d'accélération et les techniques de réduction.

Complexité de l'accélération. La question de la complexité de l'accélération n'a guère été étudiée. Dans [BFL04], nous montrons que l'accélération plate définie dans [FL02] peut être bornée en temps et en espace par 3-EXP dans la taille du domaine. Les autres facteurs sont le nombre de variables et la plus grande constante intervenant dans la définition de f. Nous nous intéressons à définir des algorithmes d'accélération plate plus efficaces pour des systèmes à compteurs utilisant des translations convexes et des translations positives.

Les réductions. Dans [FL02], Finkel et Leroux présentent une procédure diminuant exponentiellement le nombre de cycles de longueur k à considérer pour calculer l'ensemble d'accessibilité. Nous montrons que cette technique est bien une k-réduction au sens du chapitre 2. Nous montrons que dans certains cas cette technique permet de s'affranchir du cadre strict de l'accélération plate, c'est-à-dire qu'elle permet de calculer l'ensemble d'accessibilité

de systèmes non applatissables. Enfin nous la comparons aux réductions génériques définies au chapitre 2.

4.2 Systèmes à compteurs affines

Nous définissons dans cette section les systèmes à compteurs affines. Nous discutons la pertinence de cette classe vis-à-vis de l'accélération plate. Enfin nous introduisons la notion de monoïde d'un système.

4.2.1 Systèmes à compteurs

Nous définissons les systèmes à compteurs selon le schéma introduit au chapitre 2. Pour cela nous devons définir l'ensemble des formules Φ étiquettant les transitions d'un système à compteurs, le domaine D des données manipulées et une sémantique $[\![\cdot]\!] : \Phi \to 2^{D \times D}$. Les systèmes à m compteurs s'obtiennent à partir du cadre des systèmes comme suit : les actions Φ_m sont les formules de Presburger sur $\mathbb{Z}^{2m}$, le domaine D est l'ensemble des vecteurs d'entiers relatifs $\mathbb{Z}^m$, la concrétisation $[\![\cdot]\!]$ associe à $\varphi \in \Phi_m$ la relation Presburger-définissable classiquement définie.

Définition 4.2.1 (Système à m compteurs). Un *système à m compteurs S* est un système $S = (Q, \Phi_m, T, \mathbb{Z}^m, [\![\cdot]\!])$ où Q est un ensemble fini de locations, Φ_m est l'ensemble des formules de Presburger sur $2m$ variables libres, $T \subseteq Q \times \Phi_m \times Q$ est un ensemble fini non vide de *transitions* et $[\![\cdot]\!]$ associe à $\varphi \in \Phi$ la relation Presburger-définissable.

Notation. Dans la suite nous notons le système à m compteurs $S = (Q, \Phi_m, T, \mathbb{Z}^m, [\![\cdot]\!])$ par le triplet $S = (Q, T, m)$ et nous parlons de systèmes à compteurs.

Remarque 4.2.1. L'étude d'un système à $|Q|$ locations et m compteurs $S = (Q, T, m)$ peut toujours se ramener à l'étude d'un système $S' = (\{q'\}, T', m+1)$ à une seule location et $m + 1$ compteurs. Il suffit pour cela de coder la structure de contrôle dans une variable supplémentaire x_Q et de rajouter les opérations nécessaires à chaque transition. Ainsi, si l'on identifie une location avec sa valuation, T' contient exactement les transitions t'_i de la forme $(q', l(t_i) \wedge x_Q = \alpha(t_i) \wedge x'_Q = \beta(t_i), q')$ où $t_i \in T$ et x' désigne la valeur de la variable x après la transition t_i.

Les systèmes à compteurs sont très expressifs puisqu'ils peuvent simuler une machine de Turing. Nous montrons qu'ils n'admettent pas d'accélération plate.

Lemme 4.2.1. Les systèmes à compteurs n'admettent pas d'accélération plate.

Démonstration. Comme la logique de Presburger est close par disjonction, on peut ramener tout système à compteurs $S = (Q, T, m)$ à un système à compteurs $S' = (\{q'\}, \{t'\}, m+1)$ tel que t' boucle sur q' et $l(t') = \bigcup_{t \in T} l(t)$. On en déduit que si les systèmes à compteurs admettaient une accélération plate, alors on pourrait calculer leur ensemble d'accessibilité. Or le problème de l'accessibilité dans les systèmes à compteurs étant décidable, on en déduit que les systèmes à compteurs n'admettent pas d'accélération plate. □

4.2.2 Systèmes à compteurs affines

Nous considérons une restriction des systèmes à compteurs admettant une accélération plate. Nous définissons tout d'abord les fonctions Presburger-affines, puis les systèmes à compteurs affines.

Définition 4.2.2 (Fonction Presburger-affine [FL02]). Une *fonction Presburger-affine* f sur m compteurs est un triplet $f = (M, v, G)$ tel que $\forall x \in G, f(x) = M.x + v$, où $M \in \mathcal{M}_m(\mathbb{Z})$ est une matrice carrée sur les entiers relatifs, $v \in \mathbb{Z}^m$ est un vecteur sur les entiers relatifs et $G \subseteq \mathbb{Z}^m$ est un ensemble Presburger-définissable appelé *la garde* de f.

Dans la suite une fonction Presburger-affine est donnée syntaxiquement par la matrice carrée M, le vecteur constant v et un uba encodant l'ensemble G.

Définition 4.2.3 (Système à compteurs affine). Un système à compteurs $S = (Q, T, m)$ est dit affine si pour tout $t = (q, l(t), q') \in T$, il existe une fonction Presburger-affine $f_t = (G_t, M_t, v_t)$ telle que $[\![l(t)]\!] = f_t$.

Notation. Dans la suite, les systèmes à compteurs affines seront notés (Q, T, m) *ou* $(Q, T, (G_t, M_t, v_t)_{t \in T})$.

4.2.3 Monoïde d'un système à compteurs affine

Une notion centrale dans l'accélération plate des systèmes affines est *la finitude du monoïde du système*. Nous définissons tout d'abord le *monoïde d'une fonction Presburger-affine* $f = (M, v, G)$ comme le monoïde multiplicatif M^* engendré par la matrice M, soit $M^* = \{Id, M, M^2, \ldots\}$. Nous définissons maintenant le monoïde d'un système à compteurs affine. Pour cela nous notons $\mathcal{M}$ l'ensemble $\mathcal{M} = \{M_t ; t \in T\}$.

Définition 4.2.4. Le monoïde $\mathcal{M}^*$ d'un système à compteurs affine S est le monoïde multiplicatif engendré par l'ensemble des matrices $\mathcal{M}$, soit :
$\mathcal{M}^* = \bigcup_{n \geq 0} \bigcup_{M_1,\ldots,M_n \in \mathcal{M}} M_1 \ldots M_n$.

Remarque 4.2.2. Savoir si le monoïde d'une fonction Presburger-affine est fini est décidable en temps polynomial [Boi98]. Savoir si le monoïde d'un système à compteurs affine est fini est également décidable, en temps exponentiel [MS77].

Les machines de Minsky, les réseaux de Petri étendus par des remises à zéro, tests à zéro ou transferts [DFS98, FS00] et les protocoles broadcast [EN98, EFM99, Del00a, Del00b] sont des systèmes affines à monoïde fini.

4.2.4 Sur les systèmes à compteurs affines

Les systèmes à compteurs affines à monoïde fini satisfont deux propriétés essentielles pour notre approche. Tout d'abord, ils sont suffisamment expressifs pour englober la plupart des sous-classes connues de systèmes à compteurs. Aussi toutes les techniques décrites dans ce chapitre peuvent s'appliquer à ces familles de systèmes, ce qui permet d'unifier les procédures de vérification.

D'autre part les systèmes à compteurs affines à monoïde fini admettent une accélération plate (voir la section 4.4). Du point de vue de l'accélération plate, les systèmes affines ont trois avantages spécifiques, par exemple par rapport aux machines de Minsky :

Les transitions sont stables par composition ce qui simplifie les procédure d'accélération plate, puisqu'une séquence de transitions se comporte comme une transition.

Les transitions sont plus expressives que celles des machines de Minsky ce qui réduit le nombre de transitions des systèmes et donc de boucles imbriquées, qui sont le principal obstacle à la procédure d'applatissement du chapitre 2.

Les gardes des transitions sont closes par union et ce résultat est central pour la technique de réduction décrite dans la section 4.5. Cette technique de réduction est elle-même cruciale pour la vérification pratique de systèmes (voir les expérimentations du chapitre 5).

4.3 Cadre symbolique : automates binaires

Nous décrivons dans cette section un cadre symbolique pour la vérification de systèmes à compteurs. Rappelons que ce cadre se définit principalement

par un ensemble de régions (interprétées comme des ensembles infinis de configurations) clos par union et calcul de successeurs tel que l'inclusion est décidable. Les ensembles Presburger-définissables (voir la définition à la section 1.2.2), ont toutes les propriétés requises. Nous les encodons au moyen d'automates.

4.3.1 Automates binaires

Soit $r \geq 2$ un entier appelé *base de décomposition*. On note Σ_r l'ensemble $\Sigma_r = \{0, \ldots, r-1\}$. Un entier peut être vu comme le mot de sa décomposition en base r, soit un mot sur l'alphabet Σ_r. Aussi, un ensemble d'entiers en base r est un langage sur Σ_r. Si ce langage est régulier, il peut être reconnu par un automate fini. En fait le résultat s'étend à $\mathbb{Z}$ en utilisant un codage en complément à r et aux tuples d'entiers sur $\mathbb{Z}^m$, m fixé. Les mots de l'automate sont alors soit des tuples sur Σ_r^m comme dans [BC96], soit des entrelacements de mots sur Σ_r [Boi98, WB00, Ler03a] : on lit une lettre de la première composante, puis une lettre de la seconde, etc. On pourra consulter [BHMV94] pour un tour d'horizon des différents résultats concernant le sujet.

La figure 4.2 montre un automate reconnaissant l'ensemble $\{(x, y, z)|x + y = z\}$. La base r vaut 2, l'automate lit des vecteurs de lettres et la lecture se fait bit de poids faible en premier.

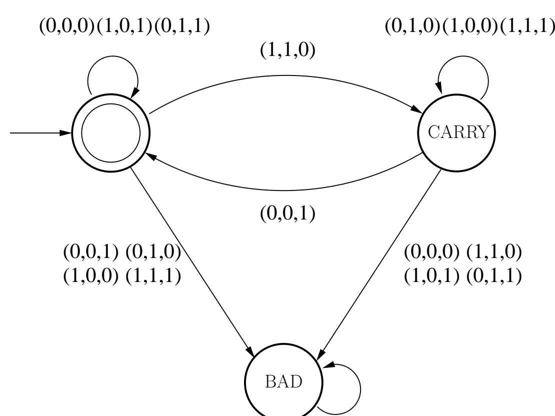


FIG. 4.2: Un automate représentant $\{(x, y, z)|x + y = z\}$.

Un vecteur $x \in \mathbb{Z}^m$ a une infinité de codages sur Σ_r^*, pusqu'on peut rajouter un nombre arbitraire de 0 à la décomposition de x. Nous dirons

qu'un langage $\mathcal{L} \subseteq \Sigma_r^*$ est non ambigu si : pour $x \in \mathbb{Z}^m$, soit aucun des codages de x n'est dans $\mathcal{L}$, soit tous ses codages sont dans $\mathcal{L}$.

Définition 4.3.1. Les deux classes principales d'automates pour reconnaître les ensembles d'entiers sont les suivantes :
- les automates binaires non ambigus, ou uba [Ler03a, Ler03b], sont des automates déterministes et complets sur l'alphabet $\{0, \ldots, r-1\}$ tel que le langage reconnu soit non ambigu ;
- les diagrammes de décision numérique, ou ndd [Boi98, WB00], sont des uba tels que la longueur de tout mot accepté par l'automate est divisible par m.

Les uba et les ndd représentent les mêmes ensembles de vecteurs d'entiers et de plus leur taille est similaire à un facteur linéaire près [Ler03b]. Plus précisément, pour un ensemble $X \subseteq \mathbb{Z}^m$, l'uba minimal U_X et le ndd minimal N_X représentant X (s'ils existent) vérifient $\frac{1}{m}.|N_X| \leq |U_X| \leq |N_X|$.

La suite de ce chapitre peut s'appliquer aussi bien à des uba qu'à des ndd. Nous emploierons le terme générique d'automate binaire pour désigner ces deux approches. Nous considérons toujours des automates minimaux en base de décomposition $r = 2$, les variables sont lues de droite à gauche (bit de poids faible en premier). Pour les résultats de complexité, nous utilisons les uba.

4.3.2 Cadre symbolique

Les automates binaires sont un cadre symbolique pour les systèmes à compteurs. Les opérations ensemblistes requises s'expriment en terme d'opérations classiques sur les automates.

Théorème 4.3.1 ([Boi98, WB00, Ler03a])**.** *Les automates binaires sont des cadres symboliques pour les systèmes à compteurs.*

Démonstration. L'union et l'inclusion se définissent par des unions et inclusion sur les langages des automates. Les opérations de successeurs et prédécesseurs s'expriment en introduisant des variables de prédecesseurs et en les quantifiant. Par exemple pour une relation Presburger-définissable R et un ensemble Presburger-définissable X, l'image de X par R est définie par la formule de Presburger $\exists X_0, R(X_0, X)$. La quantification est réalisée par une projection d'arcs de l'automate. □

Les automates binaires apportent de plus d'autres opérations intéressantes : intersection, complément, produit cartésien, test à vide et calcul

de prédécesseurs par exemple. De plus, la représentation par automate est unique à un isomorphisme près. Enfin il est possible d'extraire en temps polynomial la formule de Presburger associée à l'automate binaire [Ler05].

4.3.3 Complexité de la construction

Nous donnons ici quelques résultats de complexité sur les automates binaires. Ces résultats sont résumés dans [BB02]). Soit $|\mathcal{A}|$ la taille (en nombre de nœuds) d'un automate $\mathcal{A}$, φ une formule de Presburger et $\mathcal{A}(\varphi)$ son automate binaire associé.

- l'automate binaire $\mathcal{A}(X)$ où $X = \{x \in \mathbb{Z}^m;\ \sum_{i=1}^m \alpha_i.x_i \# c\}$ avec $\alpha_i, c \in \mathbb{Z}$ et $\# \in \{\leq, \geq, =\}$ peut être calculé en temps et espace bornés par $m.(\sum_{i=1}^m |\alpha_i| + |c|) + 1$.
- l'automate binaire $\mathcal{A}(X)$ où $X = \{x \in \mathbb{Z}^m;\ \sum_{i=1}^m \alpha_i.x_i = c[k]\}$ avec $\alpha_i, c, k \in \mathbb{Z}$ et $c[k]$ désigne *c modulo k*, peut être calculé en temps et en espace bornés par $2.m.|k| + 1$.
- les automates binaires $\mathcal{A}(X \cap Y)$ et $\mathcal{A}(X \cup Y)$ peuvent être calculés en temps et en espace bornés par $|\mathcal{A}(X)|.|\mathcal{A}(Y)|$.
- l'automate binaire $\mathcal{A}(X \times Y)$ peut être calculé en temps et en espace bornés par $4.|\mathcal{A}(X)|.|\mathcal{A}(Y)|$.
- les automates binaires $\mathcal{A}(X \times \mathbb{Z}^m)$ et $\mathcal{A}(\mathbb{Z}^m \times X)$ peuvent être calculés en temps et en espace bornés par $2.|\mathcal{A}(X)|$ si X a m variables libres.
- l'automate binaire $\mathcal{A}(\Pi(X))$ où Π est une fonction de projection se calcule en temps et espace bornés par $m.2^{|\mathcal{A}(X)|}$.
- l'automate binaire $\mathcal{A}(\mathbb{Z}^m \backslash X)$ se calcule en temps et espace bornés par $|\mathcal{A}(X)|$.

Récemment il a été démontré dans [Kla04] que la construction de l'automate binaire à partir d'une formule de Presburger quelconque est 3-EXPSPACE complet.

4.3.4 Ensembles reconnaissables par automates binaires

On peut se poser la question de l'expressivité des ensembles reconnus par automates binaires. On sait depuis longtemps que les ensembles Presburger-définissables sont reconnus par des automates binaires. Dans [BHMV94], Bruyère, Hansel, Michaux et Villemaire présentent un tour d'horizon des résultats sur le sujet. Ce résultat s'applique naturellement aux relations.

Théorème 4.3.2. *Les ensembles Presburger-définissables sont reconnus par les automates binaires.*

Dans [BHMV94] il est prouvé que les ensembles d'entiers reconnus par les automates binaires sont exactement les ensembles de solutions de l'arithmétique de Presburger augmentée d'un prédicat permettant de tester la valeur du i-ème bit d'une variable x. Dans la suite nous appelerons logique des automates binaires la logique ainsi définie. Cette logique permet par exemple d'exprimer que la valeur d'une variable est une puissance de la base.

4.4 Accélération plate

Nous nous concentrons dans cette section sur l'efficacité des algorithmes d'accélération plate. Tout d'abord nous rappelons les résultats d'accélération des systèmes à compteurs affines à monoïde fini. Ensuite, nous identifions les translations convexes et les translations positives pour lesquelles des algorithmes d'accélération plus efficaces sont proposés.

4.4.1 Accélération plate pour fonctions Presburger-affines

Nous désignons par $||v||_\infty$ la norme infinie du vecteur v (c'est-à-dire la plus grande valeur absolue des éléments de v). Pour une fonction Presburger-affine $f = (M, v, G)$, nous notons $\bar{f}$ la fonction $\bar{f} = (M, v, \mathbb{Z}^m)$.

Théorème 4.4.1 ([FL02]). *Les systèmes à compteurs affines à monoïde fini admettent une accélération plate.*

Dans [Boi98] le résultat est démontré pour des gardes et des ensembles de configurations convexes. Les conditions sur la fonction sont des conditions sur ses valeurs propres. [FL02] introduit la notion de monoïde fini et étend le résultat aux ensembles et gardes Presburger-définissables. En fait le résultat va plus loin puisque pour toute fonction Presburger-affine f, la relation f^* est Presburger-définissable.

Théorème 4.4.2 ([FL02]). *Soit une fonction Presburger-affine $f = (M, v, G)$ à monoïde fini alors la relation f^* est effectivement Presburger-définissable.*

Démonstration. La preuve est tirée de [FL02]. De manière générale f^* peut s'exprimer sous la forme $f^* = \{(x, x')|x \in G \wedge (\exists k \geq 0; x' = \bar{f}^k(x) \wedge (\forall i; 0 \leq i < k, \bar{f}^i(x) \in G))\}$. Intuitivement on calcule le n-ième successeur sans prendre en compte la garde, puis on vérifie que ses prédécesseurs sont bien dans la garde. Ensuite on quantifie sur n. Le problème est le calcul des $\bar{f}^i$.

L'hypothèse de finitude du monoïde M^* permet de déduire l'existence de deux entiers a et b vérifiant $M^{a+b} = M^a$. Dès lors, on peut montrer que $\forall n \in \mathbb{N}, \forall x \in \mathbb{Z}^m, \bar{f}^{a+n.b}(x) = \bar{f}^a(x)+n.M^a.\bar{f}^b(0)$ On peut définir alors l'ensemble $F = \{(i, x, x') \in \mathbb{N} \times \mathbb{Z}^m \times \mathbb{Z}^m, x' = \bar{f}^i(x)\}$ par

$$\bigvee_{r=0}^{a-1} [(x' = M^r.x + v_r) \wedge (i = r)] \bigvee_{r=0}^{b-1} \left[\begin{array}{l} (n \geq 0) \\ \wedge(x' = M^{a+r}.x + v_{a+r} + n.M^{a+b+r}.v_b) \\ \wedge(i = a + r + n.b) \end{array} \right]$$

La relation $f^* = \{(x, x'), \exists i \geq 0, x' = f^i(x)\}$ se définit aisément à partir de F par

$$\{(x, x'), \exists i \geq 0[(i, x, x') \in F \wedge (\forall k(0 \leq k < i), \exists x'' \in G, (k, x, x'') \in F)]\}$$

Cette formule est bien une formule de Presburger, d'où la conclusion. □

L'accélération plate est ici définie en terme de formules de Presburger, indépendamment de la représentation (formules, automates, ou autre).

La question de la complexité de l'accélération n'a guère été étudiée. Les différents travaux de [BW94, Boi98, Boi03, FL02] observent tous que même si la complexité théorique de l'accélération plate est certainement au moins exponentielle, la construction pratique fonctionne bien et le pire des cas est rarement atteint. Nous proposons une borne supérieure du calcul d'accélération plate dans le cas d'un codage par automates binaires [BFL04].

Proposition 4.4.1. Soit $f = (M, v, G)$ une fonction Presburger-affine à monoïde fini. Un uba représentant la relation f^* peut être calculé en temps et en espace bornés par 3-EXP en $|\mathcal{A}(G)|$, $||v||_\infty$ et $||M||_\infty$ et 5-EXP en m.

Démonstration. D'abord nous bornons les matrices du monoïde M^*. Comme le cardinal de M^* est borné par $(4.m)^{2.m}$ ([Boi98], page 222), pour toute $M' \in M^*$ il existe $k \leq (4.m)^{2.m}$ tel que $M' = M^k$. Donc $||M'||_\infty \leq (m. ||M||_\infty)^{(4.m)^{2.m}}$. Pour tout $k \geq 0$, nous définissons le vecteur v_k par $v_k = f^k(0)$. Une induction sur $k \geq 0$ montre que v_k est la somme des k vectors $M_1.v$, ..., $M_k.v$ où $M_i \in M^*$. Donc pour tout $M' \in M^*$, $||M'.v_k||_\infty \leq m. ||v||_\infty .k.(m. ||M||_\infty)^{(4.m)^{2.m+1}}$. Nous reprenons l'expression de f^* en formule de Presburger donnée dans la preuve du théorème 4.4.2, et notons F la formule

$$\bigvee_{r=0}^{a-1} [(x' = M^r.x + v_r) \wedge (i = r)] \bigvee_{r=0}^{b-1} \left[\begin{array}{l} (n \geq 0) \\ \wedge(x' = M^{a+r}.x + v_{a+r} + n.M^{a+b+r}.v_b) \\ \wedge(i = a + r + n.b) \end{array} \right]$$

Les entiers a et b vérifiant $M^{a+b} = M^a$ peuvent être bornés par $0 \leq a \leq m - 1$ et $1 \leq b \leq |M^*|$. En utilisant les résultats de complexité de la section 4.3.3, on montre qu'un uba $\mathcal{A}(F)$ représentant F peut être calculé en temps et espace borné par $T_m = m. ||v||_\infty .(m.(||M||_\infty + 1))^{(4.m)^{2.m+5}}$. En utilisant la formule de Presburger représentant f^* donnée dans la preuve du théorème 4.4.2, nous montrons qu'un uba representant f^* peut être calculé en temps et espace bornés par :

$$|\mathcal{A}(G)|.2^{T_m.2^{2^{|\mathcal{A}(G)|.T_m}}}$$

Ceci prouve le théorème. □

En pratique m, $||v||_\infty$ et $||M||_\infty$ sont de l'ordre de quelques dizaines, tandis que $|\mathcal{A}(G)|$ peut aller jusqu'à plusieurs centaines de milliers (voir la section 5.6 du chapitre 5 par exemple). Le facteur limitant est donc principalement $|\mathcal{A}(G)|$.

Remarque 4.4.1. Les résultats récents de [Kla04, Ler05] permettent de déduire une meilleure borne de complexité. Leroux montre dans [Ler05] que d'un uba $\mathcal{A}(\phi)$ représentant un ensemble Presburger-définissable $[\![\phi]\!]$, on peut extraire une formule de Presburger ϕ' équivalente à ϕ et de taille polynomiale en $|\mathcal{A}(\phi)|$. Or dans [Kla04] le passage d'une formule de Presburger quelconque vers un automate binaire est démontré triple exponentiel en espace dans le pire des cas. En reprenant l'expression de f^* donnée dans la preuve du théorème 4.4.2, on déduit un algoritme d'accélération plate borné en espace par 3-EXP.

Problème rencontré dans la pratique. Considérons la fonction f présentée dans la figure 4.3, tirée du protocole TTP (section 5.6 du chapitre 5). Durant le calcul effectif par l'outil FAST de f^*, la taille des automates manipulés devient trop grande et dépasse les capacités de notre représentation[1]. Ainsi, même si les bornes de complexité données dans le théorème 4.4.1 sont des surapproximations, l'explosion en espace se produit dans des cas réels d'utilisation.

$$\begin{array}{l} Cp_1 \geq N \wedge Cp_2 < N \wedge d_{11} < C_{11} \wedge \\ d_1 + d_{11} - dA_{11} - dF_{11} - dA_{10} + dF_{10} - d_0 - d_{10} - d_{00} + dA_{00} + dF_{00} \leq 0 \\ \quad \rightarrow dF' := dF{+}1, Cp_1' := Cp_1{+}1, Cp_2' := Cp_2{+}1, dF_{11}' := dF_{11}{+}1, C_{11}' := C_{11}{+}1 \end{array}$$

FIG. 4.3: Une fonction faisant exploser notre accélération plate.

4.4.2 Accélération plate convexe

Nous nous concentrons sur l'amélioration de l'algorithme d'accélération plate, en considérant des classes plus restreintes mais toujours réalistes de fonctions Presburger-affines. Les résultats sur les translations convexes proviennent de [BFL04].

[1]Les automates manipulés par FAST sont limités à 2^{24} noeuds.

Définition 4.4.1 (Translation convexe)**.** Une translation convexe f est une fonction Presburger-affine $f = (I, v, G)$ où I est la matrice identité et G est un polyèdre convexe.

Les translations convexes ne constituent pas une restriction rédhibitoire. En effet les transitions des machines de Minksy et des VASS sont des translations convexes. Remarquons bien que c'est le domaine de définition de la fonction qui doit être convexe, et pas nécessairement l'ensemble de configurations auquel on applique la fonction.

Les translations convexes forment une sous-classe des fonctions Presburger-affines à monoïde fini, donc elles admettent une accélération plate. Cependant nous pouvons utiliser les propriétés géométriques des ensembles convexes pour diminuer la complexité de la construction de la clôture transitive. En effet, pour une translation convexe $f = (I, v, G)$ donnée, il n'est pas nécessaire de tester si tous les successeurs sont dans la garde : tester le premier et l'avant-dernier point est suffisant. La relation f^* peut se représenter alors par : $f^* = \{(x, x')|x \in G \wedge (\exists k \geq 0; x' = \bar{f}^k(x) \wedge k > 0 \Rightarrow \bar{f}^{k-1}(x) \in G)\}$.

Cette construction est déjà plus efficace que celle du cas général, mais on peut faire mieux !

Proposition 4.4.2. Soit $f = (I, v, G)$ une translation convexe. La relation f^* est égale à :

$$f^* = I \cup \{(x, x') \in G \times (G + v);\ x' - x \in \mathbb{N}.v\}$$

Démonstration. Soit $R = \{(x, x') \in G \times (G+v);\ x - x' \in \mathbb{N}.v\}$. Considérons $(x, x') \in f^*$ et prouvons que $(x, x') \in I \cup R$. Il existe $n \geq 0$ tel que $x' = f^n(x) = x + n.v$. Si $n = 0$ alors $(x, x') = (x, x) \in I \cup R$, sinon $n \geq 1$. À partir de $f^{n-1}(x) \in G$, nous déduisons que $f^n(x) \in f(G) = G+v$. Donc $(x, x') \in I \cup R$. Prouvons maintenant le sens inverse en considérant $(x, x') \in I \cup R$. Remarquons que si $(x, x') \in I$, alors $(x, x') \in f^*$. Aussi nous pouvons supposer que $(x, x') \notin I$. Dans ce cas, $(x, x') \in G \times (G + v)$ et il existe $n \geq 1$ tel que $x' = x + n.v$. Comme $x + n.v \in G+v$, nous avons $x + (n-1).v \in G$. Comme G est un ensemble convexe et que x et $x + (n-1).v$ sont dans G, pour tout $k \in \{0, \ldots, n-1\}$, $\bar{f}^k(x) = x + k.v \in G$. D'où $x' = f^n(x)$ et nous avons prouvé $(x, x') \in f^*$. □

On montre maintenant que cette construction a une complexité en temps et en espace au plus quadratique dans la taille du domaine, et au plus exponentielle dans le nombre de compteurs.

Théorème 4.4.3. *Soit $f = (I, v, G)$ une translation convexe. Un* uba $\mathcal{A}(f^*)$ *représentant la relation f^* peut être calculé en temps et en espace borné par*

$$|\mathcal{A}(f^*)| \leq 16.|\mathcal{A}(G)|^2.(4.m.\,||v||_\infty + 1)^{3.m}$$

Démonstration. Soit R la relation $R = \{(x, x') \in \mathbb{Z} \times \mathbb{Z};\ x' - x \in \mathbb{N}.v\}$. Nous notons $I_0 = \{i \in \{1, \dots, m\};\ v_i \neq 0\}$ et $I = \{i \in \{1, \dots, m\};\ v_i = 0\}$. Remarquons que si $I_0 = \emptyset$ alors $f^* = I$ et dans ce cas $|\mathcal{A}(f^*)| = 4$. Aussi nous pouvons supposer qu'il existe un indice $i_0 \in I_0$. Soit ϕ la formule de Presburger : $\phi := (x'_{i_0} - x_{i_0} \geq 0)$ si $v_{i_0} > 0$ et $\phi := (x'_{i_0} - x_{i_0} \leq 0)$ sinon.

Nous désignons par $a[b]$ la valeur de a modulo b. Nous prouvons maintenant que R est défini par la formule suivante :

$$\phi \bigwedge_{i \in I} (x'_i = x_i) \bigwedge_{i \in I_0 \setminus \{i_0\}} ((x'_i - x_i).v_{i_0} = (x'_{i_0} - x_{i_0}).v_i) \bigwedge_{i \in I_0} (x'_i - x_i = 0[v_i])$$

Soit $(x, x') \in R$. Il existe $n \geq 0$ tel que $x' - x = n.v$. Pour tout $i \in I$, nous avons $x'_i = x_i$. De plus pour tout $i \in I_0$, nous avons $x'_i - x_i = n.v_i$ et $x'_{i_0} - x_{i_0} = n.v_{i_0}$. D'où $(x'_i - x_i).v_{i_0} = (x'_{i_0} - x_{i_0}).v_i$ et $x'_i - x_i = 0[v_i]$. De $x'_{i_0} - x_{i_0} = n_{i_0}.v_{i_0}$, nous déduisons que ϕ est vraie. Prouvons l'implication inverse en considérant la paire (x, x') telle que ϕ est vraie et pour tout $i \in I$, nous avons $x'_i = x_i$ et pour tout $i \in I_0$ nous avons $(x'_i - x_i).v_{i_0} = (x'_{i_0} - x_{i_0}).v_i$ et $x'_i - x_i = 0[v_i]$. Comme $x'_i - x_i = 0[v_i]$, il existe $n_i \in \mathbb{Z}$ tel que $x'_i - x_i = n_i.v_i$. De l'égalité $(x'_i - x_i).v_{i_0} = (x'_{i_0} - x_{i_0}).v_i$, nous déduisons que $n_i.v_i.v_{i_0} = n_{i_0}.v_{i_0}.v_i$. Comme $v_i.v_{i_0} \neq 0$, nous avons $n_i = n_{i_0}$ pour tout $i \in I_0$. En particulier nous avons $x' = x + n_{i_0}.v$. Comme ϕ est vraie, nous déduisons que $n_{i_0} \geq 0$ à partir de $x'_{i_0} - x_{i_0} = n_{i_0}.v_{i_0}$. Donc $(x, x') \in R$.

Un uba qui représente ϕ ou $(x'_i = x_i)$ peut être calculé en temps et espace borné par $2.m+1$. Le uba représentant $(x'_i - x_i).v_{i_0} = (x'_{i_0} - x_{i_0}).v_i$ peut se calculer en temps et espace borné par $m.(2.|v_{i_0}| + 2.|v_i|) + 1 \leq 4.m.\,||v||_\infty + 1$. De plus, le calcul de $x'_i - x_i = 0[v_i]$ est borné par $2.m.|v_i| + 1 \leq 2.m.\,||v||_\infty + 1$. Donc un uba représentant R peut être calculé en temps et espace borné par $(4.m.\,||v||_\infty + 1)^{2.m}$.

De l'égalité $f^* = I \cup ((G \times (G + v)) \cap R)$, nous déduisons que f^* est calculable en temps et espace borné par $|\mathcal{A}(I)|.4.|\mathcal{A}(G)|.|\mathcal{A}(G+v)|.|\mathcal{A}(R)|$. En utilisant [BB03], $G + v$ se calcule en $|\mathcal{A}(G)|.(m.\,||v||_\infty + 1)^m$. De plus rappelons que $|\mathcal{A}(I)| = 4$.

Nous avons prouvé que f^* se calcule en temps et espace borné par :

$$\begin{aligned} &|\mathcal{A}(G)|^2.16.(4.m.\,||v||_\infty + 1)^{2.m}.(m.\,||v||_\infty + 1)^m \\ &\leq 16.|\mathcal{A}(G)|^2.(4.m.\,||v||_\infty + 1)^{3.m} \end{aligned}$$

□

Comme la représentation est canonique, l'automate obtenu finalement est le même pour les deux algorithmes d'accélération plate. Par contre les calculs intermédiaires consomment moins de mémoire avec l'accélération convexe.

Pour l'accélération convexe, la complexité est quadratique en la taille de l'automate représentant la garde G, polynomiale en la plus grande constante du système $||v||_\infty$ et exponentielle en le nombre de compteurs m. C'est une amélioration notable par rapport à l'accélération standard (voir la proposition 4.4.1), où la complexité est 3-EXP en $\mathcal{A}(G)$ et $||v||_\infty$ et 5-EXP en m.

Remarque 4.4.2. Boigelot et Wolper étudient dans [BW94, Boi98] des fonctions à domaines convexes. Cependant comme leurs fonctions permettent la remise à zéro, les arguments ci-dessus ne s'appliquent pas. De plus, leur technique fonctionne pour des ensembles convexes de configurations. Notre approche n'a pas cette limitation.

4.4.3 Accélération plate de translations positives

Nous étudions une autre classe particulière de fonctions Presburger-affines : les *translations positives*. Dans ce cas l'accélération plate est linéaire dans le domaine de définition.

Définition 4.4.2 (Translation positive)**.** Une translation positive f est une translation convexe $f = (I, v, G)$ où le vecteur $v \in \mathbb{Z}^m$ vérifie $v \geq 0$ et G est un ensemble clos par le haut.

Les translations avec domaine clos par le haut correspondent typiquement aux séquences de transitions apparaissant dans les VASS. Si une séquence de transitions d'un VASS est infiniment itérable, c'est nécessairement une translation positive.

Proposition 4.4.3. Soit $f = (I, v, G)$ une translation positive. La relation f^* est égale à : $f^* = I \cup \{(x, x') \in G \times \mathbb{Z}^m;\ x' - x \in \mathbb{N}.v\}$. Un uba $\mathcal{A}(f^*)$ représentant la relation f^* peut être calculé en temps et en espace bornés par

$$|\mathcal{A}(f^*)| \leq 8.|\mathcal{A}(G)|.(4.m.\,||v||_\infty + 1)^{2.m}$$

Démonstration. Nous montrons tout d'abord que la relation f^* est égale à $f^* = I \cup \{(x, x') \in G \times \mathbb{Z}^m);\ x' - x \in \mathbb{N}.v\}$. En utilisant la proposition 4.4.2, nous obtenons que $f^* = I \cup \{(x, x') \in G \times (G + v);\ x' - x \in \mathbb{N}.v\}$. Nous notons $(1) = I \cup \{(x, x') \in G \times (G + v);\ x' - x \in \mathbb{N}.v\}$ et $(2) = I \cup \{(x, x') \in G \times \mathbb{Z}^m);\ x' - x \in \mathbb{N}.v\}$. Montrons que $(1) = (2)$. L'inclusion $(1) \subseteq (2)$ est évidente. Nous montrons $(2) \subseteq (1)$. Comme les deux ensembles contiennent l'identité, nous prenons le cas d'un couple $(x, x') \in \{(x, x') \in G \times \mathbb{Z}^m);\ x' - x \in \mathbb{N}.v\}$ tel que $x \neq x'$. Il existe $k > 0$ tel que $x' = x + k.v$. Montrons que x' appartient à $G + v$. Comme G est clos par le haut et que $v > 0$, $x'' = x + (k - 1).v$ appartient à G. Or $x' = x'' + v$. Donc $x' \in G + v$. On a bien l'égalité cherchée, et $f^* = \{(x, x') \in G \times \mathbb{Z}^m);\ x' - x \in \mathbb{N}.v\}$.

Nous notons $R = \{(x, x') \in \mathbb{Z}^m \times \mathbb{Z}^m;\ x' - x \in \mathbb{N}.v\}$. De l'égalité $f^* = I \cup ((G \times \mathbb{Z}^m) \cap R)$, nous déduisons que f^* est calculable en temps et en espace bornés par $|I|.2.|\mathcal{A}(G)|.|\mathcal{A}(R)|$. Nous avons prouvé que f^* se calcule en temps et espace borné par : $8.|\mathcal{A}(G)|.(4.m.\,||v||_\infty + 1)^{2.m}$. □

Nous résumons dans la figure 4.4 les complexités des différents algorithmes d'accélération plate introduits dans ce chapitre.

paramètre	générale	convexe	positive
$\lvert\mathcal{A}(G)\rvert$	3-EXP	quadratique	linéaire
m	5-EXP	EXP	EXP
$\lvert\lvert v\rvert\rvert_\infty$	3-EXP	poly. en m	poly. en m
$\lvert\lvert M\rvert\rvert_\infty$	3-EXP	$= 1$	$= 1$

FIG. 4.4: Complexité des différentes accélérations plates

4.4.4 Expérimentations

Les algorithmes d'accélération plate ont été implantés dans l'outil FAST (voir le chapitre 5). Nous comparons le calcul de f^* pour différentes fonctions Presburger-affines. Les résultats sont résumés dans la figure 4.5. Les transitions trans. pos. 1 et trans. pos. 2 sont des translations positives, les autres sont des translations convexes. Les translations convexes sont désignées par le système duquel elles sont tirées et leur nom. Les translations positives ont été créées pour le test. Pour chaque transition le nombre de variables du système est donné. La taille de l'automate représentant f^* est notée $|\mathcal{A}(f^*)|$. On désigne par G (resp. C,P) l'accélération plate générale (resp. convexe, positive). Le symbole "-" indique que l'accélération considérée ne s'applique pas à la transition. Le symbole "↑↑↑" indique une explosion temps/mémoire. Le symbole " ?" indique un résultat inconnu. "$> k$" indique que le calcul n'a pas terminé en moins de k secondes/Mo. Les calculs ont été effectués sur un Pentium M à 1.7Ghz avec 512 Mo de mémoire vive.

Les expérimentations montrent que l'accélération convexe et l'accélération positive passent mieux à l'échelle que l'accélération générale, conformément au théorème 4.4.3 et aux propositions 4.4.1 et 4.4.3. Pour les translations convexes, la différence devient nette quand la valeur de $|\mathcal{A}(f^*)|$ dépasse 20000. Pour la transition t_{22} du système TTP, l'accélération convexe termine en 34 secondes et 534 Mo alors que l'accélération générale explose en mémoire. Pour les translations positives les gains en temps vont jusqu'à un facteur 10 par rapport à l'accélération générale et à l'accélération convexe.

fonction f	$\|\mathcal{A}(f^*)\|$	Temps (secondes) G/C/P	Mémoire (Mo) G/C/P
r_3 (Dekker, 22 var)	1536	0,71/0,86/-	4,6/4/-
t_1 (Mesh32, 52 var)	1614	2,15/2,5/-	8/7,8/-
r_{22} (Dekker, 22 var)	1830	0,68/1,08/-	4,2/3,8/-
r_9 (TTP, 10 var)	1971	0,28/0,31/-	3,7/3,6/-
r_{10} (TTP, 10 var)	7553	0,6/0,7/-	6/7/-
$t_1 \bullet t_{39}$ (Mesh32, 52 var)	16762	12/8,5/-	30/40/-
$(t_1 \bullet t_{39})^2$ **(Mesh32, 52 var)**	**16766**	**10,35/7,4/-**	**31/13/-**
t_2 (TTP2, 19 var)	26409	5,6/2,3/-	17/18/-
$r_{22} \bullet r_3^2$ (Dekker, 22 var)	41950	18/10,2/-	52/30/-
t_{18} (TTP2, 19 var)	190986	50/9/-	400/140/-
t_{22} (TTP2, 19 var)	380332	↑↑↑/34/-	↑↑↑/534/-
$r_{22} \bullet r_3^2 \bullet r_9$ (Dekker, 22 var)	?	↑↑↑/>900/-	↑↑↑/>500/-
trans. pos. 1 (22 var)	875	0,4/0,4/0,4	3,9/3,9/3,9
trans. pos. 1 (52 var)	2045	1,35/1,7/1,35	8,4/8/8
trans. pos. 2 (22 var)	8851	1,2/1/0,1	5,6/5,2/4
trans. pos. 2 (52 var)	19073	4,8/4,6/1	13/13/11

trans. pos. 1 : $x_{22} \geq 1 \wedge x_{13} \geq 1; x'_3 := x_3 + 1$
trans. pos. 2 : $x_{22} \geq 1 \wedge x_{13} \geq 1 \wedge x_3 \geq 1 \wedge x_{11} \geq 1; x'_4 := x_4 + 1, x'_{12} := x_{12} + 2$

FIG. 4.5: Comparaisons des accélérations

4.5 Réduction pour les systèmes à compteurs

4.5.1 Une réduction dédiée aux compteurs

Nous étudions maintenant une réduction spécifique aux systèmes à compteurs définie dans [FL02], que nous appelons ici réduction par union. Nous montrons tout d'abord que cette réduction satisfait bien les critères de la définition 2.6.1 du chapitre 2.

Deux fonctions Presburger-affines $f_1 = (M_1, v_1, G_1)$ et $f_2 = (M_2, v_2, G_2)$ ont *même action* si $M_1 = M_2$ et $v_1 = v_2$. Nous définissons l'union de deux fonctions Presburger-affines f_1 et f_2 ayant même action, notée $f_1 + f_2$, par $f_1 + f_2 = (M, v, G_1 \cup G_2)$. Nous étendons cette définition aux transitions Presburger-affines $t_1 = (q, f_1, q')$ et $t_2 = (q, f_2, q')$ telles que f_1 et f_2 ont même action par $t_1 + t_2 = (q, f_1 + f_2, q')$.

Définition 4.5.1 (Réduction par union [FL02])**.** Soit un système à compteurs affine $S = (Q, T, m)$ et $k \leq 1$. On note $\{A_1, \ldots, A_n\}$ la partition de $T^{\leq k}$ en ensembles de transitions ayant même action. Alors la k-réduction par union de S, notée $S_k = (Q, T_k, m)$, est définie par $T_k = \bigcup_{A_i} \{\sum_{t \in A_i} t\}$.

Toutes les transitions t_{ij} d'un A_i ont même action, elles sont donc de la forme $t_{ij} = (q_i, (M_i, v_i, G_{ij}), q_i')$. L'ensemble T_k contient alors exactement une transition t_i pour chaque A_i et $t_i = (q_i, (M_i, v_i, \bigcup_{ij} G_{ij}), q_i')$.

Cette méthode est bien une k-réduction comme définie au chapitre 2. Premièrement tout $t' \in T_k$ vérifie bien $\xrightarrow{t'} \subseteq \xrightarrow{T^*}$. De plus, pour tout $w \in T^{\leq k}$, il existe une rlre ρ sur T_k tel que $\xrightarrow{w^*} \subseteq \xrightarrow{\rho}$. En effet, pour $w = t_1 \ldots t_n$, on prend $\rho = (\bar{t_1} \ldots \bar{t_n})^*$ où $\bar{t_i}$ est l'unique transition de T_k ayant même action que t_i. Enfin $|T_k| \leq |T^{\leq k}|$.

Dans [FL02], il est prouvé que cette technique entraine une réduction exponentielle du nombre de cycles à considérer.

Théorème 4.5.1 ([FL02])**.** *Soit le système $S = (Q, T, m)$ et $S_k = (Q, T_k, m)$ son k-réduit pour $k \geq 1$. Si le monoïde de S est fini alors $|T_k|$ est polynomial en k.*

Démonstration. La réduction fait l'union de toutes les transitions ayant la même action. Donc le cardinal de T_k est borné par le nombre de fonctions affines différentes possibles pour les transitions de $T^{\leq k}$. La démonstration présentée dans [FL02] utilise

la finitude du monoïde $\mathcal{M}^*$ de S pour montrer que pour $k \geq 1$, le nombre de M et le nombre de v possibles sont bornés polynomialement. Le cardinal de T_k est ainsi borné par $|Q|^2.|\mathcal{M}^*|.k^{|V|}$ où V est l'ensemble fini défini par $V = \{M.v_t | M \in \mathcal{M}^* \wedge t \in T\}$. □

Remarque 4.5.1. Si le système S n'a qu'une location et que $|\mathcal{M}^*| = 1$, alors le cardinal de T_k est borné par $k^{|T|}$. C'est le cas par exemple des VASS avec remise à zéro et test à zéro.

4.5.2 Accélération de boucles imbriquées

La réduction par union permet de calculer l'accélération de certaines boucles imbriquées. Aussi pouvons nous nous demander si cette réduction permet de sortir strictement du cadre de l'accélération plate, ou si au contraire ces boucles imbriquées peuvent toujours être applaties.

Pour des fonctions Presburger-affines définies sur la logique des automates binaires, la réduction permet effectivement de sortir strictement du cadre plat, c'est-à-dire qu'il existe des systèmes qui ne sont pas applatissables au sens strict, et qui sont pourtant calculables en utilisant la réduction et l'applatissement.

Proposition 4.5.1. Soit S un système à compteurs affine avec des gardes définies dans la logique des automates binaires. Alors l'utilisation de la réduction permet de calculer l'ensemble d'accessibilité de certains systèmes (S, X) non Presburger-applatissables.

Démonstration. Notre exemple est le suivant. Soit le système $S = (Q, T, 1)$ à un compteur x, une location $Q = \{q\}$ et deux transitions $t_1 = (q, f_1, q)$ et $t_2 = (q, f_2, q)$ où les fonctions Presburger-affines f_1 et f_2 sont définies par : $f_1 = (I, +1, \exists n.x = 2^n)$ et $f_2 = (I, +1, \neg\exists n.x = 2^n)$. Nous notons $T = \{t_1, t_2\}$. Nous considérons la configuration initiale réduite au singleton $\{0\}$. Soit S' la 1-réduction de S. Alors S' vaut $S' = (Q, \{(q, f, q)\}, 1)$ avec $f = (I, +1, \mathbb{Z})$. (S', x_0) est trivialement applatissable et $\text{post}_{S'}(\{0\}) = \mathbb{N}$, il suffit d'accélérer la fonction f.

Nous montrons au contraire que (S, x_0) n'est pas applatissable. Nous raisonnons par l'absurde en supposant qu'il existe $w \in T$ et $x \in \mathbb{Z}$ tels que la séquence w soit itérable un nombre arbitraire de fois à partir de x. Nous notons $w = f_{i_1} \bullet \ldots \bullet f_{i_k}$, où $i_j \in \{1, 2\}$. Nous désignons par G_{i_j} la garde de la fonction f_{i_j}. Puisque w est itérable un nombre arbitraire de fois à partir de x et que chaque f_{i_j} incrémente x de 1, toutes les expressions $x + j + n.k$ avec $1 < j < k$ et $n \in \mathbb{N}$ satisfont la garde $G_{i_{j+1}}$ et les $x + n.k$ satisfont la garde G_{i_1}. Nous envisageons deux cas. Soit aucun des G_{i_j} n'est de la forme "x est une puissance de 2". On obtient une contradiction puisqu'alors il n'y aurait aucune puissance de 2 supérieure à x. Soit il existe un G_{i_j} de la forme "x

est une puissance de 2". Donc tous les $x+j+n.k$ sont des puissances de 2, ce qui est contradictoire. On a donc prouvé qu'il n'existe aucun $w \in T$ et aucun $x \in \mathbb{N}$ tels que w est itérable infiniment à partir de x. Donc une rlre sur T ne peut calculer qu'un nombre fini de successeurs de $x_0 = 0$. Or $\mathsf{post}^*_S(x_0) = \mathbb{N}$ est infini. Donc (S, x_0) n'est pas applatissable.

Comme nous avons montré précédement que (S', x_0) est applatisssable, la propriété est vraie. □

Problème ouvert 4.5.1. Est-ce que la réduction par union permet de calculer l'ensemble d'accessibilité de systèmes à compteurs affines (avec garde Presburger-définissable) non applatissables ?

4.5.3 Réduction de la longueur des cycles à considérer

Le théorème 4.5.1 montre que la réduction par union diminue exponentiellement le nombre de cycles k à considérer. On montre dans cette section que cette technique permet de plus de diminuer la longueur des cycles k à considérer.

La définition des réductions assure que si un système S est applatissable avec des cycles de longueur k alors le système 1-réduit S_1 est applatissable avec des cycles de longueur k. La longueur des cycles à considérer avec réduction n'est jamais plus grande que la longueur des cycles sans réduction. La réduction par union permet de faire mieux, puisque pour tout k, il existe un système S applatissable avec des cycles de longueur k tel que son 1-réduit S_1 soit applatissable avec des cycles de longueur 1.

Proposition 4.5.2. Pour tout $k \in \mathbb{N}$, il existe un système à 1 compteur $(S_{|k}, x_0)$ L-applatissable tel que
- le calcul de $\mathsf{post}^*_{S_{|k}}(x_0)$ nécessite des accélérations de la forme $w^*, w \in T^k_{|k}$;
- le calcul de $\mathsf{post}^*_{S'}(x_0)$, où $S' = (Q, T', 1)$ est le 1-réduit de S, peut se faire avec des accélérations de la forme $w^*, w \in T'$.

Démonstration. Soit $k \geq 1$. Nous considérons le système à 1 compteur $S_{|k} = (Q, T_{|k}, 1)$ suivant : Q est réduit à un singleton $\{q\}$ et $T_{|k}$ est l'ensemble des transitions $t_i = (q, x \equiv i[k] \rightarrow x' := x+1, q)$ pour $i \leq k$. Nous prenons comme ensemble de configurations initiales le singleton $X_0 = \{(q, 0)\}$. Alors $\mathsf{post}^*_{S_{|k}}(X_0) = \{q\} \times \mathbb{N}$. Le calcul de l'ensemble d'accessibilité nécessite la rlre $\rho \subseteq T^*_{|k}$ définie par $\rho = w_1^* \dots w_k^*$, où les w_i sont les permutations circulaires du cycle $(t_1 \bullet \dots \bullet t_k)$, de longueur k. Le système 1-réduit de $S_{|k}$, noté S', vaut : $S' = (Q, \{t'\}, 1)$ avec $t' : x \geq 0 \rightarrow x' := x+1$.

Il s'ensuit que $\mathsf{post}_{S'}(t'^*, X_0) = post^*_{S'}(X_0)$. Le calcul de l'ensemble d'accessibilité de S' ne nécessite donc qu'un cycle de longueur 1. □

4.5.4 Expérimentations

La réduction par union et la réduction par commutation (voir le chapitre 2) ont été implantées dans l'outil FAST (voir le chapitre 5). Nous testons l'efficacité de ces techniques sur des exemples concrets de systèmes. Nous donnons aussi le résultat de la réduction par conjugaison. Nous désignons par k la longueur des cycles considérés. Le nombre de séquences valides de longueur inférieure ou égale à k est $|C^{\leq k}|$. Les cardinaux des ensembles k-réduits sont notés U, Cm, Cj pour respectivement : la réduction par union, la réduction par commutation et la réduction par conjugaison. U+Cm et U+Cm+Cj désignent l'emploi de plusieurs réductions (respectivement union et commutation, union et commutation et conjugaison). Les lignes en gras correspondent à la longueur de cycle k avec laquelle FAST calcule le point fixe.

La réduction implantée dans l'outil FAST correspond à U+Cm. Une version de U+Cm+Cj, restreinte à la longueur de cycle 2, a été implantée pour les tests à titre comparatif.

système	$\|T\|$	k	$\|C^{\leq k}\|$	U	Cm	Cj	U+Cm	U+Cm+Cj
csm	13	1	14	14	14	14	14	14
	13	**2**	**183**	**103**	**57**	**99**	**35**	**34**
consistency	8	1	9	9	9	9	9	9
	8	2	68	45	44	39	30	27
	8	**3**	**484**	**172**	**299**	**178**	**98**	?
swimming pool	6	1	7	7	7	7	7	7
	6	2	43	21	24	25	16	15
	6	3	259	56	114	97	28	?
	6	**4**	**1555**	**126**	**614**	**421**	**47**	?
	6	5	9331	252	3591	1977	86	?

FIG. 4.6: Diminution du nombre de cycles pour chaque réduction.

Les résultats reportés dans la figure 4.6 confirment que la réduction par union (U) passe mieux à l'échelle que les réductions par commutation (Cm) et par conjugaison (Cj). La combinaison de plusieurs techniques réduit considérablement le nombre de fonctions à considérer. On peut cependant remarquer que pour les faibles valeurs de k, le nombre de fonctions semble être exponen-

tiel en k plutôt que polynomial. Finalement l'utilisation des trois réductions simultanément ne semble pas devoir apporter un gain significatif par rapport à U+Cj. Cependant cette question mériterait d'être plus examinée en avant, avec une implantation adaptée à des cycles de plus grande taille.

4.6 Conclusion

Nous avons montré dans ce chapitre comment instancier le cadre de l'accélération plate défini au chapitre 2 pour la famille des systèmes à compteurs affines. Nous nous sommes intéressés plus particulièrement aux algorithmes d'accélération plate et aux réductions, dans l'optique d'une implantation efficace en pratique.

Nous proposons deux nouveaux algorithmes d'accélération plate pour les translations convexes et les translations positives. Ces deux algorithmes ont des bornes de complexité polynomiales (quadratique pour les translations convexes et linéaire pour les translations positives) dans la taille du domaine tandis que l'algorithme général est exponentiel. Les expérimentations montrent un gain net à la fois en temps et en espace quand la taille du domaine devient très grande.

Nous avons étudié la réduction par union de [FL02]. On savait que cette réduction diminuait exponentiellement le nombre de cycles d'une longueur donnée. On a montré ici que la réduction par union permet de plus de diminuer la taille des cycles à considérer, et permet également de calculer l'ensemble d'accessibilité de certains systèmes non applatissables. Enfin les expérimentations montrent que la réduction par union passe mieux à l'échelle que les réductions génériques du chapitre 2. L'emploi simultané de ces techniques réduit considérablement le nombre de cycles utiles sur les exemples traités.

Chapitre 5

FAST et la vérification du TTP

5.1 Introduction

Nous présentons dans ce chapitre l'outil FAST [BFLP03, Fas] conçu pour vérifier des propriétés d'accessibilité sur des systèmes à compteurs affines. FAST se place dans le cadre de l'accélération plate décrit aux chapitres 2 et 4. La vérification des propriétés se fait en calculant tout d'abord l'ensemble d'accessibilité.

FAST fournit un moteur d'analyse puissant capable en pratique de calculer les ensemble d'accessibilité de nombreux systèmes a priori Turing-complets. La section 5.4 donne un panorama des systèmes vérifiés avec FAST. Les comparaisons effectuées à la section 5.5.2 montrent que FAST surpasse les outils concurrents.

Le moteur de FAST est flexible puisqu'il est facilement adaptable à d'autres types de calcul sur les configurations. Nous montrons à la section 5.5.3 comment calculer des ensembles de co-accessibilité et des couvertures avec FAST. Les performances observées sur ces problèmes sont très raisonnables et parfois au niveau de celles des outils spécialisés.

Enfin l'utilisateur a la possibilité d'intéragir avec le moteur de calcul pour influencer le calcul d'accessibilité. FAST se situe ainsi à mi-chemin entre l'approche complètement automatique et l'approche vérification assistée où l'utilisateur choisit les calculs à effectuer.

De nombreux systèmes ont été vérifiés avec succès par FAST. Nous en décrivons plus particulièrement deux. Dans la section 5.6 nous présentons l'ana-

lyse du protocole industriel TTP [KG94] [1]. Ce protocole vise à assurer une communication tolérante aux pannes entre différents microprocesseurs embarqués, appelés stations. FAST permet de montrer automatiquement que le protocole TTP est correct pour N stations et 1 défaillance. De plus FAST calcule une abstraction suffisante pour prouver la correction pour 2 défaillances. Ces résultats proviennent de [BFL04]. Dans la section 5.7 nous présentons l'analyse du protocole CES de Philips pour l'échange de contenus multimedia. Ce protocole est naturellement un système à files non fiables. Après modélisation en système à compteurs, FAST vérifie automatiquement des propriétés quantitatives sur l'ensemble d'accessibilité du CES, ce qui est hors de portée des techniques habituelles sur les files. FAST permet de plus de vérifier la correction des hypothèses de modélisation des files.

5.1.1 Architecture

FAST est organisé autour d'une architecture client-serveur. Le serveur est le moteur de calcul de FAST. Il contient la définition des automates binaires, les algorithmes d'accélération plate et les heuristiques de recherche. Un parseur permet de recevoir des requêtes de l'utilisateur et de retourner les réponses. Le client est une interface graphique qui permet de communiquer des requêtes au serveur (figure 5.1). Le moteur de calcul peut aussi être utilisé seul. Le moteur de FAST est écrit en `C++` (7.400 lignes de code). La bibliothèque d'automates MONA [KMS02, Mon] fournit la base de la bibliothèque d'automates binaires. Le client est écrit en `java`.

5.1.2 Outils similaires

LASH [Las] et TREX [ABS01] partagent la même approche que FAST : modèle sous-jacent Turing-complet et calcul exact de l'ensemble d'accessibilité via des techniques d'accélération. L'outil ALV [Alv] utilise le même cadre symbolique à base d'automates binaires, mais il n'implante pas d'accélération. Ces outils sont comparés à FAST à la section 5.5.

[1] Le TTP est soutenu par un consortium d'industriels du transport incluant entre autre Audi, Airbus et Renault.

5.2 Moteur de calcul

5.2.1 Architecture logicielle

Le moteur de FAST est structuré selon le cadre de l'accélération plate. Il y a 4 classes principales : les fonctions Presburger-affines, les automates binaires, les algorithmes d'accélération et les heuristiques de recherche.

Les automates binaires sont codés en base 2, bit de poids faible en premier. La classe d'automates binaires fournit les opérations ensemblistes basiques comme l'union, l'intersection, le complément et la projection, ainsi que la synthèse d'un automate binaire à partir d'une formule de Presburger. Cette implantation utilise la bibliothèque MONA [KMS02, Mon] pour la structure d'automate sous-jacente. Pour des raisons d'efficacité l'identité des nœuds (plus quelques autres informations) est codée sur 32 bits dans MONA, aussi les automates ont au plus 2^{24} noeuds.

Les algorithme d'accélération plate pour les fonctions Presburger-affines, pour les translations convexes et pour les translations positives sont implantés. Les algorithmes peuvent être utilisés aussi bien pour du calcul en avant qu'en arrière.

L'heuristique de calcul de l'ensemble des configurations accessibles suit la procédure ACCESS3 décrite au chapitre 2. Les adaptations de la procédure générique sont décrites à la section 5.2.2. Enfin les techniques de réduction par commutation (chapitre 2) et par union (chapitre 4) ont été implantées.

5.2.2 Adaptation de l'heuristique ACCESS3

Pour utiliser la procédure ACCESS3 en pratique, il faut choisir une implantation des procédures Choisir et Watchdog. Nous décrivons ici les solutions apportées dans FAST. Nous pensons que ces choix sont suffisament généraux pour être adaptés à d'autres domaines de variables.

Procédure Choisir. Il n'y a pas de relation monotone entre la taille d'un ensemble Presburger-définissable (au sens de l'inclusion) et le nombre de nœuds de l'automate binaire associé. Aussi les automates calculés lors des étapes intermédiaires du calcul de point fixe peuvent s'avérer être beaucoup plus gros que l'automate final représentant le point fixe. Choisir essaie permettre d'éviter au maximum de telles étapes de calculs.

Choisir sélectionne le prochain $w \in T^{\leq k}$, tel que $|\text{POST_STAR}(w, \mathtt{x})| \leq |\mathtt{x}|$. S'il n'y en a pas alors le prochain (dans un ordre cyclique) est sélectionné. Nous discutons de la pertinence de cette heuristique à la section 5.4.4.

Procédure Watchdog. La procédure doit d'une part repérer suffisament tôt que la longueur k des cycles est insuffisante pour éviter des calculs inutiles, et d'autre part ne pas incrémenter inutilement k pour éviter que $|T^{\leq k}|$ ne devienne trop grand.

Appelons profondeur le nombre d'itérations dans le code k-flattable à la ligne 4 (la profondeur est remise à 0 quand k est incrémenté). Notre critère d'arrêt pour Watchdog est une limite maximale sur profondeur. Ceci est justifié par l'observation suivante. En pratique quand un k suffisamment grand est atteint, le point fixe est calculé en quelques itérations.

Complétude ? Ces implantations de Choisir et Watchdog ne respectent pas les critères d'équité pour ACCESS3 définis au chapitre 2. Aussi la terminaison n'est-elle plus assurée en théorie sur les systèmes applatissables. Cependant ce point ne semble pas très important en pratique puisque FAST calcule l'ensemble d'accessibilité de nombreux exemples, comme mentionné à la section 5.2.2.

5.2.3 Autres choix techniques

La procédure générique ACCESS3 et les structures de données et algorithmes du chapitre 4 fournissent l'ossature de FAST. Cependant de nombreux problèmes pratiques ne sont pas couverts par ces résultats. Par exemple, les locations peuvent être codées explicitement ou comme des compteurs, les cycles peuvent être calculés statiquement ou à la demande.

Nous décrivons ici quelques choix d'implantation de FAST. À l'heure actuelle la meilleure solution à chacun des problèmes listés ci-dessous n'est pas connue.

Variables dans $\mathbb{N}^m$. Les résultats du chapitre 4 sont valides pour des compteurs sur $\mathbb{Z}$. Cependant dans FAST les compteurs sont à valeurs dans $\mathbb{N}$. Ceci permet de gagner un facteur sur les calculs puisque les automates sont plus simples. De plus ce n'est pas vraiment une limitation puisque d'une part les études de cas utilisent toujours des compteurs positifs et d'autre part un compteur dans $\mathbb{Z}$ peut toujours être codé par deux compteurs positifs $x^+, x^- \in \mathbb{N}$ tels que $x = x^+ - x^-$ et $(x^+ = 0 \vee x^- = 0)$.

Codage des locations. En suivant la remarque 4.2.1 (chapitre 4, section 4.2), les locations et les booléens sont codés dans des variables de compteur. D'un côté cela permet de mieux partager la structure de l'ensemble d'accessibilité et d'éviter un produit explicite des structures de contrôle quand le système a plusieurs composantes. De l'autre, les variables de contrôle sont considérées comme des variables quelconques : on ne tire pas parti de leur caractère borné ce qui peut être génant pour des systèmes manipulant massivement des booléens et seulement quelques compteurs. Une solution est peut-être de coder le contrôle par un `bdd`, intégré à l'automate binaire.

Calcul statique des cycles. Nous avons choisi de calculer statiquement, au début de la procédure k-applatissable, l'ensemble $T^{\leq k}$. Les études de cas de la section 5.4 montrent que cette approche est utilisable en pratique grâce aux réductions.

5.3 Entrées-sorties

FAST prend en entrée un fichier contenant une description du système à compteurs à analyser, et une stratégie permettant à l'utilisateur d'avoir "un contrôle" sur ce que FAST doit calculer. Les sorties sont des messages indiquant si le système est sûr ou non. Enfin une interface graphique est disponible pour faciliter la prise en main de l'outil.

5.3.1 Le système

La description du système à compteurs se fait dans un format propre à FAST. Cependant comme beaucoup de nos études de cas sont des réseaux de Petri (VASS) étendus, nous avons développé un outil [BP04] qui tranforme une description de réseau de Petri étendu au format PNML en un système dans le format de FAST. Le langage PNML [BCvH$^+$03] est un format en cours de standardisation pour la description de réseaux de Petri étendus.

5.3.2 La stratégie

La stratégie est un script décrivant la suite de calculs à enchaîner pour vérifier la validité du système. Le langage de script opère sur des variables de type `region` (ensemble de configurations), `transition` et `boolean`. Tous les opérateurs usuels sur les ensembles sont fournis. L'utilisateur peut définir des ensembles de transitions $T' \subseteq T^*$ et les primitives pour calculer $\mathsf{post}(T', X_0)$ et $\mathsf{pre}(T', X_0)$ sont fournies. Une analyse basique peut ainsi être spécifiée en

seulement quatre instructions : déclarer la région initiale X_0, lancer le calcul des accessibles $\mathsf{post}^*(X_0)$, déclarer la région P représentant la propriété à vérifier et enfin tester $\mathsf{post}^*(X_0) \subseteq P$.

Le langage offre d'autres possibilités pour guider l'outil plus finement. On peut par exemple analyser incrémentalement des systèmes en les décomposant en plusieurs sous-systèmes plus simples à analyser (voir l'analyse du protocole TTP à la section 5.6), indiquer des cycles à utiliser en priorité, choisir l'algorithme d'accélération (standard, convexe ou positif) et enfin régler l'heuristique. Deux paramètres sont modifiables. Pour la sous-procédure Choisir, on peut opter pour l'implantation décrite à la section 5.2.2 ou pour une énumération cyclique. Pour Watchdog, on peut régler la profondeur d'arrêt.

Ce langage de script offre une liberté appréciable à l'utilisateur, qui devient cruciale quand les méthodes automatiques ne suffisent pas. FAST se situe ainsi à mi-chemin entre l'approche complètement automatique, justifiée quand la terminaison est garantie mais restrictive sinon, et l'approche vérification assistée où l'utilisateur choisit tous les calculs à effectuer, impossible sur de gros systèmes.

5.3.3 Interface graphique

Une interface graphique est disponible [BFL+] pour faciliter la prise en main de l'outil. Cette interface fournit une édition assistée des systèmes et des stratégies, avec du pretty printing et des stratégies prédéfinies. Une fois le calcul commencé, l'interface fournit un retour à l'utilisateur par le biais de différentes mesures et graphiques (mémoire consommée, temps écoulé, etc.).

5.4 Expérimentations

Dans cette section nous validons l'outil FAST sur un large jeu de tests. Tout d'abord nous expérimentons la capacité de FAST à calculer les ensembles d'accessibilité de ces systèmes. Les bons résultats obtenus valident l'architecture et les choix centraux de l'outil. Ensuite nous justifions l'heuristique de FAST en la comparant à une autre heuristique.

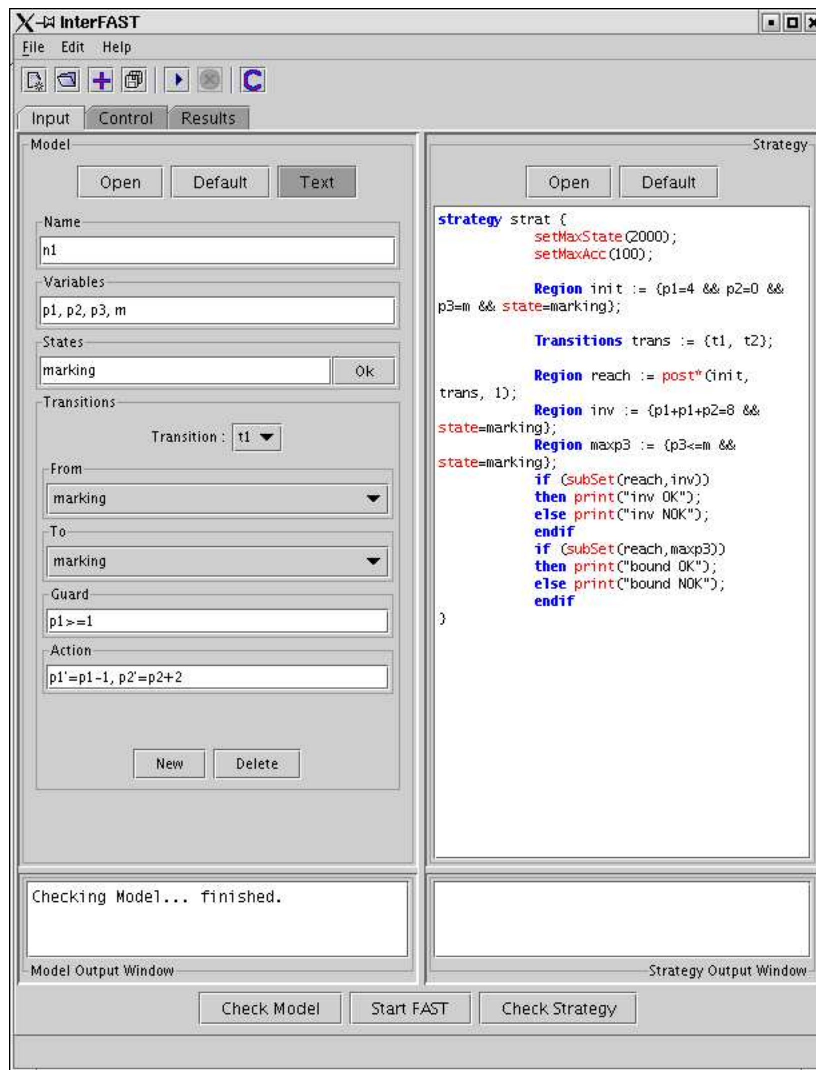


FIG. 5.1: Interface utilisateur de FAST.

5.4.1 À propos des tests

Tous les tests de ce chapitre sont effectués sur un Intel Pentium 933 Mhz avec 512 Mo. Les temps sont donnés en secondes et la mémoire en Mo. FAST est utilisé avec les réglages standards : heuristique ACCESS3 avec les adaptations de la section 5.2.2, accélération plate générale, pas de stratégie élaborée de calcul. Les abréviations utilisées pour les tests dans ce chapitre sont résumées dans la figure 5.2.

Symbole	Signification
m	nombre de compteurs
T	ensembles des transitions
k	taille des circuits utilisés par FAST
$C^{\leq k}$	circuits de longueur k
$\|\mathcal{A}\|$	nombre de nœuds de l'automate binaire calculé
$\|\rho\|$	longueur de la rlre construite par ACCESS3
$\uparrow\uparrow\uparrow$	dépassement de capacité mémoire
$\geq c$	indique un temps de calcul (mémoire de calcul)
	supérieur à c secondes (Mo)
?	valeur inconnue
-	le calcul ne s'applique pas

FIG. 5.2: Abréviations utilisées

5.4.2 Le jeu de tests

Nous utilisons l'ensemble des exemples et études de cas analysés par les outils ALV, BABYLON[2], BRAIN, LASH ou TREX. Ces 37 systèmes sont disponibles sur [Fas].

Les exemples vont des puzzles académiques complexes comme le swimming pool [FO97b] à des études de cas industriels comme le protocole de cohérence de cache Futurebus. Ils sont répartis en trois catégories : systèmes à compteurs à ensemble d'accessibilité fini, systèmes à compteurs monotones à ensemble d'accessibilité infini, systèmes à compteurs affine à ensemble d'accessibilité infini.

[2] http://www.ulb.ac.be/di/ssd/lvbegin/CST/

5.4.3 Résultats

Le tableau suivant montre le comportement de FAST sur les différents exemples. Les résultats sont ceux du calcul en avant de l'ensemble d'accessibilité. Le nombre de cycles $|C^{\leq k}|$ est donné après réduction (union et commutation).

Système	m	$\|T\|$	sec	Mo	$\|\rho\|$	k	$\|C^{\leq k}\|$
systèmes à compteurs bornés							
Producer/Consumer	5	3	0,41	2,37	3	1	3
RTP	9	12	2,24	2,76	8	1	12
Lamport ME	11	9	2,70	2,88	11	1	9
Reader/Writer	13	9	9,68	23,14	23	1	9
Peterson ME	14	12	4,97	3,78	12	1	12
Dekker ME	22	22	21,72	5,48	36	1	22
systèmes à compteurs monotones non bornés							
Manufacturing	7	6	$\geq$ 1800	?	?	?	?
swimming pool	9	6	111	29,06	9	4	47
CSM	13	13	45,57	6,31	32	2	35
Kanban	16	16	10,43	6,54	2	1	16
Multipoll	17	20	22,96	5,13	13	1	20
FMS	22	20	157,48	8,02	23	2	46
extended ReaderWriter	24	22	$\geq$ 1800	?	?	?	?
pncsa	31	38	$\geq$ 1800	?	?	?	?
Mesh2x2	32	32	$\geq$ 1800	?	?	?	?
Mesh3x2	52	54	$\geq$ 1800	?	?	?	?
Systèmes à compteurs non bornés							
Synapse Cache Coherence	3	3	0,30	2,23	2	1	3
Berkeley Cache Coherence	4	3	0,49	2,75	2	1	3
M.E.S.I. Cache Coherence	4	4	0,42	2,44	3	1	4
M.O.E.S.I. Cache Coherence	4	5	0,56	2,49	3	1	5
lift controller - N	4	5	4,56	2,90	4	3	20
Illinois Cache Coherence	4	6	0,97	2,64	4	1	6
Firefly Cache Coherence	4	8	0,86	2,59	3	1	8
Dragon Cache Coherence	5	8	1,42	2,72	5	1	8
Esparza-Finkel-Mayr	6	5	0,79	2,55	2	1	5
ticket 2i	6	6	0,88	2,54	5	1	6
ticket 3i	8	9	3,77	3,08	10	1	9
barber m4	8	12	1,92	2,68	8	1	12
bakery	8	20	$\geq$ 1800	?	?	?	?
Futurebus+ Cache Coherence	9	10	2,19	3,38	8	1	10
Consistency	12	8	200	7,35	9	3	98
Central Server	13	8	20,82	6,83	11	2	25
Last-in First-served	17	10	1,89	2,74	12	1	10
Producer/Consumer Java - 2	18	14	13,27	3,81	53	1	14
Producer/Consumer Java - N	18	14	401,5	12,46	86	2	75
Inc/Dec	32	28	$\geq$ 1800	?	?	?	?
2-Producer/2-Consumer Java	44	38	$\geq$ 1800	?	?	?	?

FIG. 5.3: FAST en pratique

Les études de cas sont très encourageantes, FAST calculant l'ensemble d'accessibilité de 78% des systèmes considérés. Ce pourcentage passe à 74%

si l'on considère uniquement les systèmes à ensemble d'accessibilité infini. Nous verrons à la section 5.5.2 que FAST dépasse les outils similaires.

Ces bons résultats permettent de valider les choix au cœur de FAST. Tout d'abord tous les exemples s'expriment naturellement en termes de systèmes à compteurs affines. Ensuite l'hypothèse du monoïde fini est toujours vérifiée sur ces exemples. Les systèmes considérés sont applatissables dans au moins 78% des cas et donc au moins 78% de ces systèmes ont un ensemble d'accessibilité Presburger-définissable. Enfin dans 19% des tests, la longueur des cycles utilisés est strictement supérieure à 1. Cette proportion passe à 22% si l'on considère uniquement les systèmes à ensemble d'accessibilité infini. Ceci montre l'utilité d'accélérer des cycles et pas seulement des boucles.

En fait il semble que les limitations de l'outil soient plus d'ordre pratique, temps et espace de calcul, que d'ordre théorique. Les facteurs cruciaux sont bien entendu le nombre de variables, mais surtout la structure de l'espace d'accessibilité et la longueur k des cycles nécessaires au calcul. En effet quand k devient trop grand, le calcul statique de cycles devient très coûteux.

5.4.4 Validation de l'heuristique

Nous validons ici l'heuristique décrite à la section 5.2.2. Nous comparons l'implantation décrite de ACCESS3 à une implantation dans laquelle la procédure Choisir se contente d'énumérer les $w \in C^{\leq k}$. Plus exactement Choisir ordonne arbitrairement les $w \in C^{\leq k}$. Nous notons $w_1, \ldots, w_{|C^{\leq k}|}$ cet ordre. Alors à létape de calcul i, Choisir sélectionne $w_{i'}$ où i' est égal à i modulo $|C^{\leq k}|$. Nous notons `enum-cyclique` cette version de ACCESS3. Les résultats sont présentés dans la figure 5.4. Le temps et la mémoire consommés par `enum-cyclique` sont présentés dans les colonnes Temps et Mémoire. Le temps et la mémoire consommés par l'heuristique de FAST sont rappelés entre parenthèses.

L'heuristique de FAST consomme toujours moins de mémoire que l'heuristique `enum-cyclique` et dans le cas du swimming pool, FAST termine en utilisant 30 Mo tandis que le calcul par énumération cyclique sature la mémoire. De plus à part pour le système Producer/Consummer, l'heuristique de FAST s'avère plus rapide. La taille réduite des automates calculés permet de simplifier les calculs, et compense le surcoût lié à la sélection plus fine des cycles.

Système	m	$\|T\|$	k	$\|\mathcal{A}\|$	Temps enum-cyclique (FAST)	Mémoire enum-cyclique (FAST)
swimming pool	9	6	4	583	↑↑↑(111)	↑↑↑(30)
Consistency	12	8	3	86	205 (200)	70 (70)
Central Server system	13	8	2	69	32 (20,95)	7 (7)
Reader/Writer	13	9	1	117	12 (9,7)	24 (23)
CSM	13	13	2	59	57 (45)	6 (6)
Peterson ME	14	12	1	60	6 (5)	4,4 (4,4)
Kanban	16	16	1	49	21 (10,5)	9,6 (7)
Multipoll	17	20	1	559	28 (23)	5 (5)
Producer/Consumer java	18	14	2	2023	140 (400)	14 (13)
FMS	22	20	2	380	200 (157)	17 (7)
Dekker ME	22	22	1	259	10 (22)	7 (6,9)

FIG. 5.4: Comparaison des heuristiques

Ces résultats confortent les choix faits à la section 5.2.2 dans la définition de l'heuristique de FAST.

5.5 Comparaison à d'autres outils

Nous comparons maintenant les performances de FAST à celles d'autres outils. Nous faisons deux jeux de comparaisons distincts. Le premier compare FAST et les outils les plus proches ALV, LASH et TREX, pour évaluer leurs performances sur du calcul d'accessibilité de systèmes à compteurs. Le second jeu de comparaison évalue la flexibilité des technologies mises en œuvre dans FAST, en utilisant l'outil pour du calcul de co-accessibilité et de couverture de systèmes à compteurs monotones et en comparant les résultats obtenus à ceux d'outils spécialisés.

5.5.1 Les différents outils

Tout d'abord nous présentons ALV, LASH et TREX et nous les comparons à FAST via le cadre de l'accélération plate.

ALV[Alv] considère des systèmes à compteurs sans restriction. Les régions sont des automates binaires. Il n'y a pas d'accélération, et l'heuristique de calcul est similaire à ACCESS1. ALV encode les variables booléennes comme des bdd.

LASH[Las] travaille sur des systèmes à compteurs affines avec des automates binaires. L'accélération plate est implantée pour des fonctions dont le monoïde est fini (voir le chapitre 4). Sans aide de la part de l'utilisateur

LASH est restreint à l'accélération de boucle (ACCESS2 avec w choisi dans T au lieu de T^*) car aucune recherche de cycle n'est disponible.

TREX[ABS01] utilise des systèmes à compteurs restreints aux mêmes opérations que des automates temporisés[3] : les gardes sont des conjonctions finies de contraintes $x_i \leq x_j + c$ et les actions ont la forme $x_i := x_j + c$ où x_i est une variable, c est une constante dans $\mathbb{Z}^m$ et x_j est une variable ou la constante 0. Les régions manipulées sont les pdbm : des dbm sur des paramètres contraints par une formule arithmétique. Aussi l'inclusion est-elle indécidable. Une procédure d'accélération est disponible, qui permet au moins toutes les accélérations de FAST et LASH (voir la section 2.4.2). Cependant cette procédure produit des formules de l'arithmétique et est non-récursive. L'heuristique utilisée est ACCESS2 restreinte aux séquences $C^{\leq k}$, pour un k défini statiquement par l'utilisateur. [DFV04] propose une comparaison approfondie de FAST et de TREX. Remarquons que TREX n'essaie pas de calculer tous les cycles de longueur inférieure ou égale à k comme FAST mais les découvre à la volée lors du parcours.

Le tableau 5.5 fait le point entre les différents outils, en se basant sur les critères du cadre de l'accélération plate (cf. chapitre 2). La colonne "terminaison" indique la classe de systèmes pour laquelle l'heuristique termine (A : applatissable, k-A : k-applatissable, Unif-b : uniformément borné). La direction indique si l'outil peut calculer en avant (Av) et/ou en arrière (Ar).

	système	rep. symb.	accélération	terminaison	direction
ALV	complet	aut. binaire	non	Unif-b	Av-Ar
FAST	affine	aut. binaire	plate	A	Av-Ar
LASH	affine	aut. binaire	boucle	1-A	Av-Ar
TREX	restreint	pdbm	interpolation	k-A (*)	Av

(*) La terminaison se fait modulo un oracle pour les tests d'inclusion.

FIG. 5.5: Différents outils pour la vérification de systèmes à compteurs.

[3] En fait TREX est conçu pour l'analyse de systèmes avec compteurs et horloges, nous prenons ici sa restriction aux compteurs.

5.5.2 Comparaison calcul en avant exact

Nous comparons maintenant la capacité des outils ALV, LASH, FAST et TREX à calculer des ensembles d'accessibilité, de manière exacte et en avant [4]. Les systèmes à compteurs choisis ont tous un ensemble d'accessibilité infini, sauf les systèmes RTP, Lamport et Dekker.

Les résultats sont présentés dans le tableau 5.6. Nous ne donnons qu'un échantillon représentatif des tests effectués. Un calcul réussi de l'ensemble d'accessibilité en moins de 1200 secondes est noté T et son échec en moins de 1200 secondes est noté ↑. Le symbole − indique que le système ne peut être modélisé dans TREX.

Système	ALV(*)	LASH	FAST	k	TREX
RTP (borné)	T	T	T	1	T
Lamport (borné)	T	T	T	1	T
Dekker (borné)	T	T	T	1	T
ticket 2	T	T	T	1	T
kanban	↑	T	T	1	T
multipoll	↑	T	T	1	↑
prod/cons (2)	↑	T	T	1	-
prod/cons (N)	↑	↑	T	2	-
lift control, N	↑	↑	T	2	T
train	↑	↑	T	2	T
csm, N	↑	↑	T	2	↑
consistency	↑	↑	T	3	-
swimming pool	↑	↑	T	4	↑
pncsa	↑	↑	↑	?	↑
incdec	↑	↑	↑	?	↑
bigjava	↑	↑	↑	?	↑

(*) Ces résultats sont cohérents avec ceux de Bultan et Bartzis dans [BB04].

FIG. 5.6: Comparaison de différents outils

Les résultats expérimentaux montrent un décrochage des outils ALV et LASH à mesure que k augmente. FAST respecte complètement le cadre de l'accélération plate et a les meilleurs résultats de terminaison. À l'opposé, l'outil ALV qui n'intègre aucun mécanisme d'accélération ne termine pas sur ces exemples complexes. Entre les deux, LASH propose une accélération plate

[4] Les propriétés ne sont pas vérifiées à la volée.

restreinte aux boucles. Il termine seulement sur les exemples les plus simples ($k = 1$). Remarquons que LASH a des performances similaires à celles de FAST lorsqu'on lui indique les cycles à accélérer. La différence se fait donc sur la recherche de cycles, et pas sur l'efficacité de l'implantation des automates binaires. Enfin les performances de TREX sont moins nettement corrélées à k puisque l'outil termine pour le système lift où $k = 2$ et échoue pour le système multipoll alors que $k = 1$.

Ces résultats montrent une forte corrélation entre la terminaison pratique et le respect du cadre de l'accélération. La comparaison entre ALV et LASH prouve l'utilité de l'accélération, et la comparaison entre FAST et LASH prouve l'intérêt de rechercher des cycles de taille arbitraire.

Les résultats de TREX montrent que le cadre symbolique des pdbm n'est pas le mieux adapté aux systèmes à compteurs[5], puisque d'une part de nombreux systèmes ne sont pas représentables et d'autre part malgré l'accélération la terminaison en un temps raisonnable est moins fréquente.

5.5.3 Co-accessibilité et couverture

La technologie des automates binaires est très souple et peut être adaptée à de nombreux types de calculs sur les configurations d'un système. Nous testons ici les performances de FAST pour le calcul en arrière et le calcul de couverture. À chaque fois FAST est comparé à un outil spécialisé.

Calcul exact en arrière. Le calcul en arrière est testé sur des extensions monotones de VASS avec configurations closes par le haut. Ainsi la terminaison du calcul symbolique standard est assurée, et l'ensemble d'accessibilité est clos par le haut. Les outils spécialisés pour ce type de calculs intègrent des structures de données légères et le calcul des prédécesseurs est efficace.

Nous considérons en plus de FAST et ALV les deux approches suivantes : BRAIN et le calcul par cst. BRAIN[RV02, Bra] manipule des linéaires représentés par base/périodes. Le cadre des Covering Sharing Tree, ou cst [DRV04], a été développé par Delzanno, Raskin et Van Begin. Les cst sont des bdd dans lesquels les nœuds sont étiquetés par des intervalles et non des booléens. Les cst s'avèrent particulièrement adaptés aux calculs de co-accessibilité sur des VASS monotones.

[5] Ce cadre est prévu à la base pour des systèmes à compteurs et horloges.

Les temps de calculs (en secondes) sont résumés dans la figure 5.7. Les résultats sur les `cst` sont repris de [GRV05]. Certains protocoles sont testés en rajoutant dans les gardes des invariants pour simplifier le calcul. Ils sont notés `nom-inv`. FAST est utilisé sans accélération, avec un calcul itératif de prédécesseurs (noté FAST-pred).

	ALV	BRAIN	CST	FAST-pred
csm 4	18	0.8	0.22	70
csm 6	108	5	(*)	156
csm 8	354	25	(*)	300
csm 10	960	100	(*)	720
csm 12	1934	328	(*)	1200
consprod 2	↑↑↑	1268	(*)	800
consistency	4.6	0	(*)	200
incdec	560	39	(*)	2480
csm-inv 50	0.2	7.8	0.18	2
consistency -inv	1	5	(*)	1.4
consprod2 -inv	0	0	0	38
incdec-inv	1	5	(*)	920

(*) les résultats ne sont pas disponibles.

FIG. 5.7: Comparaison pour le calcul exact en arrière

Tout d'abord on peut remarquer que FAST et ALV se comportent de manière globalement similaire, même si sur certains exemples l'un ou l'autre a clairement l'avantage (consprod2 pour FAST, incdec pour ALV). Par contre FAST tire moins parti des invariants. L'outil BRAIN a globalement de meilleures performances que ces deux outils. Les quelques résultats disponibles sur les `cst` laissent penser que cette approche est la plus efficace sur ce genre de calcul.

Sans surprise l'outil le plus spécialisé est le plus efficace. Cependant ces tests montrent que FAST est tout à fait utilisable pour des calculs de co-accessibilité, avec des temps de calculs très corrects.

Ensembles de couverture. La couverture d'un système est la clôture par le bas de son ensemble d'accessibilité [Fin93]. Dans le cas des VASS cette approximation suffit à vérifier des propriétés d'accessibilité closes par le haut par exemple. Karp et Miller ont montré en 1968 que la couverture d'un VASS est calculable, mais leur algorithme n'est pas élémentaire [Fin93].

Récemment Raskin, Gaeerts et Van Begin ont proposé un calcul par raffinements successifs de la couverture de VASS monotones efficace en pratique [GRV04, GRV05].

L'opération de couverture par le bas s'exprime directement par la relation de Presburger $\{(x, x')|x \leq x'\}$. Nous intégrons ce calcul dans la procédure ACCESS3 : après chaque opération POST_STAR, le résultat est clos par le bas. L'intérêt est double : d'une part la terminaison est assurée sur les extensions monotones de VASS, d'autre par comme les ensembles clos par le bas sont plus simples on peut espérer de meilleurs performances. Les tests sont répertoriés dans la figure 5.8. La colonne FAST-couv liste les résultats de calcul de couverture avec FAST. Les temps sont donnés en secondes.

	k	FAST-couv	FAST	EEC
consprod N	1	>1800	400	0.15
multipoll	1	20	23	2
kanban	1	18	10	12.5
csm	2	10.2	45	0.05
fms	2	150	157.5	163

FIG. 5.8: Temps de calcul d'arbres de couverture

Si sur certains exemples le calcul de couverture de FAST fait jeu égal avec la technique spécialisée EEC, les écarts peuvent être très importants. De plus le calcul de la couverture n'est pas forcément beaucoup plus rapide que celui de l'ensemble d'accessibilité, même si les automates calculés sont beaucoup plus petits. En fait notre clôture par le bas est très coûteuse, ce qui annule le bénéfice obtenu en simplifiant les automates. Il faudrait probablement intégrer cette opération directement dans l'algorithme d'accélération pour obtenir de meilleurs résultats.

5.5.4 Commentaires

FAST s'avère être un outil très efficace pour le calcul en avant d'ensembles d'accessibilité. Sur les expérimentations FAST surpasse les outils similaires ALV, LASH et TREX. De plus les tests prouvent la flexibilité du moteur de FAST qui peut être facilement adapté à d'autres types de calcul (co-accessibilité, couverture) avec des performances correctes, dépassant parfois les outils spécialisés.

5.6 Vérification du protocole TTP

Cette section décrit la vérification du protocle TTP avec FAST. Dans des travaux antérieurs, le protocole a été vérifié correct par des méthodes manuelles (nombre arbitraire de défaillances) et semi-assistées avec les outils ALV et LASH pour une défaillance, mais ces outils sont insuffisants pour deux défaillances. FAST vérifie automatiquement la correction du protocole pour une défaillance, et avec des abstractions la correction pour deux défaillances est prouvée.

5.6.1 Présentation du protocole TTP

Nous considérons des microprocesseurs embarqués, appelés ici stations, communicant tous entre eux. Le TTP sert à prévenir la partition des stations en cliques isolées après une panne. Une clique est un sous-ensemble de stations qui ne communiquent plus qu'entre elles. En comportement normal, il ne doit y avoir qu'une seule clique. Le protocole assure que suite à une défaillance, après un certain temps, les stations qui sont encore actives appartiennent toutes à la même clique.

Description succinte. Le temps [6] est divisé en tours. Chaque tour est divisé en autant de créneaux de communication qu'il y a de stations. Le protocole se déroule comme suit (une description plus complète peut être trouvée par exemple dans [KG94, BM02]).

1. Chaque station s_i à l'information suivante : une liste l_i de booléens déclarant pour chaque station s_j, si oui ou non s_i la *considère* comme valide, et deux compteurs C_{Ack} et C_{Fail}.
2. À chaque créneau, une seule station a le droit d'émettre, les autres écoutent. La station émettrice diffuse son message à toutes les autres. Le message envoyé est la liste l_i.
3. Quand une station s_j reçoit un message d'une station s_i : si $l_j \neq l_i$, ou si rien n'est reçu, la station s_i est considérée comme invalide. La liste l_j est mise à jour, et C_{Fail} est incrémenté. Sinon, C_{Ack} est incrémenté.
4. Quand une station s_i est à son créneau de transmission : si $C_{Ack} \leq C_{Fail}$, alors la station s_i se *considère invalide*, elle devient inactive et n'émet plus. Sinon, C_{Ack} et C_{Fail} sont remis à 0 et l_i est diffusé à toutes les stations.

[6] Les horloges sont synchronisées par d'autres mécanismes.

5.6.2 Modélisation

Nous partons de la modélisation de Merceron et Bouajjani dans [BM02]. Cette modélisation considère un nombre arbitraire N de stations, mais seulement un nombre fixé de défaillances. Merceron et Bouajjani fournissent une famille de systèmes à compteurs indexée par le nombre de défaillances.

Le système à compteurs pour $P = 1$ est donné dans la figure 5.10. La variable N est le nombre initial de stations. La variable C_W (resp. C_F) est le nombre de stations actives (resp. inactives). La variable C_p compte le nombre de créneaux de transmission écoulés pendant le tour. Un tour compte N créneaux de transmission. Aussi, quand $C_p = N$, C_p est remis à 0 et un nouveau tour commence. La location `normal` représente le fonctionnement normal du protocole. Quand une défaillance se produit, le protocole entre dans un comportement anormal. La location `Round1` est le premier tour suivant l'erreur, la location `later` les autres tours. Une défaillance divise les stations actives en deux cliques de stations actives C_1 et C_0. Les stations de C_1 (resp. C_0) ne communiquent qu'entre elles. Nous notons par C_1 et C_0 le nombre de stations dans les cliques C_1 et C_0. La variable d (resp. d_0, d_1, d_F) est le nombre de stations actives (resp. de la clique C_0, de la clique C_1, défaillantes) ayant déjà émis durant le tour.

La propriété de sûreté à vérifier est que deux tours après la défaillance une des cliques est vide (les stations actives communiquent toutes entre elles), ce qui se traduit par : (P_1) $location = later \wedge C_p = N \Rightarrow (C_1 = 0 \vee C_0 = 0)$.

Remarque 5.6.1. Cette spécification est incomplète, il faudrait en fait vérifier : *pour tout chemin, si la défaillance survient alors le protocole atteint la location `later` et (P_1) est satisfaite.* Alors que nous vérifions : *pour tout chemin, si la défaillance survient et que le protocole atteint la location `later` alors (P_1) est satisfaite.* Cependant cette propriété sort du cadre des propriété de sûreté auquel nous nous sommes restreints.

Originalité du TTP. Le nombre de variables n'est pas très élévé (9), et les actions sont classiques. L'originalité de ce système à compteurs tient dans les gardes, des inégalités linéaires mêlant un nombre important de variables. Ce système n'appartient pas aux familles classiques de VASS étendus ou aux systèmes à compteurs restreints de TREX. De plus la forte interconnexion des variables (via les gardes) confère à l'ensemble d'accessibilité une structure très complexe, bien que Presburger-définissable.

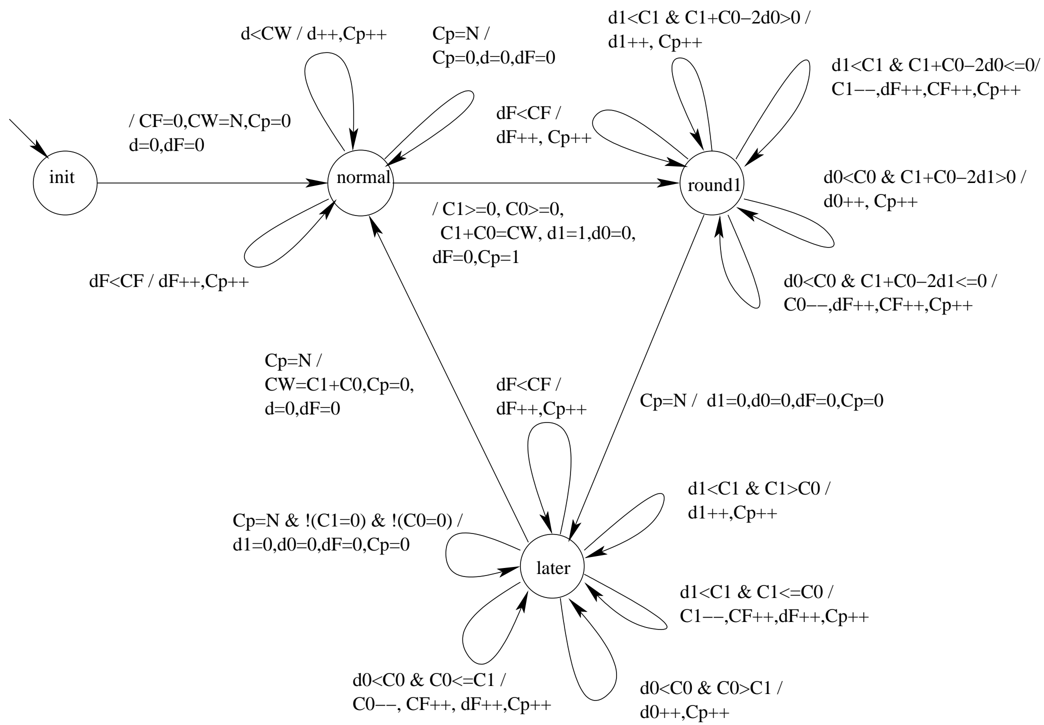


FIG. 5.9: Modélisation du protocole TTP, 1 défaillance et N stations

Intérêt du TTP. Le protocole TTP est une étude de cas intéressante pour nous, puisque (1) c'est une étude de cas industriel (2) Bouajjani et Merceron proposent une famille infinie de systèmes à compteurs de plus en plus complexes, ce qui est utile pour tester les limites des outils.

5.6.3 Vérification automatique pour une défaillance

Le système à compteurs de la figure 5.9 n'est pas affine à cause de l'affectation non déterministe de la location `normal` à la location `round1`. La transition de `later` à `normal` modélise le retour au comportement normal du protocole. Comme la propriété P_1 ne concerne que la location `later` et qu'à la location `normal` le protocole revient dans son état initial, nous pouvons enlever la transition de `later` à `normal`. L'affectation non déterministe ne se produit alors qu'une fois et peut être codée dans la configuration initiale du système. Le système à compteurs affine obtenu est présenté dans la figure 5.10. Son monoïde est fini.

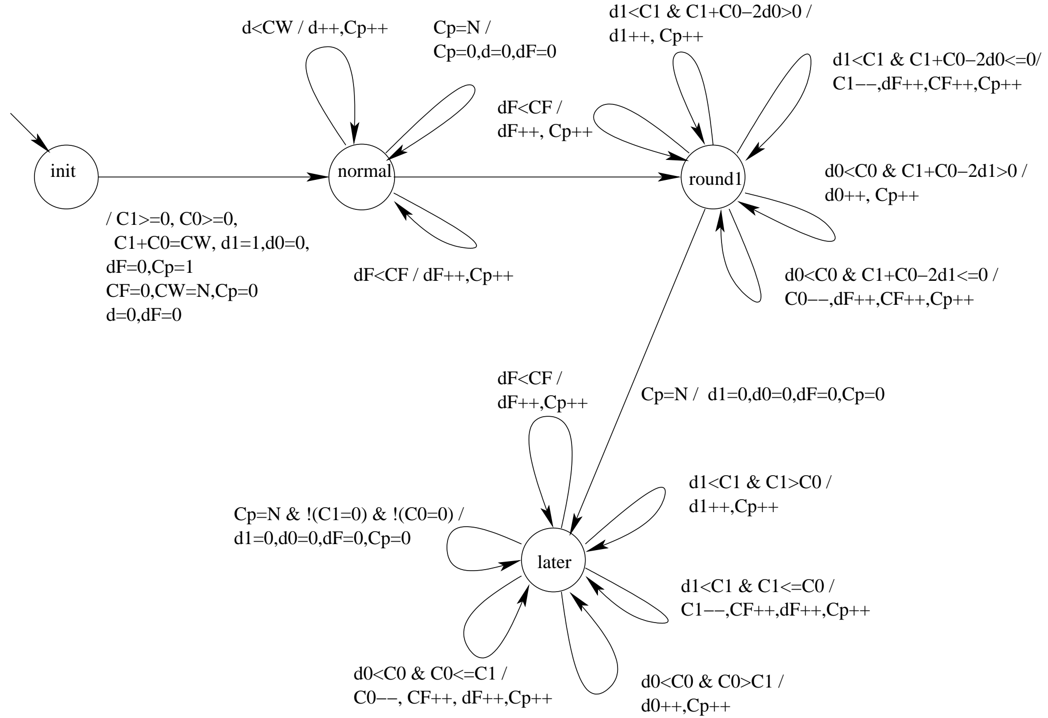


FIG. 5.10: Système à compteurs affine pour le TTP, 1 défaillance et N stations

Résultats. FAST vérifie *automatiquement* que la propriété P_1 est satisfaite. Le calcul des configurations accessibles requiert seulement des cycles de longueur 1. Le calcul prend 1880 secondes et 73 Mo. L'automate binaire associé a 27.932 nœuds.

Analyse incrémentale. Le temps de calcul peut être amélioré en utilisant un guidage des calculs plus fin, via les scripts de stratégie. En effet, la procédure de calcul de FAST ne prend pas en compte la structure bien particulière du graphe de contrôle du système à compteurs. Comme l'on ne peut pas revenir en arriée dans les locations, on peut restreindre le calcul de point fixe aux transitions bouclant sur la location `normal`, atteindre un point fixe r_1, tirer la transition pour aller à `round1`, et réitérer le processus. Cette décomposition est explicitée à la figure 5.11. En utilisant cette méthode, le temps de calcul passe à 203 secondes et 55 Mo.

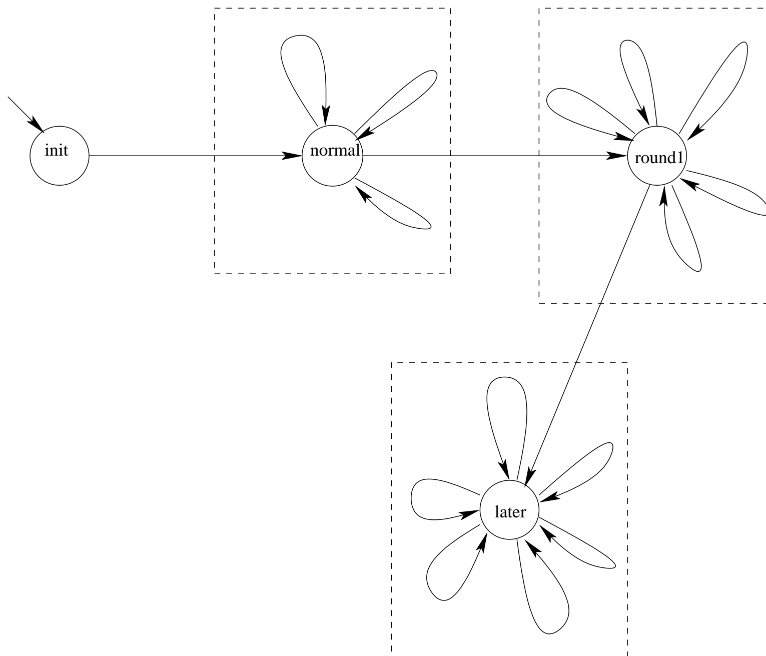


FIG. 5.11: Décomposition modulaire du protocole TTP

5.6.4 Vérification pour deux défaillances

Le système à compteurs affine pour deux défaillances est présenté à la figure 5.12. Le comportement normal du protocole n'est pas représenté sur la figure. Le système est beaucoup plus large que pour une défaillance, avec 18 variables et des fonctions impliquant jusqu'à 14 variables. Il y a trois cliques différentes C_{00}, C_{10}, C_{11}. L'absence de clique (P_2) s'exprime par :

(P_2) $location = later \wedge C_{p2} = N \Rightarrow$

$(C_{11} \neq 0 \wedge C_{10} = C_{00} = 0) \vee (C_{11} = C_{00} = 0 \wedge C_{10} \neq 0) \vee (C_{11} = C_{10} = 0 \wedge C_{00} \neq 0)$

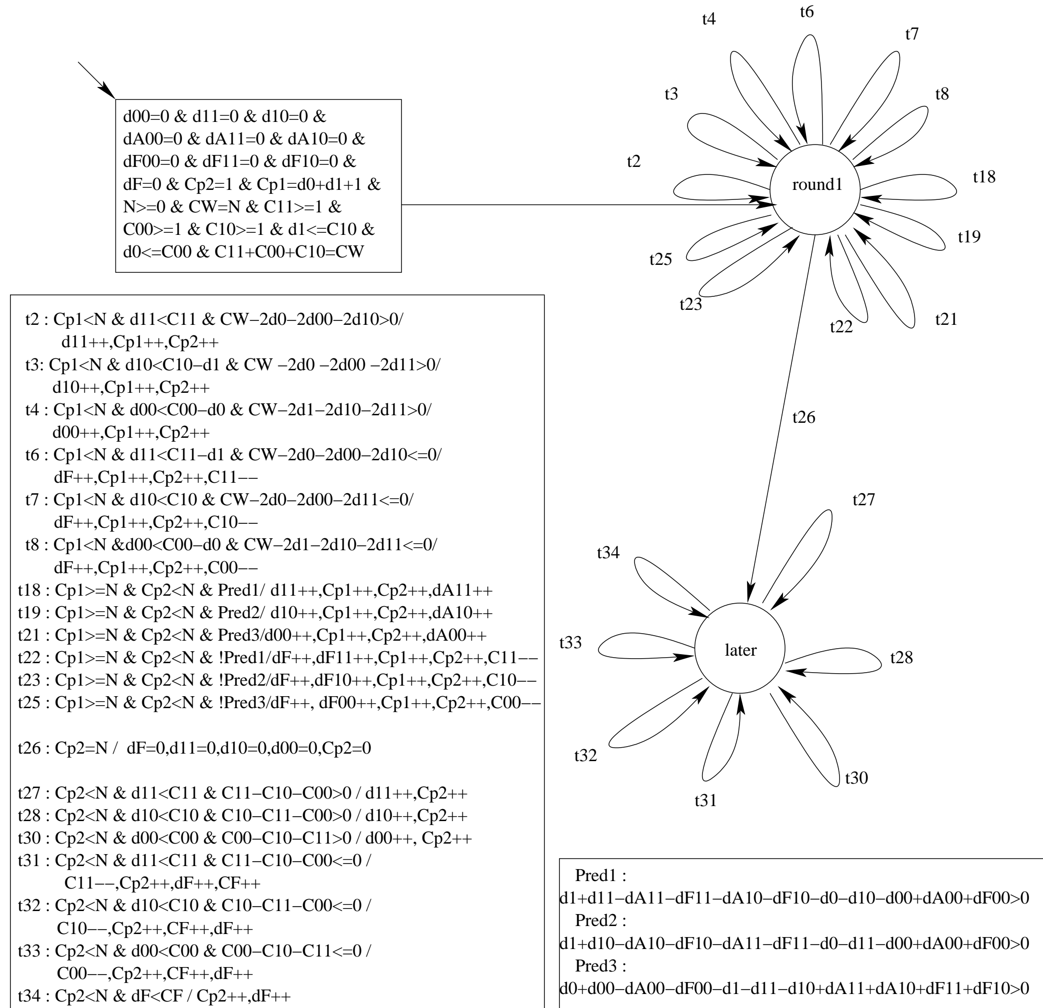


FIG. 5.12: Système à compteurs affine pour le TTP, 2 défaillances et N stations

Nécessité de l'accélération convexe. Lors du calcul de la clôture transitive des transitions, la taille des automates binaires manipulés dépasse les capacités de FAST. Heureusement, toutes les transitions à l'exception de t_{26} sont des translations convexes et t_{26} n'a pas besoin d'être accélérée puisqu'elle ne fait partie d'aucun cycle. Aussi pouvons nous utiliser l'accélération convexe sur le TTP. Avec l'accélération convexe, les clôtures transitives des transitions sont effectivement calculées.

Vérification pour un nombre fixé de stations. En fixant la valeur de N à 5, l'ensemble d'accessibilité est calculé et P_2 est vérifiée. Le calcul prend 900 secondes et 588 Mo. L'automate binaire calculé a 5.684 nœuds. Le calcul fonctionne encore pour $N = 10$, mais échoue (dépassement de capacité des automates) pour $N = 15$.

Vérification pour un nombre arbitraire de stations. Les automates intermédiaires calculés dépassent la taille maximale de 2^{24} nœuds. Plutôt que de calculer l'ensemble d'accessibilité, nous en calculons une surapproximation et vérifions que P_2 est valide sur la surapproximation, ce qui prouve la correction du protocole. Nous utilisons les modifications suivantes :

Réduction du nombre de variables : nous utilisons des invariants comme $C_W = C_{11} + C_{10} + C_{00}$ et $C_{p1} = C_{p2} + d_0 + d_1$ pour enlever C_W et C_{p1}. Nous enlevons aussi la variable dF qui n'est pas utilisée à la location `round1` et vaut $N - C_{11} - C_{00} - C_{10}$ à la location `later`.

Surapproximation du comportement en simplifiant les gardes : nous supprimons certaines clauses et testons sur le cas fini si la surapproximation obtenue permet de vérifier P_2. Si oui, nous conservons cette modification. Ainsi les gardes de t_2, t_3, t_4, t_6, t_7, t_8, t_{18}, t_{19}, t_{21}, t_{22}, t_{23}, t_{25} sont simplifiées. Ces simplications permettent à la fois de réduire la structure des automates binaires calculés mais aussi de supprimer de nouvelles variables, par exemple d_0 et d_1 disparaissent tandis que dA_{11}, dA_{10} et dA_{00} sont retirées de la location `round1`.

Stratégie de calcul modulaire : pour améliorer les performances nous décomposons le calcul en plusieurs étapes comme décrit pour une défaillance à la figure 5.11.

L'abstraction du système est présentée à la figure 5.13. De cette façon, FAST permet de vérifier que le TTP est valide pour un nombre arbitraire de stations et 2 défaillances.

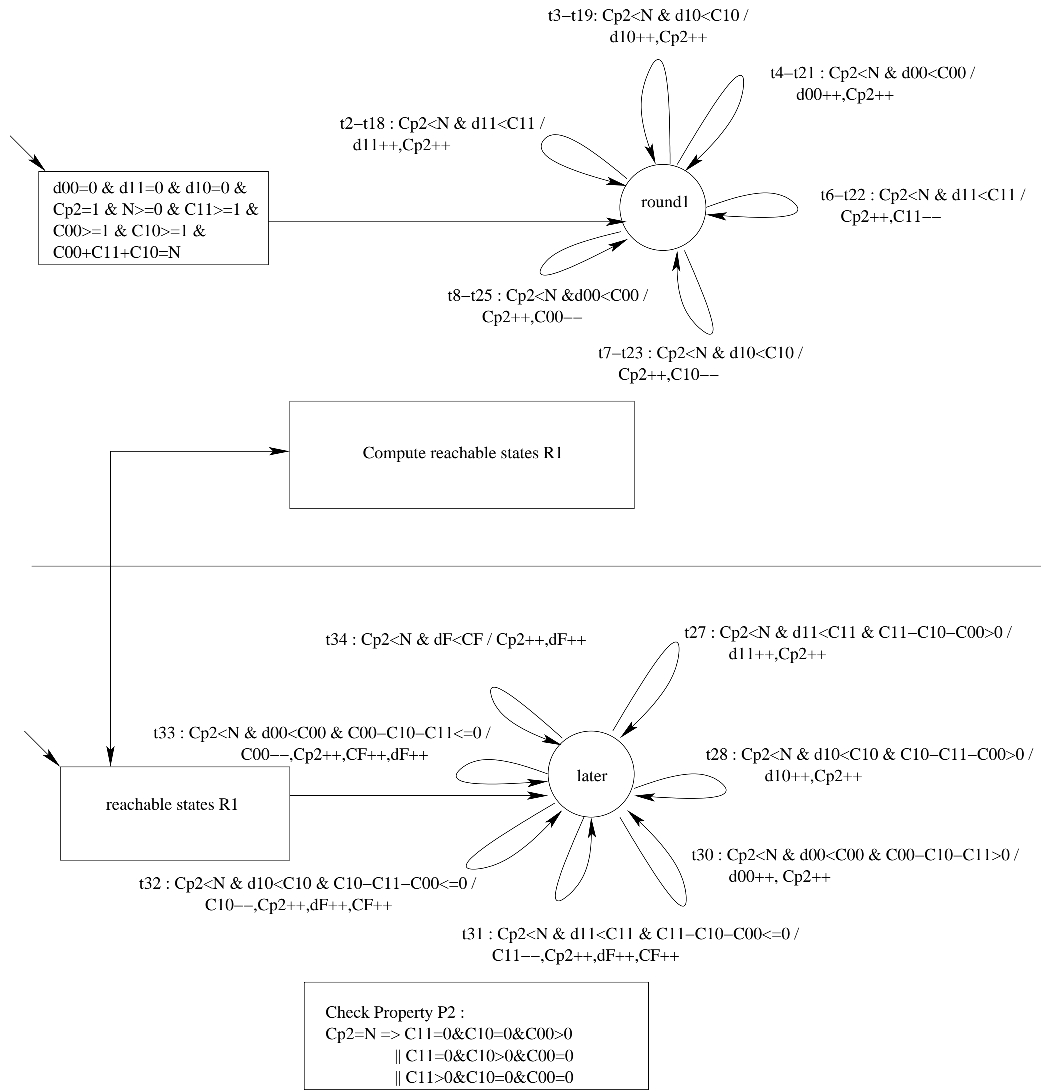


FIG. 5.13: Surapproximation du protocole TTP, 2 défaillances et N stations

5.6.5 Résultats

Les résultats sont résumés dans la figure 5.14. L'accélération convexe fonctionne toujours mieux que l'accélération générale, à la fois en temps et en espace. Pour le calcul de l'abstraction, la quantité de mémoire utilisée est la même dans chaque cas. Comme les fonctions sont plus simples le pic d'utilisation mémoire ne représente pas le calcul d'une clôture transitive comme dans les autres tests, mais le calcul de l'ensemble d'accessibilité qui est indépendant de l'accélération employée.

	accélération générale		accélération convexe		
	temps secondes	mémoire Mo	temps secondes	mémoire Mo	nombre de nœuds
1 faute, N stations	1.880	73	1.200	63	27.932
2 fautes, 5 stations	↑↑↑ (*)	↑↑↑ (*)	892	588	5.684
2 fautes, 10 stations	↑↑↑ (*)	↑↑↑ (*)	24.365	588	273.427
2 fautes, 15 stations	↑↑↑ (*)	↑↑↑ (*)	↑↑↑	↑↑↑	↑↑↑
2 fautes, N stations	↑↑↑ (*)	↑↑↑ (*)	↑↑↑	↑↑↑	↑↑↑
2 fautes, N stations (abstraction)	420	200	350	200	11.036

(*) Les accélérations saturent la mémoire et ne sont pas calculées.

FIG. 5.14: Benchmark pour la vérification du TTP avec FAST

5.6.6 Vérification avec ALV, LASH et TREX

Nous décrivons ici nos tentatives de vérification du TTP avec d'autres outils que FAST.

Avec ALV le calcul d'accessibilité ne termine pas pour pour une défaillance et N quelconque. La méthode de vérification décrite dans [BM02] est en fait semi-automatique : ALV vérifie des invariants fournis par les auteurs. Ces invariants assurent la correction, mais l'ensemble d'accessibilité n'est jamais calculé.

LASH permet de calculer l'ensemble d'accessibilité pour une défaillance et N quelconque, puisque seules les boucles doivent être accélérées. Cependant pour deux défaillances les accélérations saturent la mémoire et la vérification

échoue.

Enfin le TTP est trop complexe pour être modélisé par les systèmes à compteurs (restreints) de TReX.

5.7 Vérification du protocole CES

Cette section s'intéresse à la vérification du protocole *Capability Exchange Signalling (CES)* de Philips pour l'échange de contenus multimedia. Dans [LB02] Billington et Liu prouvent "à la main" la structure de l'ensemble d'accessibilité. FAST vérifie automatiquement cette propriété.

5.7.1 Le protocole CES

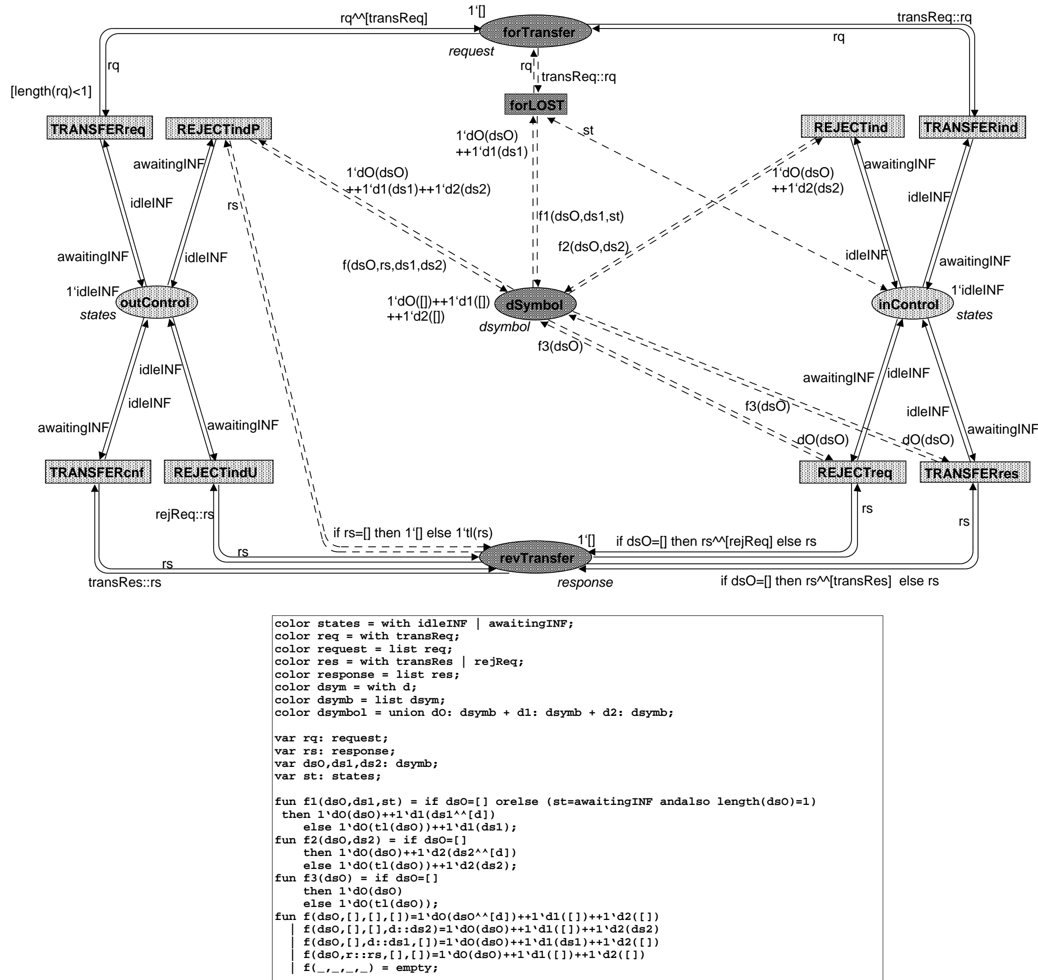
Le but du protocole est d'assurer une transmission complète de messages malgré la présence de canaux de communications à pertes. Le protocole est paramétré par la taille des canaux.

La figure 5.15 montre un réseau de Petri coloré modélisant le CES. Ce réseau est tiré de [LB02]. Les réseaux de Petri Colorés (RPC) sont une extension des réseaux de Petri dans laquelle les jetons transportent des données typées, appelées couleurs (exprimées en ML ici). Les jetons de couleur consommés et produits par les transitions sont définis par des fonctions [7], attachées aux transitions (exprimées en ML ici).

Les places *OutControl* et *InControl* contiennent toujours un seul jeton pouvant prendre deux valeurs. La place *forTransfer* contient une file, les messages correspondants sont tous les mêmes. La place *revTransfer* contient une file avec deux sortes de messages : *transRes* et *rejReq*. Enfin la place *dsymbol* a trois sortes de jetons : *dsymbol0*, *ddsymbol1* et *dsymbol2*. La transition *forLOST* simule la perte de messages. Remarquons que la longueur de la file *forTransfer* est bornée à 1 dans ce modèle, car les RPC n'ont pas de représentations symboliques pour l'infini. Nous voulons enlever cette limitation dans notre modélisation pour avoir une analyse paramétrique.

Billington et Liu étudient dans [LB02] la forme de l'ensemble d'accessibilité. Ils prouvent qu'il contient 12 configurations pour une file de taille 1, et qu'à chaque incrémentation de la file, 4 nouvelles configurations sont ajou-

[7] Évidement les RPC peuvent simuler des machines de Turing.




```
color states = with idleINF | awaitingINF;
color req = with transReq;
color request = list req;
color res = with transRes | rejReq;
color response = list res;
color dsym = with d;
color dsymb = list dsym;
color dsymbol = union dO: dsymb + d1: dsymb + d2: dsymb;

var rq: request;
var rs: response;
var dsO,ds1,ds2: dsymb;
var st: states;

fun f1(dsO,ds1,st) = if dsO=[] orelse (st=awaitingINF andalso length(dsO)=1)
 then 1`dO(dsO)++1`d1(ds1^^[d])
    else 1`dO(tl(dsO))++1`d1(ds1);
fun f2(dsO,ds2) = if dsO=[]
    then 1`dO(dsO)++1`d2(ds2^^[d])
    else 1`dO(tl(dsO))++1`d2(ds2);
fun f3(dsO) = if dsO=[]
    then 1`dO(dsO)
    else 1`dO(tl(dsO));
fun f(dsO,[],[],[])=1`dO(dsO^^[d])++1`d1([])++1`d2([])
  | f(dsO,[],[],d::ds2)=1`dO(dsO)++1`d1([])++1`d2(ds2)
  | f(dsO,[],d::ds1,[])=1`dO(dsO)++1`d1(ds1)++1`d2([])
  | f(dsO,r::rs,[],[])=1`dO(dsO)++1`d1([])++1`d2([])
  | f(_,_,_,_) = empty;
```

Fig. 5.15: Un RPC modélisant le protocole CES

tées. Ces configurations additionnelles sont aussi entièrement caractérisées [LB02, table 2, page 28].

Intérêts du CES. Cet exemple est intéressant pour trois raisons. Tout d'abord le CES est naturellement un système à files et FAST n'est donc pas a priori l'outil le plus adapté. Nous montrons comment modéliser le CES en système à compteurs. FAST permet de vérifier que les hypothèses de modélisation sont correctes. Ensuite la propriété de sûreté à vérifier est assez originale et nécessite une description quantitative de l'ensemble d'accessibilité. Ainsi non seulement les approches traditionnelles sur les files à base d'expressions régulières (`qdd`, `sre`) ne permettent pas d'exprimer une telle propriété, mais de plus cette propriété nécessite un formalisme très expressif puisque l'ensemble décrit n'est pas convexe. Enfin, le calcul de l'ensemble d'accessibilité du système à compteurs associé nécessite des cycles de longueur 2.

5.7.2 Simulation par système à compteurs

Nous commençons tout d'abord par transformer le RPC en système à compteurs. Les files sont gérées comme suit. Comme la place *forTransfer* contient une file ayant un alphabet réduit à un seul message, la file est directement représentée par un compteur *forTrans*. La seconde file *revTransfer* est plus complexe, puisque son alphabet a deux messages *transRes* et *rejReq*. En suivant [LB02] nous faisons l'hypothèse que les deux messages ne coexistent pas dans la file. Ainsi la file peut être modélisée par deux compteurs *revTrans* et *rejTrans*, un par type de message.

La figure 5.16 montre un extrait de la description au format FAST du système obtenu. Seules les parties intéressantes ont été gardées. La variable *bufl̲g* représente la taille maximale de la file *forTransfer*. Le système a une seule location `marking`. Il y a plus de variables que de places dans le RPC, à cause des couleurs complexes. Les places *OutControl* et *InControl* peuvent être représentée toutes les deux par une seule variable prenant les valeurs 0 ou 1. La place *dsymbol* est éclatée en trois variables (variables *d0*, *d1* et *d2*), selon les couleurs considérées. Les transitions *TRANSFERreq*, *TRANSFERcnf*, *REJECTIndU* et *TRANSFERInd* sont écrites facilement. Les autres transitions doivent être tranformées en plusieurs transitions élémentaires, selon les différentes couleurs. Par exemple la transition *REJECTreq* est scindée en 2, selon que la valeur de *ds0* est nulle ou non.

```
model CES
 var outControl, inControl, forTrans, revTrans, rejTrans, d0, d1, d2, buflg;
 states marking;

 transition TRANSFERreq :=
  from   := marking;
  to     := marking;
  guard  := outControl=0 && forTrans<buflg;
  action := outControl'=1, forTrans'=forTrans+1;
 ;
...
 transition REJECTreq1 :=
  from   := marking;
  to     := marking;
  guard  := inControl=1 && d0=0;
  action := inControl'=0, rejTrans'=rejTrans+1;
 ;

 transition REJECTreq2 :=
  from   := marking;
  to     := marking;
  guard  := inControl=1 && !d0=0;
  action := inControl'=0, d0'=d0-1;
 ;
```

FIG. 5.16: Description au format FAST du CES (extrait)

5.7.3 Correction de la modélisation des files

Tout d'abord on vérifie la correction des hypothèses de modélisation des files. Pour assurer que deux types de messages ne peuvent coexister dans la file *revTransfer*, nous montrons que toutes les configurations atteignables vérifient : soit la valeur de la variable *revTrans* est nulle, soit la valeur de la variable *rejTrans* est nulle. Cela s'exprime par la déclaration d'une région *bad* :

```
Region bad := (!(revTrans)=0) && (!(rejTrans)=0);
```

La région *bad* est ensuite intersectée avec l'ensemble d'accessibilité. La résultat est la région vide, donc l'hypothèse de modélisation est correcte.

5.7.4 Vérification avec FAST

Dans [LB02], les auteurs prouvent que l'ensemble d'accessibilité a 12 configurations pour une file de taille 1, et à chaque incrémentation de la taille de la file, 4 nouvelles configurations sont ajoutées. Ces configurations additionnelles sont caractérisées. Une région est déclarée pour chacune des 12 configurations initiales du système (*mark1* à *mark12*). Les configurations additionnelles sont définies par les formules de [LB02], table 2, page 287

(*mark1n* à *mark4n*). L'ensemble d'accessibilité est donc supposé être égal à l'union de toutes ses régions (région *fullOG*). Grâce au script suivant, nous prouvons automatiquement ce résultat.

```
strategy forward
Transitions t := TRANSFERreq, TRANSFERcnf, REJECTindU, TRANSFERind, ...;

Region mark1 := outControl=0 && inControl=0 && forTrans=0 && revTrans=0
    && rejTrans=0 && d0=0 && d1=0 && d2=0 && state=marking;
...
Region mark12 := outControl=1 && inControl=1 && forTrans=0 && revTrans=0
    && rejTrans=0 && d0=1 && d1=1 && d2=0 && state=marking;

Region marks1n := forTrans<=buflg && d0=forTrans-1 && outControl=1 &&
    inControl=0 && revTrans=0 && rejTrans=0 && d1=0 && d2=0 &&
    state=marking;
...
Region marks4n := forTrans<=buflg && d0=forTrans+1 && outControl=0 &&
    inControl=1 && revTrans=0 && rejTrans=0 && d1=0 && d2=0 &&
    state=marking;

Region fullOG := mark1 || mark2 || ...  || marks1n || ... || marks4n;

Region reach := post*(mark1 && buflg>0, t, 2);

if (eqSet(fullOG && buflg>0,reach)) then
    print("fullOG OK");
endif
```

L'ensemble d'accessibilité est calculé en utilisant des cycles de longueur 2 (région *reach*), et l'égalité des 2 régions est testée. Le résultat est positif, l'ensemble d'accessibilité a bien la forme voulue. Remarquons que ce calcul est paramétré par la taille de la file, aussi ce résultat décrit exactement le nombre de configurations accessibles pour chaque valeur de *buflg*. Fast termine en moins de 30 secondes.

5.7.5 Résultats

Fast a été utilisé avec succès pour vérifier une propriété quantitative complexe sur un système à files lossy. Une étape importante est le passage des files lossy aux compteurs. Cette modélisation repose sur l'hypothèse que les files ne contiennent pas plus d'un type de message simultanément. Fast prouve également la validité de cette hypothèse.

5.7.6 Vérification avec Alv et Lash

Nous essayons de vérifier le protocole CES avec les outils Alv et Lash. L'outil Alv ne termine pas en moins de 1200 secondes. Lash ne termine pas en moins de 1200 secondes (avec uniquement l'accélération de boucles) et

termine si on lui indique les cycles de longueur 2 à accélérer. Nous n'avons pas expérimenté TReX sur cette étude de cas.

5.8 Conclusion

Nous avons présenté dans ce chapitre Fast, un outil permettant de vérifier des propriétés d'accessibilité sur des systèmes à compteurs affines. Fast respecte le cadre de l'accélération plate, et implante les structures de données, algorithmes et procédures définis aux chapitres 2 et 4.

Les études de cas sont très encourageantes, Fast parvenant à calculer l'ensemble d'accessibilité de près de 75% des systèmes analysés. Ces performances sont très supérieures à celles des outils similaires.

De plus la technologie derrière Fast est très souple et peut s'adapter facilement à d'autres types de calculs. Ainsi Fast a été comparé, sur du calcul de co-accessibilité et de couverture de systèmes à compteurs monotones, à des outils spécialisés. Les performances de Fast se sont avérées très correctes, égalant même parfois celles des outils spécialisés.

Enfin de nombreux systèmes ont été vérifiés par Fast. L'accélération plate pour les translations convexes, décrite au chapitre 4, a été implantée et a permis la première vérification automatique du protocole TTP, un protocole complexe de reprise sur panne de systèmes embarqués. Fast a aussi été utilisé avec succès pour vérifier automatiquement des propriétés quantitatives sur un système à files lossy, le protocole multimedia CES. Les techniques habituelles à base d'expressions régulières sont inadaptées à de telles propriétés.

Les expériences menées avec Fast montrent que le cadre de l'accélération plate est pertinent pour la vérification de systèmes à compteurs. De plus les comparaisons entre Fast, Lash et Alv montrent clairement que l'accélération plate améliore grandement la terminaison en pratique du calcul de l'ensemble d'accessibilité des systèmes à compteurs.

Troisième partie

Vérification de systèmes à pointeurs

Chapitre 6

Cadre symbolique des systèmes à pointeurs

6.1 Introduction

Nous adaptons dans ce chapitre les techniques de model checking symboliques à la vérification des programmes à allocation dynamique. Plus exactement nous définissons un cadre symbolique adapté à la vérification quantitative de systèmes à pointeurs, et une abstraction conservant les propriétés de fuite mémoire et violation mémoire.

6.1.1 Contexte

Une cause courante de bogues informatiques est la gestion explicite par le programmeur de la mémoire allouée au programme. Ces mécanismes *d'allocation dynamique* sont disponibles depuis longtemps dans les langages de programmation impératifs comme C, et les langages fonctionnels intégrent souvent des mécanismes similaires pour des raisons d'efficacité (par exemple OCaml).

Le tas mémoire. Le tas mémoire est la zone de la mémoire où sont stockées les cellules mémoires allouées dynamiquement par le programme. En première approximation, le tas mémoire est une collection de cellules mémoires contenant des données et des adresses (du tas mémoire). On peut accéder à une adresse de deux manières. Soit on lit directement la valeur de l'adresse dans une *variable de pointeur* du programme. Dans ce cas, on dit que la variable pointe sur l'adresse. Lire la valeur d'un pointeur et accéder à la cellule mémoire correspondante s'appelle *déréférencer* le pointeur. Soit on

calcule la valeur de l'adresse par des décalages à partir d'une adresse donnée (*arithmétique de pointeurs*[1]).

Le programmeur gère le tas mémoire au moyen de deux instructions principales **new** et **free**. Typiquement, **new** réserve au programme une cellule mémoire dans le tas mémoire, et retourne l'adresse de cette cellule. On dit que **new** *alloue* de la mémoire. **free** est l'opération inverse, et permet de libérer, ou *désallouer*, une cellule mémoire.

Les adresses du tas mémoire sont de trois types : (1) les adresses de cellules mémoires valides, c'est-à-dire les adresses de cellules mémoire allouées explicitement par le programme et non désalouées depuis ; (2) les adresses de cellules mémoire invalides. Un programme manipule des adresses invalides lorsque des variables de pointeurs ne sont pas initialisées, ou la cellule correspondante a été désallouée, ou le programme utilise abusivement l'arithmétique de pointeurs. *Il n'y a pas de primitive pour tester si une adresse est invalide* ; (3) une adresse spéciale (**null** dans le langage C par exemple) ne pointant sur rien ; cette adresse spéciale *peut être testée.*

Propriétés des programmes à allocation dynamique. Nous étudions les propriétés de sûreté listées ci-après.

Fuite mémoire : il y a fuite mémoire lorsqu'une zone mémoire allouée n'est plus pointée par aucune variable de pointeur. Cette mémoire est alors perdue et ne peut plus être ni utilisée ni libérée.

Violation mémoire : il y a violation mémoire lorsque le programme déréférence une adresse invalide. Dans ce cas le programme peut accéder aux données d'autres programmes.

Les propriétés qualitatives : expriment des propriétés sur la forme du tas mémoire, par exemple que la variable `x` pointe bien sur une liste (de taille arbitraire) dont la queue pointe sur **null**, ou que le programme retourne bien une liste circulaire.

Les propriétés quantitatives : donnent des informations sur le nombre de cellules mémoires vérifiant certains prédicats. Par exemple, un programme `reverse` retourne-t-il une liste dont la longueur est égale à celle de la liste d'entrée ?

[1]Typiquement les tableaux en C

6.1.2 Cadre des systèmes à pointeurs

Dans ce chapitre nous définissons un cadre symbolique (cf. chapitre 2) pour les programmes à allocation dynamique. Ce cadre symbolique est composé d'un système : les systèmes à pointeurs ; d'un domaine d'interprétation : les graphes mémoires ; et d'une représentation symbolique : les états mémoires symboliques. Ces résultats proviennent de [BFN04].

Systèmes à pointeurs. Les systèmes à pointeurs (définition 6.2.2) sont des programmes séquentiels non récursifs manipulant des variables de pointeurs. De plus nous faisons les deux abstractions suivantes : nous ne nous intéressons pas aux données contenues dans les cellules mémoires mais seulement aux adresses, et encore nous ne considérons pas la valeur exacte de ces adresses, mais uniquement le fait qu'une cellule mémoire pointe sur une autre[2].

Graphes mémoires. Nous imposons comme restriction sur le tas mémoire qu'une cellule mémoire ait au plus un pointeur vers une autre cellule mémoire. Nous nous restreignons donc à des programmes manipulant des listes simplement chaînées *(listes)*. Ces listes peuvent être linéaires ou circulaires, et plusieurs listes peuvent partager des éléments. Ceci nous permet de représenter le tas mémoire par des graphes mémoires (définition 6.2.1).

Model-checking symbolique. Nous donnons une sémantique opérationnelle aux systèmes à pointeurs, et nous définissons les propriétés de violation mémoire et de fuite mémoire. On ne peut pas décider si l'ensemble d'accessibilité d'un système à pointeurs contient une fuite mémoire ou une violation mémoire [BFN04]. En revanche on peut décider si un graphe mémoire est une fuite mémoire, et si une action sur un graphe mémoire produit une violation mémoire. Nous définissons un cadre symbolique pour les systèmes à pointeurs, les états mémoire symboliques (ems, définition 6.3.3). La violation mémoire et la fuite mémoire sont décidables sur un état mémoire symbolique. Les états mémoire symboliques admettent une forme minimale. Enfin nous proposons une abstraction des états mémoire symboliques ($\mathsf{ems}^\sharp$, définition 6.4.2) correcte vis-à-vis des propriétés de fuite mémoire et de violation mémoire. Les états mémoire symboliques abstraits ont de meilleures propriétés algorithmiques que les états mémoire symboliques.

[2]Nous nous interdisons donc l'arithmétique de pointeurs.

6.1.3 Approches existantes

Un état de l'art est donné dans [Hin01]. Parmi toutes les approches envisagées, nous décrivons les trois suivantes qui se rapprochent le plus de nos travaux. Les modèles de programmes considérés sont tous similaires aux systèmes à pointeurs.

PALE [MS01] traduit le programme et l'état de la mémoire dans une logique décidable, puis décide des propriétés qualitatives. PALE travaille sur des programmes annotés. L'approche n'est pas limitée aux listes chaînées, mais les différentes structures manipulées (listes, arbres, ...) ne peuvent pas partager d'éléments et l'utilisateur doit annoter le programme avec des invariants de boucle.

TVLA [LARSW00, LAS00, SRW02]. Des prédicats indiquent typiquement que le contenu d'une cellule mémoire pointe sur une autre. Les prédicats sont ensuites évalués dans une logique trivaluée, "oui, non, peut-être". Cette abstraction assure le calcul effectif d'une surapproximation de l'ensemble d'accessibilité, mais la vérification perd en précision. L'approche peut traiter des structures plus générales que des listes, mais l'utilisateur doit fournir des prédicats contrôlant la finesse de l'abstraction.

Model checking régulier. Dans [BHMV05] le tas mémoire est abstrait par des expressions régulières. La méthode suit celle du raffinement d'abstraction : on essaie de décider la propriété sur une abstraction, et si l'abstraction ne permet pas de conclure on recommence en l'affinant. L'approche est automatique et s'applique aux propriétés qualitatives. Les expérimentations fournies par les auteurs sont très encourageantes.

La figure 6.1 résume les différentes approches.

	PALE	TVLA	mod. ch. régulier	mod. ch. symbolique
tas mémoire	arbres ou listes sans partage	arbres ou listes	listes	listes
prop. qualitatives prop. quantitatives	oui non	oui non	oui non	oui oui
automatisation	annotation	prédicats	autom.	autom.
terminaison	oui	oui(*)	non	non

(*) le calcul peut retourner "la propriété est peut-être vérifiée".

FIG. 6.1: Différentes méthodes de vérification de systèmes à pointeurs.

Le problème de la vérification de propriétés quantitatives de listes chaînées est abordé dans [BI05a, BI05b]. Bozga et Iosif montrent que la satisfaisabilité de *la logique d'alias à compteurs* [BI05b] est décidable sur les ems [BI05a].

6.2 Systèmes à pointeurs

Tout d'abord nous modélisons les programmes à allocation dynamique et le tas mémoire, puis nous les relions par une sémantique opérationnelle. En suivant l'approche du chapitre 2, nous définissons ainsi la famille des systèmes à pointeurs. Pour cela nous devons définir une interprétation $I = (\Phi, D, [\![\cdot]\!])$. Enfin nous exprimons les propriétés de violation mémoire et fuite mémoire sur les sytèmes à pointeurs et montrons qu'elles sont indécidables.

Notation. Dans la suite nous considérons donné un ensemble fini de variables Var, qui seront utilisées comme variables de pointeurs.

6.2.1 Domaine d'interprétation : graphes mémoire

Compte tenu des abstractions exposées dans l'introduction de ce chapitre, le tas mémoire peut être vu comme un graphe orienté, avec pour nœuds les cellules mémoire valides allouées jusqu'à maintenant. Un arc du nœud n_1 au nœud n_2 indique que la cellule mémoire n_1 contient l'adresse de la cellule mémoire n_2. Chaque nœud est aussi étiqueté par l'ensemble (éventuellement vide) des variables de pointeur qui contiennent son adresse. Enfin chaque graphe mémoire contient trois nœuds spéciaux. Le nœud `null` dénote le pointeur **null**, le nœud `puits` correspond aux adresses invalides (abstraites en un seul nœud) et le nœud `erreur` représente les expressions de pointeur invalides. Typiquement, les successeurs d'une cellule mémoire désallouée ou de **null**. Chaque nœud a exactement un successeur, puisqu'une cellule mémoire pointe soit sur une cellule valide (un autre nœud), soit sur **null** (`null`), soit sur une cellule invalide (`puits`).

Définition 6.2.1 (Graphe mémoire)**.** Un *graphe mémoire* G est un triplet $G = (N, s, var)$ tel que

- N est un ensemble fini de nœuds,
- $s : N \longrightarrow N$ est la fonction successeur,
- $var : N \longrightarrow 2^{Var}$ est une fonction injective (d'étiquetage) ;

où N contient trois nœuds spéciaux `null`, `puits` et `erreur` tels que $s($`null`$) = s($`puits`$) = s($`erreur`$) =$ `erreur` et $var($`erreur`$) = \emptyset$.

Comme il y a exactement un successeur par nœud, les arcs sont définis exactement par les $(n, s(n))$ pour $n \in N$. Dès lors l'expression $var(n)$ donne sans ambiguité l'étiquette de l'arc $(n, s(n))$.

Exemple. Voici un exemple de graphe mémoire. Par convention nous ne notons pas les étiquettes vides.

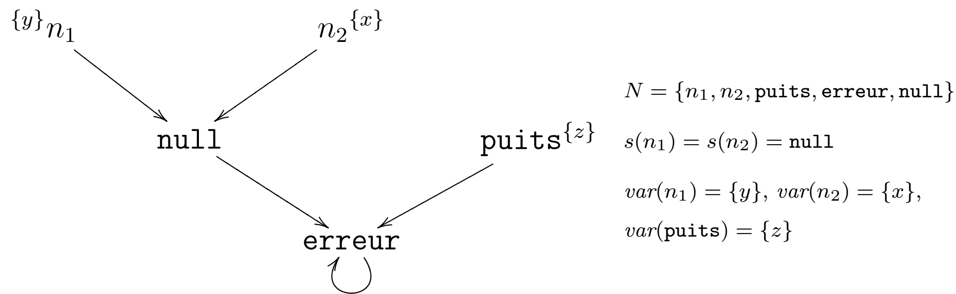


FIG. 6.2: Exemple de graphe mémoire.

Isomorphisme. Deux graphes mémoire G_1 et G_2, notés $G_1 = (N_1, s_1, var_1)$ et $G_2 = (N_2, s_2, var_2)$, sont dits *isomorphes* si les graphes sous-jacents (N_1, s_1) et (N_2, s_2) sont eux-mêmes isomorphes et que pour tout $(n_1, n_2) \in N_1 \times N_2$, si n_1 et n_2 sont équivalents via l'isomorphisme, alors $var_1(n_1) = var_2(n_2)$. L'isomorphisme de graphes mémoire est décidable en temps exponentiel. Pour la classe des graphes mémoire étanches introduits à la section 6.4.2, l'isomorphisme est décidable en temps linéaire.

Nœuds accessibles et fuites mémoire. Un nœud n est dit *accessible* si et seulement si il existe un chemin de destination n dont l'origine est étiquetée par au moins une variable de pointeur. Un nœud non accessible est dit *mort*, car la cellule mémoire correspondante ne pourra jamais plus être utilisée. Un graphe mémoire ayant au moins un nœud mort est une *fuite mémoire.*

Notation. Nous noterons $\mathcal{G}$ l'ensemble de tous les graphes mémoire sur Var.

6.2.2 Actions des systèmes à pointeurs

Nous définissons maintenant l'ensemble des formules Φ_p qui étiquetent les transitions d'un système à pointeurs. Φ_p est défini par la grammaire suivante :

$$
\begin{aligned}
\varphi \quad &::= \quad \text{garde? action} \mid \text{garde? } \mathbf{nop} \\
\text{action} \quad &::= \quad x := \mathbf{null} \mid x := y \mid x := y.s \mid x.s := y \mid x := \mathbf{new} \mid \mathbf{free}(x) \\
\text{garde} \quad &::= \quad \mathit{True} \mid \mathit{IsNull}(x) \mid \neg \mathit{IsNull}(x)
\end{aligned}
$$

Dans [BFN04], les systèmes à pointeurs sont définis avec un ensemble d'actions plus grand qu'on peut retrouver en composant les six actions définies ici.

Exemple 6.2.1. La figure 6.3 présente une fonction `reverse` écrite en C, et le système à pointeurs correspondant. La figure 6.4 donne une représentation graphique de ce système. Nous ne nous intéressons qu'au comportement interne à la fonction, aussi nous ne modélisons pas la désallocation des variables locales à la sortie du programme. Cet exemple est tiré de [LAS00].

```
/* list.h */
typedef struct node
{
  struct node* n;
  int data;
}* List;
```

```
/* reverse.c */
#include ``list.h''
List reverse(List x) {
  List y,t;
  y = NULL;
  while (x!=NULL) {
    t=y;
    y=x;
    x=x->n;
    y->n=t;
    t=NULL;
  }
  return y;
}
```

$(1,\ \mathit{True},\ y := \mathbf{null},\ 2),$
$(2,\ \neg \mathit{IsNull}(x),\ t := y,\ 3),$
$(3,\ \mathit{True},\ y := x,\ 4),$
$(4,\ \mathit{True},\ x := x.s,\ 5),$
$(5,\ \mathit{True},\ y.s := t,\ 6),$
$(6,\ \mathit{True},\ t := \mathbf{null},\ 2),$
$(2,\ \mathit{IsNull}(x),\ \mathbf{nop},\ 7)$

FIG. 6.3: Un programme C qui renverse une liste et le système à pointeurs correspondant.

6.2.3 Sémantique opérationnelle

Le comportement décrit ci-dessous est très proche du comportement machine. C'est typiquement le jeu de primitives offert par des langages comme C. Certains langages fournissent des primitives de plus haut niveau. Par exemple en Java, les pointeurs non initialisés sont automatiquement mis à **null**, *et* **free** *n'est pas disponible. La sémantique que nous proposons dans cette section peut facilement être adaptée à ces variantes.*

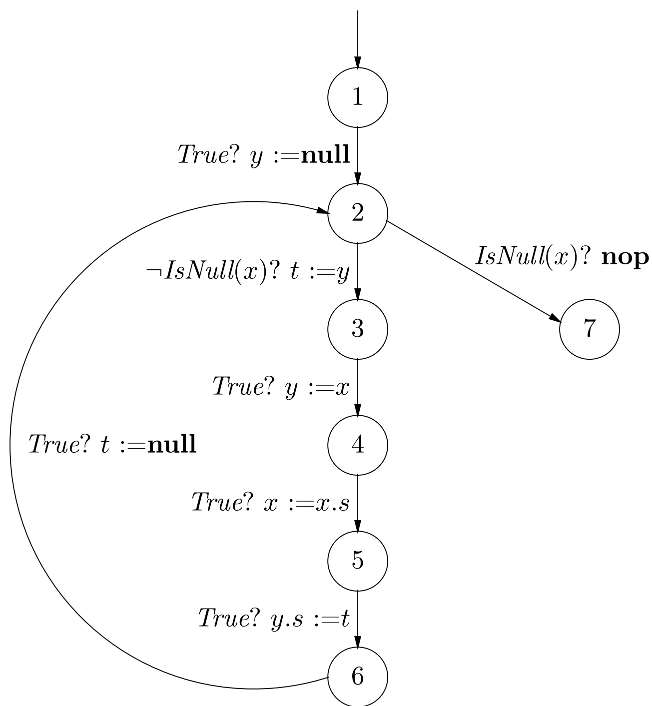


FIG. 6.4: Une représentation graphique du système à pointeurs `reverse`

La sémantique des systèmes à pointeurs sur des graphes mémoire est définie en termes d'ajouts et/ou de retraits de nœuds, d'arcs et d'étiquettes. Par exemple `x :=`**new** crée un nouveau nœud étiqueté par x relié au nœud spécial `puits` (mémoire non allouée), tandis que x est retiré de l'étiquette à laquelle il appartenait avant l'action. Ceci est le comportement standard pour C. On peut adapter facilement d'autres langages. Par exemple en Java les pointeurs sont initialisés à **null**, aussi le nouveau nœud serait relié à `null`.

free`(x)` teste si le nœud n pointé par `x` est le nœud `puits`. Si c'est le cas, il y a une violation mémoire. Si `x` pointe sur **null**, il ne se passe rien [3]. Sinon n est retiré. Les nœuds qui pointaient sur n pointent alors sur `puits`. Les variables qui étiquetaient n étiquettent maintenant `puits`.

Plus formellement, nous définissons pour chaque formule $a \in \Phi_p$ une fonction *partielle* f_a des graphes mémoire vers les graphes mémoire, qui décrit l'effet de a. Nous définissons $(N', s', var') = f_a(N, s, var)$ par :

1. Si a est de la forme x :=**null** alors :
   - $N' = N$
   - s' = s
   - $$\begin{aligned} var'(\texttt{null}) &= var(\texttt{null}) \cup \{x\} \\ var'(n_0) &= var(n_0) \setminus \{x\} \\ var'(n) &= var(n) \text{ si } n \notin \{n_0, \texttt{null}\} \end{aligned}$$

   où $n_0 \in N$ est l'unique nœud tel que $x \in var(n_0)$.

2. Si a est de la forme $x := y$ alors :
   - $N' = N$
   - $s' = s$
   - $$\begin{aligned} var'(m_0) &= var(m_0) \setminus \{x\} \\ var'(n_0) &= var(n_0) \cup \{x\} \\ var'(n) &= var(n) \text{ si } n \notin \{m_0, n_0\} \end{aligned}$$

   où $m_0 \in N$ (respectivement $n_0 \in N$) est l'unique nœud tel que $x \in var(m_0)$ (respectivement $y \in var(n_0)$).

3. Si a est de la forme $x := y.s$ alors :
   - $N' = N$
   - $s' = s$

[3] C'est le comportement spécifié en Unix, mais il n'est pas toujours respecté.

- $var'(m_0) = var(m_0)\backslash\{x\}$
  $var'(s(n_0)) = var(s(n_0)) \cup \{x\}$
  $var'(n) = var(n)$ si $n \notin \{m_0, s(n_0)\}$

  où $m_0 \in N$ (respectivement $n_0 \in N$) est l'unique nœud tel que $x \in var(m_0)$ (respectivement $y \in var(n_0)$).

4. Si a est de la forme $x.s := y$ alors :
   - $N' = N$
   - $s'(m_0) = n_0$
     $s'(n) = s(n)$ si $n \neq m_0$
   - $var' = var$

   où $m_0 \in N$ (respectivement $n_0 \in N$) est l'unique nœud tel que $x \in var(m_0)$ (respectivement $y \in var(n_0)$).

5. Si a est de la forme $x :=$**new** alors :
   - $N' = N \cup \{*\}$ où $* \notin N$
   - $s'(*) =$ `puits`
     $s'(n) = s(n)$ si $n \in N$
   - $var'(*) = \{x\}$
     $var'(n_0) = var(n_0)\backslash\{x\}$
     $var'(n) = var(n)$ si $n \notin \{*, n_0\}$

   où $n_0 \in N$ est l'unique nœud tel que $x \in var(n_0)$.

6. Si a est de la forme **free**(x) alors :
   - $N' = N\backslash\{n_0\}$,
   - $s'(n) = \begin{cases} \texttt{puits} \text{ si } s(n) = n_0 \\ s(n) \text{ sinon} \end{cases}$
   - $var'(\texttt{puits}) = var(\texttt{puits}) \cup var(n_0)$
     $var'(n) = var(n)$ si $n \neq$ `puits`

   où $n_0 \in N$ est l'unique nœud tel que $x \in var(n_0)$.

Cas de la violation mémoire. Si le graphe mémoire résultant (N', s', var') ne satisfait pas les conditions de la définition 6.2.1 (étiquetage vide du nœud `erreur`) alors f_a est indéfini. Par construction, cela correspond à une violation mémoire. Nous ajoutons le graphe mémoire spécial segf pour dénoter

une violation mémoire. Par définition, segf n'est isomorphe à aucun autre graphe mémoire, et n'admet aucun successeur.

Nous définissons maintenant la relation $G \xrightarrow{g?\ a} G'$ par
- $G = (N, s, var)$,
- $G' = f_a(G)$ si $f_a(G)$ existe ; $G' =$ segf sinon.
- et
  - g est de la forme *True*, ou
  - g est de la forme $IsNull(x)$ et $x \in var(\texttt{null})$, ou
  - g est de la forme $\neg IsNull(x)$ et $x \notin var(\texttt{null})$,

6.2.4 Famille des systèmes à pointeurs

En suivant notre démarche exposée au chapitre 2, nous définissons la famille des systèmes à pointeurs.

Définition 6.2.2 (Famille des systèmes à pointeurs)**.** La famille des systèmes à pointeurs est construite sur l'interprétation $I_p = (\Phi_p, \mathcal{G} \cup \{\text{segf}\}, [\![\cdot]\!])$ où $\mathcal{G}$ est l'ensemble des graphes mémoire, Φ_p est l'ensemble de formules défini à la section 6.2.2 et $[\![\cdot]\!]$ est la concrétisation définie à la section 6.2.3.

6.2.5 Propriétés des systèmes à pointeurs

Nous définissons formellement les propriétés de violation mémoire et fuite mémoire pour un système à pointeurs et montrons que ces propriétés sont indécidables.

PV Un système à pointeurs $(S, (q_0, G_0))$ est *sans violation mémoire* si pour toute location q de S, $(q, \text{segf}) \notin \mathsf{post}^*((q_0, G_0))$.

PF Un système à pointeurs $(S, (q_0, G_0))$ est *sans fuite mémoire* si pour toute location q de S, il n'existe pas de $(q, G) \in \mathsf{post}^*((q_0, G_0))$ tel que G est une fuite mémoire.

Théorème 6.2.1. *Les propriétés* **PV** *et* **PF** *sont indécidables pour des systèmes à pointeurs avec au moins trois variables.*

Démonstration. L'indécidabilité vient de la réduction du problème de l'accessibilité dans les systèmes à 2 compteurs (+1, -1, test à 0). Un système à 2 compteurs se simule par un système à 3 pointeurs comme suit : chaque compteur c_i est représenté par une liste dont la longueur est la valeur de c_i. Les deux premiers pointeurs pointent chacun sur une tête de liste différente, et le troisième pointeur sert à ajouter des éléments aux listes. Les opérations sur les compteurs (test à 0, incrémentation, décrémentation) sont simulées par des opérations sur les listes (test à **null**, ajouter une cellule, enlever

une cellule). Dire que 0 est accessible à partir d'une configuration initiale donnée est équivalent à l'accessibilité de segf dans le système à pointeurs. Cette propriété est la négation de **PV**. De même décider l'accessibilité de 0 se réduit à décider **PF**. L'accessibilité de 0 est un problème indécidable pour les machines à 2 compteurs. Plus en détail :

PV L'instruction d'incrément $q : c_i := c_i + 1;$ goto q' est simulée par :

$$\begin{array}{llllll} \{ & (q, & \textit{True?} & y := x_i, & q_1), & \\ & (q_1, & \textit{True?} & x_i := \textbf{new}, & q_2), & \\ & (q_2, & \textit{True?} & x_i.s := y, & q') & \} \end{array}$$

L'instruction de décrément gardé q : if $c_i = 0$ then goto q' else $c_i := c_i - 1;$ goto q'' est simulée par :

$$\begin{array}{llllll} \{ & (q, & \textit{IsNull}(x_i)? & \textbf{nop}, & q'), & \\ & (q, & \textit{IsNull}(x_i)? & y := \textbf{null}, & q_1), & \\ & (q_1, & \textit{True?} & y := y.s, & q'), & \\ & (q, & \neg\textit{IsNull}(x_i)? & y := x_i, & q_2), & \\ & (q_2, & \textit{True?} & x_i := x_i.s, & q_3), & \\ & (q_3, & \textit{True?} & \textbf{free}(y), & q'') & \} \end{array}$$

On montre facilement qu'il y a violation mémoire dans le système à pointeurs si et seulement si le système à 2 compteurs atteint une configuration où l'un des compteurs vaut 0. Cette propriété est indécidable, aussi **PV** est-elle indécidable.

PF L'incrément est simulé comme pour **PV**. Le décrément gardé est simulé par :

$$\begin{array}{llllll} \{ & (q, & \textit{IsNull}(x_i)? & y := \textbf{new}, & q_1), & \\ & (q_1, & \textit{True?} & y := \textbf{null}, & q'), & \\ & (q, & \neg\textit{IsNull}(x_i)? & y := x_i, & q_2), & \\ & (q_2, & \textit{True?} & x_i := x_i.s, & q_3), & \\ & (q_3, & \textit{True?} & \textbf{free}(y), & q'') & \} \end{array}$$

On montre qu'il y a fuite mémoire dans le système à pointeurs si et seulement si le système à 2 compteurs atteint une configuration où l'un des compteurs vaut 0. Cette propriété est indécidable, aussi **PF** est-elle indécidable. □

Pour des systèmes à 1 pointeur il est impossible d'ajouter des cellules mémoire sans provoquer de fuite mémoire. Le nombre de cellules utiles est

donc borné, et on se ramène à un système fini. Les propriétés **PF** et **PV** se décident alors en énumérant les graphes mémoire accessibles. La question suivante est ouverte.

Problème ouvert 6.2.1. Est-ce que **PV** et **PF** sont décidables pour des systèmes à 2 pointeurs ?

6.3 Cadre symbolique : états mémoire symboliques

Nous définissons dans cette section les *États mémoire symboliques.* Nous montrons qu'ils forment un cadre symbolique pour les systèmes à pointeurs et qu'ils admettent une forme minimale.

Notation. Dans la suite nous supposerons donné un ensemble infini dénombrable VarCptSet de variables de compteurs. Nous supposerons que $VarCptSet \cap Var = \emptyset$.

6.3.1 États mémoire symboliques

Nous voulons représenter des ensembles infinis de graphes mémoire. Nous définissons à cet effet les États Mémoire Symboliques (ems). Nous avons tout d'abord besoin de la notion de graphe mémoire étiqueté.

Définition 6.3.1 (Graphe mémoire étiqueté)**.** Un graphe mémoire étiqueté G_c est un quadruplet $G_c = (N, s, var, c)$ tel que
- (N, s, var) est un graphe mémoire,
- $c : N \to VarCptSet$ est une fonction injective.

La fonction c étiquète les arcs avec une variable de compteurs. Comme un nœud n définit un unique arc $(n, s(n))$, la fonction c porte sur les nœuds.

La figure 6.5 présente un exemple de graphe mémoire étiqueté $G_c = (N, s, var, c)$:

Les états mémoire sont des graphes mémoire étiquetés par des compteurs, avec une valuation pour chaque compteur. Le compteur $c(n)$ associé à l'arc $(n, s(n))$ représente une liste de tête n, de queue $s(n)$ et dont la longueur est la valeur de $c(n)$. Un état mémoire est une représentation compacte d'un graphe mémoire.

Définition 6.3.2 (État mémoire)**.** Un état mémoire M est un quintuplet $M = (N, s, var, c, \nu)$ tel que :

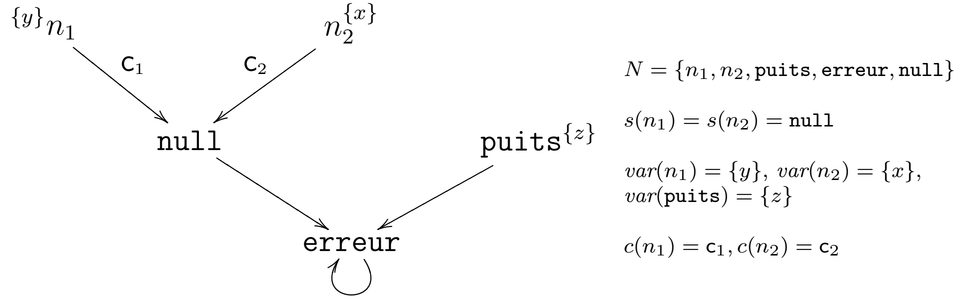


FIG. 6.5: Un exemple de graphe mémoire étiqueté.

- (N, s, var, c) est un graphe mémoire étiqueté,
- la fonction $\nu : c(N) \to \mathbb{N}\backslash\{0\}$ vérifie : $\nu(c(\texttt{null})) = 1$, $\nu(c(\texttt{puits})) = 1$, et $\nu(c(\texttt{erreur})) = 1$.

La concrétisation $[\![M]\!]$ d'un état mémoire $M = (G, c, \nu)$ est le graphe mémoire G tel que pour chaque nœud n, l'arc de sortie est remplacé par une liste linéaire de $\nu(c(n))$ arcs et $(\nu(c(n)) - 1)$ nœuds intermédiaires, tels que n est la tête et $s(n)$ la queue.

La figure 6.6 présente un exemple d'état mémoire $M = (N, s, var, c, \nu)$ et la figure 6.7 donne la concrétisation de l'état mémoire M.

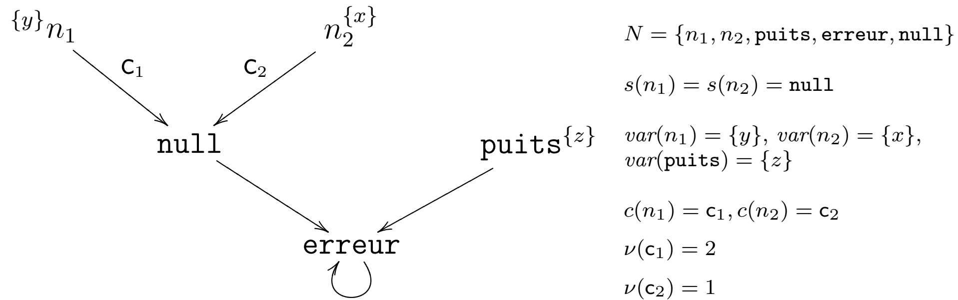


FIG. 6.6: Exemple d'état mémoire

Nous pouvons maintenant définir un état mémoire symbolique (ems) comme la donnée d'un graphe non étiqueté et d'une formule de Presburger. La formule de Presburger donne les valuations possibles des compteurs, et donc les différents états mémoire représentés par l'ems. Comme chaque état mémoire représente un graphe mémoire, un ems permet donc de représenter un ensemble infini de graphes mémoire.

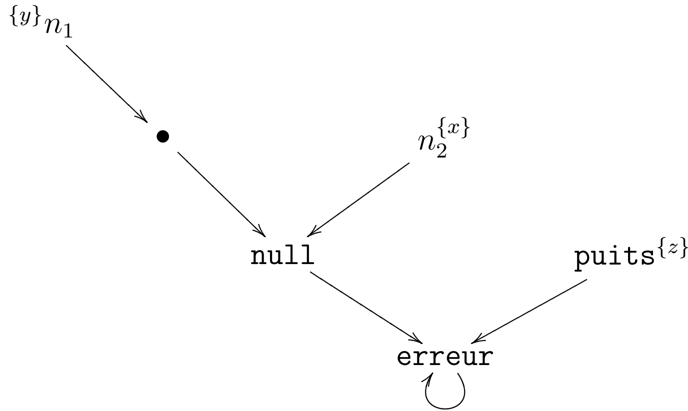


FIG. 6.7: Concrétisation de l'état mémoire de la figure 6.6

Définition 6.3.3 (État mémoire symbolique)**.** Un état mémoire symbolique est soit

1. un 5-uplet $ems = (N, s, var, c, \phi)$ tel que
   - (N, s, var, c) est un graphe mémoire étiqueté,
   - ϕ est une formule de Presburger dont les variables libres sont exactement $c(N)$,
   - pour tout vecteur $\nu \in \mathbb{N}^{|c(N)|}$, si ν est solution de ϕ, alors (N, s, var, c, ν) est un état mémoire, ou
2. segf, ou
3. $\perp$.

Remarque 6.3.1. Soit ν une solution de ϕ, (N, s, var, c, ν) est alors un état mémoire. Ceci implique que toutes les composantes de ν, et donc toutes les variables libres de ϕ, sont différentes de 0. Donc pour tout $k \in c(N)$, la formule $(\phi \Rightarrow k > 0)$ est valide.

Concrétisation d'états mémoire symboliques. Un état mémoire $M = (G, c, \nu)$ est *intégré* dans un état mémoire symbolique $ems = (G, c, \phi)$ si la valuation de $c(N)$ induite par ν est solution de ϕ. La concrétisation d'un état mémoire symbolique $ems = (G, c, \phi)$ est l'ensemble (éventuellement infini) des graphes mémoire obtenus par concrétisation des états mémoire intégrés dans ems. Ainsi, si ϕ n'est pas satisfaisable on a $[\![(G, c, \phi)]\!] = \emptyset$. Les concrétisations des ems spéciaux segf et $\perp$ sont $[\![\text{segf}]\!] = \text{segf}$ et $[\![\perp]\!] = \emptyset$.

Nous utiliserons comme représentation symbolique des unions finies d'états mémoire symboliques, notées $EMS = \{ems_1, \ldots, ems_n\}$. La concrétisation

d'une union finie d'états mémoire symboliques est l'union de leurs concrétisations.

Un ensemble $X \subseteq \mathcal{G}$ est dit *ems-définissable* s'il existe un ems EMS tel que $X = [\![EMS]\!]$.

Notation. Dans la suite nous écrivons état mémoire symbolique pour désigner une union finie d'états mémoire symboliques $EMS = \{ems_1, \ldots, ems_n\}, n \geq 2$, et état mémoire symbolique atomique ems pour un singleton.

Taille d'un ems. Nous définissons la taille d'un état mémoire symbolique atomique $ems = (N, s, var, c, \phi)$, notée $|ems|$, par $|ems| = |N|$. Par convention, $|\bot| = |\mathsf{segf}| = 1$. Pour un ems quelconque $EMS = \{ems_1, \ldots, ems_n\}$, $|EMS| = \sum_i |ems_i|$.

6.3.2 Isomorphisme d'états mémoire symboliques

Nous définissons l'isomorphisme d'états mémoire symboliques. Les états mémoire symbolique segf et $\bot$ ne sont isomorphes qu'à eux-mêmes. On dira que deux ems atomiques $ems_1 = (G_1, c_1, \phi_1)$ et $ems_2 = (G_2, c_2, \phi_2)$ sont *isomorphes* si G_1 et G_2 sont isomorphes, et qu'après renommage $\phi_1 \Leftrightarrow \phi_2$. Deux ems $EMS' = \{ems_1, \ldots, ems_m\}$ et $EMS' = \{ems'_1, \ldots, ems'_n\}$ sont isomorphes si : pour tout $ems \in EMS$ il existe $ems' \in EMS'$ tel que ems et ems' sont isomorphes, et vice-versa. Deux ems isomorphes ont même concrétisation.

Théorème 6.3.1. *Deux ems isomorphes ont même concrétisation.*

Démonstration. On raisonne sur les ems atomiques. L'extension aux ems non atomiques se déduit facilement. Les cas de segf et $\bot$ sont directs. Nous considérons deux ems atomiques $ems_1 = (G_1, c_1, \phi_1)$ et $ems_2 = (G_2, c_2, \phi_2)$ isomorphes. Comme ils sont isomorphes, G_1 et G_2 sont eux-mêmes isomorphes. Nous rappelons que pour $i \in \{1, 2\}$, la concrétisation de ems_i est l'ensemble des concrétisations des états mémoire $M = (G_i, c_i, \nu_i)$ tels que $\nu_i \in \phi_i$. Si les deux graphes G_1 et G_2 sont isomorphes et qu'après renommage des compteurs correspondants $\phi_1 \Leftrightarrow \phi_2$, alors après renommage tout $\nu_1 \in \phi_1$ est aussi une solution de ϕ_2, et vice versa. On a donc bien $[\![ems_1]\!] = [\![ems_2]\!]$. □

6.3.3 Graphes mémoires minimaux

Nous définissons dans cette section une notion de graphe mémoire minimal. Nous notons par *deg*(n) le nombre d'arcs entrants du nœud n dans un graphe mémoire G.

Définition 6.3.4 (Nœud central)**.** Soit G un graphe mémoire. Un nœud n de G est un *nœud central* si *deg*(n) $\neq 1$ ou *var*(n) $\neq \emptyset$ ou n est un des nœuds spéciaux `erreur`, `puits`, `null`.

Un nœud non central est dit *nœud intermédiaire*. Soit un graphe mémoire $G = (N, s, Var)$. Le retrait d'un nœud intermédiaire $n \in N$ de G produit le graphe mémoire $G' = (N', s', Var)$ tel que $N' = N \backslash \{n\}$, $s'(n') = s(n')$ si $s'(n') \neq n$, et $s'(n') = s(n)$ sinon. Intuitivement le nœud n' qui pointait sur n pointe maintenant sur $s(n)$.

On dit que le graphe mémoire G_1 est plus petit que le graphe mémoire G_2, noté $G_1 \preceq G_2$, si il existe G'_2 tel que G'_2 est obtenu à partir de G_2 en enlevant des nœuds intermédiaires et G_1 est isomorphe à G'_2.

La relation ainsi définie est un pré-ordre partiel sur l'ensemble $\mathcal{G}$ des graphes mémoire. Cette relation n'est pas un ordre car elle n'est pas antisymétrique : deux graphes mémoire isomorphes vérifient $G_1 \preceq G_2$ et $G_2 \preceq G_1$, mais pourtant G_1 est différent de G_2. Nous notons $G_1 \prec G_2$ si $G_1 \preceq G_2$ et G_1 et G_2 ne sont pas isomorphes.

Nous définissons maintenant la notion de graphe mémoire minimal.

Définition 6.3.5 (graphe mémoire minimal)**.** Un graphe mémoire G est minimal s'il n'existe pas de $G' \in \mathcal{G}$ tel que $G' \prec G$.

On remarque qu'un graphe mémoire est minimal si et seulement si tous ses nœuds sont centraux. Pour tout graphe mémoire G, on peut calculer un graphe mémoire minimal G' tel que $G' \preceq G$.

Théorème 6.3.2. *Pour tout graphe mémoire $G \in \mathcal{G}$, il existe un unique graphe mémoire minimal (à un isomorphisme près), noté $min(G)$, tel que $min(G) \preceq G$.*

Démonstration. À isomorphisme près on peut considérer que $min(G)$ a un ensemble de nœuds inclus dans celui de G. Comme $min(G)$ est minimal il n'a aucun nœud intermédiaire. De plus comme $min(G) \preceq G$, les nœuds centraux de $min(G)$ sont exactement ceux de G. Donc le seul graphe mémoire $min(G)$ possible est le graphe

mémoire G duquel on a retiré tous les nœuds intermédiaires. Ceci montre l'unicité et l'existence puisque le graphe mémoire G privé de ses nœuds intermédiaires convient effectivement. □

Enfin, on peut remarquer qu'il existe un nombre infini de graphes mémoire minimaux non isomorphes. Il suffit de considérer une famille de graphes mémoire en structure de peigne de taille croissante.

Lemme 6.3.1. Il existe un nombre infini de graphes mémoire minimaux non isomorphes.

6.3.4 Forme atomique minimale

Différents ems peuvent avoir la même concrétisation. Nous montrons malgré tout que tout ems admet une forme minimale unique à *isomorphisme* près. Pour un ems $ems = (G, c, \phi)$ tel que ϕ soit satisfaisable, le choix naturel pour une forme minimale est d'avoir le graphe mémoire le plus simple, c'est à dire avec le moins de nœuds possible. Ceci correspond exactement à un graphe mémoire minimal.

Définition 6.3.6. Les ems atomiques minimaux sont définis comme suit :
- (G, c, ϕ) est minimal si G est minimal et ϕ est satisfaisable ;
- $\perp$ est minimal ;
- segf est minimal.

On démontre tout d'abord un résultat sur les ems atomiques quelconques, qui nous servira à établir le corollaire 6.3.1 sur les ems atomiques minimaux. Par définition de la concrétisation des ems on déduit que si $ems = (G, c, \phi)$ et $G' \in [\![ems]\!]$, alors $G \preceq G'$. On montre aussi le lemme suivant.

Lemme 6.3.2. Soient deux ems atomiques $ems = (G, c, \phi)$ et $ems = (G', c', \phi')$. Si $[\![ems]\!] \cap [\![ems']\!] \neq \emptyset$ alors $min(G)$ et $min(G')$ sont isomorphes.

Démonstration. Il existe $G'' \in \mathcal{G}$ tel que $G'' \in [\![ems]\!]$ et $G'' \in [\![ems']\!]$. D'après la remarque précédente, on déduit que $G \preceq G''$ et $G' \preceq G''$. Or le graphe mémoire $min(G)$ est minimal et vérifie $min(G) \preceq G''$. Par définition de $min(G'')$, on déduit que $min(G)$ est isomorphe à $min(G'')$. De la même manière on montre que $min(G')$ est isomorphe à $min(G'')$. On en conclut que $min(G)$ et $min(G')$ sont isomorphes. □

En prenant le cas de ems atomiques minimaux, et en remarquant que dans ce cas $G_1 = min(G_1)$ et $G_2 = min(G_2)$, on déduit le corollaire suivant. Si deux ems atomiques minimaux ont une intersection non vide, alors nécessairement leurs graphes mémoire sont isomorphes.

Corollaire 6.3.1. Soient deux ems atomiques minimaux de la forme $ems_1 = (G_1, c_1, \phi_1)$ et $ems_2 = (G_2, c_2, \phi_2)$. Si $[\![ems_1]\!] \cap [\![ems_2]\!] \neq \emptyset$ alors G_1 est isomorphe à G_2.

6.3.5 Forme minimale

Nous voulons maintenant définir une *forme minimale* pour les ems quelconques. Intuitivement on impose que chaque ems atomique soit minimal, et que chacun corresponde à un ensemble distinct de graphes mémoire.

Définition 6.3.7 (Forme minimale)**.** Un ems $EMS = \{ems_1, \ldots, ems_n\}$ est minimal si

- chaque ems_i est minimal, et
- pour tout $i \neq j$, le graphe mémoire de ems_i et le graphe mémoire de ems_j ne sont pas isomorphes, et
- si $n > 1$ alors $\perp \notin EMS$.

La définition 6.3.7 correspond bien à une forme minimale. On a en effet le résultat suivant.

Théorème 6.3.3. *Soit EMS un ems minimal et EMS' un ems tel que $[\![EMS]\!] = [\![EMS']\!]$. Alors $|EMS| \leq |EMS'|$.*

Démonstration. On raisonne tout d'abord sur les ems atomiques. On suppose que $EMS = \{ems\}$ et $EMS' = \{ems'\}$, où $ems = (G, c, \phi)$ et $ems' = (G', c', \phi')$. Comme l'intersection de leurs concrétisations est non nulle, on sait que $min(G)$ est isomorphe à $min(G')$ (lemme 6.3.2). Or G est minimal, donc $min(G) = G$. On en déduit que $G \preceq G'$ et donc que $|G| \leq |G'|$. Ceci implique que $|ems| \leq |ems'|$. On raisonne maintenant sur des ems quelconques $EMS = \{ems_1, \ldots, ems_n\}$ et $EMS' = \{ems'_1, \ldots, ems'_m\}$. Pour chaque $i \leq n$ il existe au moins un $i' \leq m$ tel que $[\![ems_i]\!] \cap [\![ems'_{i'}]\!] \neq \emptyset$ (car ϕ est satisfaisable et les concrétisations sont égales). Donc $|ems_i| \leq |ems'_{i'}|$. De plus tous les $ems'_{i'}$ sont distincts. En effet comme EMS est minimal, les graphes mémoire des ems_i sont non isomorphes, donc les graphes mémoire des $ems'_{i'}$ sont distincts et donc les $ems'_{i'}$ sont distincts. On note par $ems'_{i'}$ les ems atomiques de EMS' ayant une intersection non nulle avec un ems atomique de EMS et ems'_k les autres ems atomiques de EMS'. Alors on a $|EMS'| = \sum_{i'} |ems'_{i'}| + \sum_k |ems'_k|$ et donc $|EMS'| \geq \sum_i |ems_i| + \sum_k |ems'_k|$. On en conclut que $|EMS'| \geq |EMS|$. □

6.3.6 Minimisation effective

La question de l'existence et de l'unicité d'une telle forme minimale se pose naturellement. Les deux théorèmes suivants clarifient la question. Nous

montrons tout d'abord que si deux ems ont même concrétisation ils sont isomorphes.

Théorème 6.3.4. *Soit deux* ems *minimaux* EMS_1 *et* EMS_2 *tels que* $[\![EMS_1]\!] = [\![EMS_2]\!]$. *Alors* EMS_1 *et* EMS_2 *sont isomorphes.*

Démonstration. Tout d'abord supposons que les deux ems soient atomiques, notés $ems_1 = (G_1, c_1, \phi_1)$ et $ems_2 = (G_2, c_2, \phi_2)$. Alors en utilisant le corollaire 6.3.1, il vient que G_1 et G_2 sont isomorphes. À chaque solution ν de ϕ_1 correspond un élément $G \in [\![ems_1]\!]$, ainsi qu'une valuation ν' des variables libres de ϕ_2. Comme $[\![ems_1]\!] = [\![ems_2]\!]$ on en déduit que ν' est solution de ϕ_2. Donc après renommage adéquat, $\phi_1 \Rightarrow \phi_2$ est valide. Par un raisonnement symétrique on déduit que $\phi_1 \Leftrightarrow \phi_2$, et ems_1 et ems_2 sont isomorphes. Nous montrons maintenant le cas général. Soit $EMS_1 = \{ems_{1,1}, \ldots, ems_{1,m}\}$ et $EMS_2 = \{ems_{2,1}, \ldots, ems_{2,n}\}$, tels que $[\![EMS_1]\!] = [\![EMS_2]\!]$. On ne considère pas les ems spéciaux $\perp$ et segf simples à traiter. Soit $i \leq m$. Alors $[\![ems_{1,i}]\!] \subseteq [\![EMS_2]\!]$ et il existe j tel que $ems_{1,i} \cap ems_{2,j} \neq \emptyset$. Donc d'après le corollaire 6.3.1, $G_{1,i}$ et $G_{2,j}$ sont isomorphes. Or EMS_2 est canonique, donc il n'existe pas d'autre $ems_{2,k}$ tel que $G_{1,i}$ et $G_{2,k}$ sont isomorphes et donc pour tout $k \neq j$, $ems_{1,i} \cap ems_{2,k} = \emptyset$. Donc on a nécessairement $[\![ems_{1,i}]\!] \subseteq [\![ems_{2,j}]\!]$. En raisonnant symétriquement on prouve que $[\![ems_{2,j}]\!] \subseteq [\![ems_{1,i}]\!]$ pour les mêmes i et j. Donc $[\![ems_{1,i}]\!] = [\![ems_{2,j}]\!]$, et on se ramène au cas atomique. Le raisonnement a été fait sur tous les ems_i et les ems_j donc EMS_1 et EMS_2 sont bien isomorphes. □

Nous montrons maintenant que pour un ems donné, on peut effectivement calculer un ems minimal ayant même concrétisation.

Théorème 6.3.5 (Minimisation)**.** *Pour tout* ems EMS, *on peut construire effectivement un* ems *minimal* EMS' *tel que* $[\![EMS]\!] = [\![EMS']\!]$.

Démonstration. Soit ems un ems atomique. Nous construisons d'abord un ems $ems' = (G, c, \phi)$ ayant même concrétisation et dont tous les nœuds sont centraux. Cela se fait en enlevant les nœuds intermédiaires successifs (de degré 1 et tels que $var(n) = \emptyset$), en ajoutant un nouveau compteur k étiquetant le nouvel arc et en transformant ϕ en $\exists(k_1, \ldots, k_m), \phi \wedge k = \sum_i k_i$. Cette formule assure bien que $k > 0$, puisque ϕ assure que chacun des k_i est supérieur à 0. Nous itérons l'opération sur ems_1 pour une autre suite consécutive de nœuds non centraux, et ainsi de suite jusqu'à obtenir ems' minimal. Pour un ems général $EMS = \{ems_1, ems_2, \ldots, ems_n\}$, nous faisons l'opération sur chaque ems atomique et obtenons $EMS' = \{ems'_1, ems'_2, \ldots, ems'_n\}$. Si dans EMS' il existe des ems atomiques $ems'_i = (G_i, c_i, \phi_i)$ et $ems'_j = (G_j, c_j, \phi_j)$ tels que G_i et G_j sont isomorphes, ils sont retirés et remplacés par l'ems atomique $ems_{i,j} = (G_i, c_i, \phi_i \vee \phi_j)$ (et les compteurs sont bien > 0). De plus s'il existe ϕ_i insatisfaisable, alors l'ems ems_i est remplacé par $\perp$. Enfin si il reste $\perp$ et au moins un

autre ems différent de $\perp$, alors $\perp$ est retiré. On vérifie que l'ems EMS'' ainsi obtenu est minimal et $[\![EMS'']\!] = [\![EMS]\!]$. □

6.3.7 Union, intersection, complément

Nous nous intéressons aux opérations ensemblistes sur les ensembles ems-définissables. Les opérations sont décrites sur des ems minimaux ce qui en facilite l'écriture. Pour un ems quelconque, il suffit de le minimiser avant d'appliquer les opérations.

Théorème 6.3.6. *Les ensembles ems-définissables sont effectivement clos par union et intersection.*

Démonstration. **Le cas de l'union** est le plus simple puisqu'on manipule des listes finies d'ems atomiques. On peut cependant améliorer l'opération : quand on rajoute un ems atomique $ems = (G, c, \phi)$ à un ems $EMS = \{ems_1, \ldots, ems_n\}$, on vérifie si G est isomorphe à l'un des G_i. Dans ce cas on retire ems_i de EMS, et on ajoute $ems'_i = (G, c, \phi \cup \phi_i)$. Ceci conserve la canonicité. **Pour l'intersection**, dans le cas des ems atomiques minimaux, si les graphes mémoire ne sont pas isomorphes l'intersection est vide (cf. lemme 6.3.2). Sinon l'intersection est le graphe mémoire (commun) et l'intersection des formules de Presburger. On distribue ensuite l'intersection sur chaque atome de l'ems. Cette opération conserve la minimalité. □

Toutefois, les ems ne sont pas clos par complémentation. Intuitivement ceci vient du fait que le complément d'un ems aurait une infinité de graphes mémoire, et n'est donc pas un ems.

Théorème 6.3.7. *Les ensembles ems-définissables ne sont pas clos par complément.*

Démonstration. Soit un ems atomique minimal $ems = (G, c, \phi)$. On considère l'ensemble ems-définissable $X = [\![ems]\!]$. Nous voulons montrer que le complément de X, noté $\bar{X}$, n'est pas ems-définissable. L'ensemble de graphes mémoire $\bar{X}$ contient au moins l'ensemble des graphes mémoire minimaux G' non isomorphes à G. Il existe un nombre infini de tels G'. Or un ensemble ems-définissable n'a qu'un nombre fini de graphes mémoire minimaux. On en déduit que $\bar{X}$ n'est pas ems-définissable. □

Comme le vide est ems-définissable, on déduit le corollaire suivant.

Corollaire 6.3.2. L'ensemble $\mathcal{G}$ de tous les graphes mémoire n'est pas ems-définissable.

6.3.8 Inclusion et test du vide

Nous définissons une opération d'inclusion symbolique $\sqsubseteq$ sur les ems.

Définition 6.3.8. L'opération d'inclusion symbolique sur les ems est définie inductivement par :
- deux ems atomiques $ems = (G, c, \phi)$ et $ems' = (G', c', \phi')$ vérifient $ems \sqsubseteq ems'$ si les conditions suivante sont réunies :
  - (G, c) et (G', c') sont isomorphes, et
  - la formule $\phi \Rightarrow \phi'$ est valide ;
- segf $\sqsubseteq ems$ si $ems =$ segf ;
- pour tout ems atomique $\bot \sqsubseteq ems$;
- Le cas des ems non atomiques :
  - $\{ems_1, \ldots, ems_n\} \sqsubseteq EMS$ si tout ems_i vérifie $ems_i \sqsubseteq EMS$.
  - $ems \sqsubseteq \{ems_1, \ldots, ems_n\}$ s'il existe ems_i tel que $ems \sqsubseteq ems_i$.

L'opération $\sqsubseteq$ est cohérente vis à vis de l'inclusion des ensembles ems-définissables. L'inclusion d'ensembles ems-définissables est donc décidable.

Théorème 6.3.8. *La relation d'inclusion est décidable sur les ensembles ems-définissables, et $[\![EMS_1]\!] \subseteq [\![EMS_2]\!]$ si et seulement si $EMS_1 \sqsubseteq EMS_2$.*

Démonstration. Le sens $EMS_1 \sqsubseteq EMS_2 \Rightarrow [\![EMS_1]\!] \subseteq [\![EMS_2]\!]$ découle des définitions. Le sens $EMS_1 \sqsubseteq EMS_2 \Leftarrow [\![EMS_1]\!] \subseteq [\![EMS_2]\!]$ s'obtient grâce à la forme minimale. En appliquant les définition de $\sqsubseteq$ et $[\![\cdot]\!]$, le point qui pose problème est de montrer que $[\![ems]\!] \subseteq [\![ems_1]\!] \cup [\![ems_2]\!]$ implique que soit $[\![ems]\!] \subseteq [\![ems_1]\!]$ soit $[\![ems]\!] \subseteq [\![ems_2]\!]$, avec ems, ems_1, ems_2 atomiques et minimaux, et ems_1 et ems_2 non isomorphes. Si ems a une intersection non vide avec ems_1 (resp. ems_2), comme ils sont minimaux, ems est isomorphe à ems_1 (resp. ems_2). ems ne peut avoir une intersection non vide avec les deux, sinon ems_1 et ems_2 seraient isomorphes. Donc si $[\![ems]\!] \subseteq [\![ems_1]\!] \cup [\![ems_2]\!]$, $[\![ems]\!]$ est forcément inclus totalement dans $[\![ems_1]\!]$ ou totalement dans $[\![ems_2]\!]$. □

On en déduit la propriété suivante.

Corollaire 6.3.3. Le vide est décidable sur les ems.

6.3.9 Successeurs symboliques

On s'attache maintenant à définir l'opération POST sur les ems, en adéquation avec la concrétisation $[\![\cdot]\!]$. Nous donnons la transformation pour un ems atomique minimal. Les opérations se distribuent sur l'union finie.

La donnée de la sémantique opérationnelle symbolique nécessite quelques définitions.

- Étant donné un état mémoire symbolique (M, c, ϕ), un prédicat p sur les graphes mémoire et $[\![p]\!]$ l'ensemble des graphes mémoire vérifiant p, alors $\phi|_p$ est la plus grande formule telle que $[\![(G, c, \phi|_p)]\!] \subseteq [\![p]\!]$ et $\phi|_p \Rightarrow \phi$ est valide.
- Pour le prédicat *IsNull*(x), on définit $\phi_{IsNull(x)}$ de la manière suivante. Soit le nœud n_1 tel $x \in Var(n1)$. Si `null` est accessible depuis n_1, on note $n_1 n_2 \ldots n_m$`null` l'unique chemin de n_1 à `null`, et $c(n_1), \ldots, c(n_m)$ les compteurs associés au chemin. On a alors : $\phi_{IsNull(x)} = \emptyset$ si `null` n'est pas accessible depuis n_1, $\phi_{IsNull(x)} = \phi \wedge \sum_{i=1}^{m} c_i = m$. La définition pour $\neg$*IsNull*(x) découle directement.

Nous définissons aussi deux opérations communes à de nombreuses opérations : le rajout de nœuds intermédiaires et la normalisation.

Rajout de nœud intermédiaire. Une opération récurrente est l'insertion d'un nœud dans une liste. Plus exactement on veut rajouter un nœud intermédiaire n' à la liste représentée par l'arc $(n_0, s(n_0))$, tel que n' soit le successeur direct de n_0 ($c'(n_0) = 1$). Soit un nouveau nœud $n' \notin N$ et deux nouveaux compteurx $k', k'_0 \notin c(N)$. Alors le nouvel ems $(N', s', var', c', \phi')$ est défini par :

- $N' = N \cup \{n'\}$,
- s' est définie par : $s'(n') = s(n_0)$, $s'(n_0) = n'$, sinon $s'(n) = s(n)$ pour $n \neq n'$ et $n \neq n_0$.
- var' est définie par : $var'(n') = \emptyset$, sinon $var'(n) = var(n)$ pour $n \in N$.
- c' est définie par : $c'(n') = k'$, $c'(n_0) = k'_0$, sinon $c'(n) = c(n)$ pour $n \neq n'$ et $n \neq n_0$.
- ϕ' est définie par $\phi' = \exists k_0.(\phi \wedge (k' = k0 - 1) \wedge (k'_0 = 1))$, où k_0 est le compteur tel que $c(n_0) = k_0$.

Cette opération ne garantit plus que les compteurs sont toujours strictement positifs. On a donc besoin d'une opération de normalisation pour se ramener au cas où les compteurs sont strictement supérieurs à 0.

Normalisation. Certaines des opérations précédentes peuvent rompre la contrainte d'avoir des compteurs strictement positifs sur les arcs. Nous décrivons ici une procédure de normalisation pour retourner un ems. Soit *ems* $=$ (G, c, ϕ) un ems, à la différence qu'il existe un nœud n tel que $\phi \Rightarrow c(n) > 0$ n'est pas valide. Nous transformons *ems* en considérant les deux cas où

$c(n) > 0$ (l'arc est conservé) et $c(n) = 0$ (l'arc est enlevé, les nœuds $n, s(n)$ sont fusionnés). Nous retournons donc les deux ems atomiques ems_1 et ems_2 correspondant à ces deux situations :

- $ems_1 = (G, c, \phi \wedge c(n) > 0)$;
- $ems_2 = (G', c', \phi')$ est obtenu à partir de ems en
  - remplaçant n et $s(n)$ par n' tel que $s'(n') = s(s(n))$ et tout n_i vérifiant $s(n_i) = n$ vérifie maintenant $s'(n_i) = n'$, et
  - $var'(n') = var(n) \cup var(s(n))$ et $c'(n') = c(s(n))$, et
  - ϕ' vaut $\exists c(n).(\phi \wedge c(n) = 0)$.

Sémantique opérationnelle. Nous définissons, comme dans le cas concret, une fonction f_a pour exprimer l'effet de $a \in \Phi_p$. Cette fois la fonction f_a est totale car les ems incluent le vide et segf. On note n_z le nœud pointé par le pointeur z. Soit l'ems minimal $ems = (G, c, \phi)$.

1. Si a est de la forme $x :=$**new** alors : soit G' tel que $G \xrightarrow{x := \mathbf{new}} G'$. G' a un nouveau nœud n_x étiqueté par x, relié à `puits` dont le compteur vaut 1. On retourne $f_a(ems) = (G', c, \phi \wedge c(n_x) = 1)$ et $f_a(ems)$ est minimal.

2. Si a est de la forme $x := y$ alors : soit G' tel que $G \xrightarrow{x := y} G'$. On retourne $f_a(ems) = (G', c, \phi)$ et $f_a(ems)$ est minimal.

3. Si a est de la forme $x :=$**null** alors : soit G' tel que $G \xrightarrow{x := \mathbf{null}} G'$. On retourne $f_a(ems) = (G', c, \phi)$ et $f_a(ems)$ est minimal.

4. Si a est de la forme $x := y.s$ alors : si $y.s$ pointe sur `erreur` (dans G) on retourne $f_a(ems) =$ segf. Sinon on applique l'opération d'insertion de nœud décrite plus haut à l'arc $(n_y, s(n_y))$. Appelons n' le nœud ajouté. Alors $var'(n') = \{x\}$, $var'(n_x) = var(n_x) \setminus \{x\}$ Il faut normaliser l'arc ajouté au moyen de la procédure décrite plus haut, puis minimiser (uniquement si le nœud n_x n'est plus un nœud principal).

5. Si a est de la forme $x.s := y$ alors : si $s(x) =$ `erreur` alors on retourne $f_a(ems) =$ segf. Sinon on ajoute un nœud n' à l'arc $(n_x, s(n_x))$. Alors $var'(n') = \{y\}$, $var'(n_y) = var(n_y) \setminus \{y\}$. Il faut normaliser l'arc ajouté au moyen de la procédure décrite plus haut, puis minimiser (uniquement si le nœud n_y n'est pas un nœud principal).

6. Si a est de la forme $\textbf{free}(x)$ alors : si $n_x = \texttt{puits}$ alors retourner segf, si $n_x = \texttt{null}$ retourner le même *ems*. Sinon on transforme G de façon légèrement différente du cas concret. Les nœuds n tels que $s(n) = n_x$ vérifient alors $s(n) = \texttt{puits}$. Soit $k_x = c(n_x)$ et $k' \notin c(N)$ un nouveau compteur. Alors $c'(n_x) = k'$, et $\phi' = \exists k_x.(\phi \wedge k' = k_x - 1)$. Il faut normaliser l'arc $(n_x, s(n_x))$. Le résultat $f_a(ems)$ est minimal.

Nous définissons maintenant la fonction POST par : $\text{POST}(g?\ a, ems) = ems'$

- si $ems = (G, c, \phi)$:
  - si g est de la forme *True*, alors $ems' = f_a(ems)$,
  - si g est de la forme *IsNull*(x) alors $ems' = f_a((G, c, \phi|_{IsNull(x)}))$,
  - si g est de la forme $\neg$*IsNull*(x) alors $ems' = f_a((G, c, \phi|_{\neg IsNull(x)}))$,
- si $ems = \text{segf}$ ou $ems = \perp$ alors $ems' = \perp$;
- pour les ems généraux, POST se distribue par rapport à l'union.

La construction ci-dessus amène au théorème suivant.

Théorème 6.3.9. *Les ensembles* ems*-définissables sont clos par image d'une relation de Φ_p. De plus ce résultat est effectif et la fonction récursive* POST *vérifie : pour tout $\varphi \in \Phi_p$ et pour tout état mémoire symbolique EMS, on a* $[\![\text{POST}(\varphi, EMS)]\!] = \varphi([\![EMS]\!])$.

6.3.10 Propriétés

Les propriétés de fuite mémoire **PF** et de violation mémoire **PV** sont indécidables sur les systèmes à pointeurs. Nous nous intéressons ici aux problèmes plus simples de la fuite mémoire faible, notée **PF'**, et de la violation mémoire faible, notée **PV'**.

PF' Soit un ems EMS, on veut décider s'il existe un graphe mémoire $G \in [\![EMS]\!]$ tel que G soit une fuite mémoire.

PV' Soit un ems EMS et une action $\varphi \in \Phi_p$, on veut décider s'il existe un graphe mémoire $G \in [\![EMS]\!]$ tel que $\text{post}(\varphi, G) = \text{segf}$.

Proposition 6.3.1. Les propriétés **PF'** et **PV'** sont décidables sur les états mémoire symboliques.

Démonstration. La concrétisation d'un ems atomique minimal ems $= (G, c, \phi)$ contient des fuites mémoire si et seulement si G est une fuite mémoire. Donc **PF'** est décidable. Pour **PF'**, la sémantique de POST permet de décider quand une violation mémoire se

produit. □

Ce résultat nous assure que si l'ensemble d'accessibilité $\mathsf{post}^*(X_0)$ d'un système à pointeurs (S, X_0) est ems-définissable et peut être calculé symboliquement (par exemple par la procédure ACCESS1 du chapitre 2), alors on sait décider les propriétés de violation mémoire **PV** et de fuite mémoire **PF** sur (S, X_0).

6.4 Abstraction de systèmes à pointeurs

Nous définissons dans cette section une abstraction sur les graphes mémoire consistant à "oublier" les fuites mémoire d'un graphe mémoire en ne conservant que l'information partielle : le graphe est une fuite mémoire ou le graphe n'est pas une fuite mémoire. D'une part cette abstraction conserve les propriétés **PV** et **PF**, d'autre part elle permet de manipuler une classe restreinte de graphes mémoire dont le nombre est borné et pour lesquels le test d'isomorphisme est linéaire.

6.4.1 Graphes mémoire abstraits

Tout d'abord nous définissons l'abstraction et montrons qu'elle est correcte vis à vis de **PV** et **PF**.

Un graphe mémoire abstrait $G^\sharp$ est un couple $G^\sharp = (G, b)$ tel que $G \in \mathcal{G}$ est un graphe mémoire sans fuite mémoire et $b \in \{0, 1\}$. L'ensemble des graphes mémoire abstraits est noté $\mathcal{G}^\sharp$.

Nous définissons l'abstration $\alpha : \mathcal{G} \rightarrow \mathcal{G}^\sharp$ sur les graphes mémoire par :
- $\alpha(G) = (G, 0)$ si G n'est pas une fuite mémoire ;
- $\alpha(G) = (G', 1)$, où G' est le graphe mémoire G privé des nœuds non accessibles si G est une fuite mémoire.

La concrétisation $[\![\cdot]\!] : \mathcal{G}^\sharp \rightarrow 2^{\mathcal{G}}$ d'un graphe mémoire abstrait est alors définie par : $[\![(G, 0)]\!] = [\![G]\!]$, et $[\![(G, 1)]\!]$ est le plus grand $X \subseteq \mathcal{G}$ tel que $\alpha(X) = [\![G]\!]$ et tous les graphes mémoire de X sont des fuites mémoire. Cet ensemble existe et vaut $\alpha^{-1}(X) \cap \mathcal{G}_f$, où $\mathcal{G}_f$ désigne l'ensemble des graphes mémoire fuites mémoire.

On étend l'abstraction en $\alpha : \mathcal{G} \times \{0, 1\} \rightarrow \mathcal{G}^\sharp$. Notons $\alpha(G) = (G', b)$. Alors $\alpha((G, 1))$ vaut $(G', 1)$ et $\alpha((G, 0))$ vaut (G', b). Soit l'extension de post :

$\mathcal{G} \times \{0,1\} \to \mathcal{G} \times \{0,1\}$ définie par $\mathsf{post}(G,b) = (\mathsf{post}(G), b)$. On obtient le résultat suivant :

Lemme 6.4.1. Soient $x \in \mathcal{G} \times \{0,1\}$ et l'action $\varphi \in \Phi_p$. Alors on a l'égalité $\alpha(\mathsf{post}(\varphi, \alpha(x))) = \alpha(\mathsf{post}(\varphi, x))$.

Démonstration. Les nœuds morts d'un graphe mémoire G n'influencent pas le calcul des successeurs, et le calcul des successeurs ne peut retirer les fuites mémoire d'un graphe mémoire. Aussi qu'on les enlève ou non avant l'application de post suivi de α ne change rien au résultat calculé. □

L'opération de successeurs approchée $\mathsf{post}^\sharp : \mathcal{G}^\sharp \to \mathcal{G}^\sharp$ pour les graphes mémoire abstraits est définie par $\mathsf{post}^\sharp(G^\sharp) = \alpha(\mathsf{post}(G^\sharp))$. La sémantique définie par $\mathsf{post}^\sharp$ est une surapproximation de la sémantique définie par post. C'est-à-dire que pour $G \in \mathcal{G}$ et $\varphi \in \Phi_p$, on a $\mathsf{post}(\varphi, G) \in [\![\mathsf{post}^\sharp(\varphi, \alpha(G))]\!]$. Cette sémantique est cependant assez fine, puisqu'on a le résultat suivant.

Lemme 6.4.2. Soient un graphe mémoire $G \in \mathcal{G}$ et (S, G) un système à pointeurs. Alors $\mathsf{post}^{\sharp *}(\alpha(G)) = \alpha(\mathsf{post}^*(G))$.

Démonstration. En utilisant le lemme 6.4.1 on peut montre par récurrence que pour tout $i \geq 0$, $\mathsf{post}^{\sharp i}(\alpha(G))$ est égal $\alpha(\mathsf{post}^i(G))$. D'où la conclusion. □

On désigne par $S^\sharp$ le système à pointeurs S muni de la sémantique approchée $\mathsf{post}^\sharp$. Nous pouvons maintenant déduire le résultat principal de cette section, c'est-à-dire que la sémantique approchée d'un système à pointeurs conserve les propriétés de fuite mémoire et de violation mémoire.

Théorème 6.4.1 (Conservation de **PV** et **PF**)**.** *L'abstraction α conserve les propriétés* **PV** *et* **PF**. *C'est-à-dire qu'un système à pointeurs $(S, (q_0, G_0))$ vérifie* **PV** *(resp.* **PF***) si et seulement si le système à pointeurs abstrait $(S^\sharp, (q_0, \alpha(G_0)))$ vérifie* **PV** *(resp.* **PF***).*

Démonstration. Le résultat découle directement lemme 6.4.2. □

6.4.2 États mémoire symboliques abstraits

Nous voulons maintenant étendre l'abstraction définie précédement aux états mémoire symboliques.

Comme nous voulons construire nos états mémoire symboliques abstraits sur le même modèle que les états mémoire symboliques, nous nous intéressons

aux *graphes mémoire minimaux sans fuite mémoire*, appelés graphes mémoire étanches.

Définition 6.4.1 (Graphe mémoire étanche)**.** Un *graphe mémoire étanche* est un graphe mémoire minimal sans fuite mémoire.

Nous dirons d'un ems ems qu'il est étanche si $ems = (G, c, \phi)$ et le graphe mémoire G est étanche, ou $ems = \mathsf{segf}$ ou $ems = \bot$. Nous définissons maintenant les états mémoire symboliques abstraits.

Définition 6.4.2 (États mémoire symboliques atomiques abstraits)**.** Un état mémoire symbolique atomique abstrait, aussi appelé $\mathsf{ems}^\sharp$, est une paire $\mathrm{ems}^\sharp = (ems, b)$ où ems est un ems atomique étanche et $b \in \{0, 1\}$.

Nous définissons la concrétisation d'un $\mathsf{ems}^\sharp$ atomique par :
- si $b = 0$ alors $[\![\mathrm{ems}^\sharp]\!] = [\![ems]\!]$,
- si $b = 1$ alors $[\![\mathrm{ems}^\sharp]\!]$ est le plus grand $X \subseteq \mathcal{G}$ tel que $X^\sharp = [\![ems]\!]$ et tous les graphes mémoire de X sont des fuites mémoire.

Les états mémoire symboliques abstraits sont des ensembles finis d'états mémoire symboliques atomiques abstraits $\mathrm{EMS}^\sharp = \{\mathrm{ems}_1^\sharp, \ldots, \mathrm{ems}_n^\sharp\}$. Tous les ensembles ems-définissables sont $\mathsf{ems}^\sharp$-définissables.

Opérations ensemblistes des $\mathsf{ems}^\sharp$. Les bonnes propriétés de clôture des ensembles ems-définissables sont toujours valides pour les ensembles $\mathsf{ems}^\sharp$-définissables. On peut ainsi montrer que les ensembles $\mathsf{ems}^\sharp$-définissables sont clos par union et intersection, et que l'inclusion et le vide d'ensembles $\mathsf{ems}^\sharp$-définissables sont décidables. Voici comment adapter les opérations symboliques sur les ems aux $\mathsf{ems}^\sharp$. L'union se fait toujours directement. L'intersection de deux $\mathsf{ems}^\sharp$ (ems_1, b_1) et (ems_2, b_2) est vide si $b_1 \neq b_2$ et vaut $(ems_1 \sqcap ems_2, b_1)$ sinon. Le vide se teste sur ems et l'inclusion est vide si $b_1 \neq b_2$, équivalente à $ems_1 \sqsubseteq ems_2$ sinon.

Sémantique abstraite. Nous définissons une fonction de successeurs symboliques approchée $\textsc{post}^\sharp$ pour les $\mathsf{ems}^\sharp$. La fonction d'abstraction α est étendue aux ems atomiques $ems = (G, c, \phi)$, par $\alpha(ems) = ((G, c, \phi), 0)$ si G n'est pas une fuite mémoire ; et si G est une fuite mémoire alors $\alpha(ems) = ((G', c, \exists c_1, \ldots, c_m.\phi), 1)$, où le graphe mémoire G' s'obtient à partir de G en enlevant les nœuds fuite mémoire, et les compteurs $c_1, \ldots, c_m$ sont les compteurs associés à ces nœuds. On définit de plus $\alpha(\bot) = (\bot, 0)$ et $\alpha(\mathsf{segf}) = (\mathsf{segf}, 0)$. Ensuite comme pour les graphes mémoire abstraits, on étend α aux couples formés d'un ems et d'un booléen. Alors le successeur abstrait $\textsc{post}^\sharp$ est défini par $\textsc{post}^\sharp(\mathrm{ems}^\sharp) = \alpha(\textsc{post}(\mathrm{ems}^\sharp))$. L'extension aux $\mathsf{ems}^\sharp$ quelconques est directe.

6.4.3 Graphes mémoire étanches

Nous nous intéressons maintenant à la classe des graphes mémoire étanches, apparaissant dans les abstractions manipulées. Soit F_v le nombre de graphes mémoire étanches (à un isomorphisme près) d'un ensemble Var tel que $|Var| = v$. Le résultat principal de cette section est de montrer que pour tout v, F_v est fini.

Soit G un graphe quelconque. Nous notons $Sdeg(n)$ le nombre d'arcs sortants du nœud n. Dans un graphe mémoire $Sdeg(n) = 1$. Nous montrons tout d'abord deux lemmes techniques.

Lemme 6.4.3. Soit G un graphe fini tel que, pour chaque nœud, le degré est différent de 1 et le degré de sortie est exactement 1. On note P_0 le nombre de nœuds de degré 0. Alors $|G| \leq 2P_0$.

Démonstration. Tout graphe fini G vérifie $\sum_{n\in G} deg(n) = \sum_{n\in G} Sdeg(n)$. Un nœud n est soit dans P_0 (donc $Sdeg(n) = 0$), soit dans $G - P_0$ (donc $Sdeg(n) \geq 1$). Donc $2(|G| - P_0) \leq \sum_{n\in G} Sdeg(n)$. Pour chaque nœud n, $Sdeg(n) = 1$. Donc $\sum_{n\in G} Sdeg(n) = |G|$. On déduit finalement que $|G| \leq 2P_0$. □

Lemme 6.4.4. Soit G un graphe étanche. Soit P_0 le nombre de nœuds de G de degré égal à 0. Alors $P_0 \leq |Var|$.

Démonstration. On raisonne par l'absurde. Si $P_0 > |Var|$ alors il existe au moins un nœud n de degré 0 non étiqueté par une variable de pointeur de Var. On en déduit que n n'est pas accessible, d'où une contradiction avec G sans fuite mémoire. □

Nous pouvons maintenant démontrer le résultat principal de cette section.

Théorème 6.4.2. *Soit un ensemble donné Var de variables de pointeurs. Alors le nombre de graphes mémoire étanches sur Var est fini et borné par le nombre de graphes à $2.|Var|$ nœuds.*

Démonstration. Soit $G = (N, s, var)$ un graphe mémoire étanche. Soit (N', s') le graphe fini obtenu à partir de G en supprimant les nœuds de degré 1, leurs arcs entrants et sortants, et en ajoutant un arc de leur prédécesseur à leur successeur. Nous avons : $|N| = |N'| + |\{n \in N \mid deg(n) = 1\}|$. En utilisant le lemme 6.4.3, il vient que $|N'| \leq 2P_0$ où P_0 est le nombre de nœuds de degré 0. Par définition d'un graphe mémoire étanche, un nœud $n \in N$ de degré 0 ou 1 est étiqueté par au moins une variable. Donc $|\{n \in N \mid deg(n) = 1\}| \leq |Var| - P_0$. En utilisant le lemme 6.4.4, nous obtenons finalement $|N| \leq 2|Var|$. Il n'y a donc qu'un nombre fini de graphes mémoire

étanches (à un isomorphisme près), que l'on peut construire à partir d'un ensemble fini de variables de pointeurs Var. □

6.4.4 Graphes mémoire canoniques

Nous montrons maintenant que les graphes mémoire étanches peuvent être canonisés en temps linéaire. Ceci permet de montrer que le problème de l'isomorphisme de graphes mémoire étanches est décidable en temps linéaire.

Nous définissons quelques notions nécessaires à la canonisation. Soit un graphe mémoire étanche G, et une variable $x \in Var$. Nous appelons *mot de x dans G*, noté w_x, la plus grande suite de nœuds $n_0 n_1 \ldots n_{fin}$ telle que : $n_0 = n_x$, pour tout i, $s(n_i) = n_{i+1}$ et tous les n_i sont différents. Le *mot du graphe G*, noté w_G, est la concaténation des $w_{x_i}, x_i \in Var$, ordonnés selon l'ordre lexicographique des x_i. Ainsi $w_G = w_{x_1} \ldots w_{x_{|Var|}}$ avec pour tout i, $w_{x_i} \prec w_{x_{i+1}}$.

Lemme 6.4.5. Soient deux graphes mémoire étanches G et G'. Alors $G = G'$ si et seulement si $w_G = w_{G'}$.

Démonstration. Le premier sens est évident. Nous montrons la réciproque. Comme les nœuds d'un graphe mémoire ont exactement un successeur, on peut reconstruire sans ambiguité le graphe G à partir du mot w_G. Donc si $w_G = w_{G'}$ alors $G = G'$. □

On définit maintenant la notion de graphe mémoire canonique.

Définition 6.4.3 (Graphe mémoire canonique)**.** Un graphe mémoire canonique est un graphe mémoire étanche $G = (N, s, var)$ tel que :

- $N = \{n_1, \ldots, n_{|N|-3}, \texttt{null}, \texttt{erreur}, \texttt{puits}\}$,
- w_G est minimal, c'est-à-dire que tout graphe mémoire étanche G' isomorphe à G vérifie $w_G \preceq w_{G'}$.

Tout graphe mémoire étanche est isomorphe à un unique graphe mémoire canonique. Il est de plus possible de construire effectivement ce graphe mémoire canonique en temps linéaire.

Théorème 6.4.3 (canonisation)**.** *Soit un graphe mémoire étanche G. Alors il existe un unique graphe mémoire canonique G' tel que G et G' sont isomorphes. De plus G' est calculable en temps linéaire.*

Démonstration. **Unicité.** Soient deux graphes mémoire canoniques isomorphes G et G'. Donc ils vérifient l'égalité $w_G = w_{G'}$. Le lemme 6.4.5 permet de conclure.

Existence effective. On canonise le graphe mémoire étanche G en renommant les nœuds. On part du sommet étiqueté par la plus petite variable de Var (en ordre lexicographique). Si ce nœud est différent de `puits`, `erreur`, `null`, il est étiqueté n_1. On suit les successeurs du nœud. À chaque nœud rencontré non étiqueté, on attribue le prochain n_i. On s'arrête quand on arrive sur un nœud déjà étiqueté, ou un nœud spécial : `puits`,`erreur`,`null`. Dans ce cas on recommence en prenant la plus petite variable restante de Var, et ainsi de suite. Le graphe mémoire étanche G' ainsi construit est isomorphe à G, et il est bien canonique. En effet tout G'' isomorphe à G' vérifie $w_{G'} \preceq w_{G''}$, car les nœuds de G' sont choisis pour minimiser $w_{G'}$.

L'algorithme de canonisation est linéaire en N puisque chaque nœud non spécial n'est parcouru qu'une fois, et que les nœuds spéciaux `puits` et `null` sont parcourus en tout au plus $|Var|$ fois. □

L'algorithme de canonisation peut être adapté pour tester l'isomorphisme de graphes mémoire étanches en temps linéaire.

Théorème 6.4.4. *L'isomorphisme de graphes mémoire étanches est décidable en temps linéaire.*

Démonstration. Deux graphes mémoire étanches G_1 et G_2 sont isomorphes si et seulement si ils ont même graphe mémoire canonique. Pour décider l'isomorphisme il suffit donc de canoniser G_1 et G_2, de calculer w_{G_1} et w_{G_2}. Alors G_1 et G_2 sont isomorphes si et seulement si $w_{G_1} = w_{G_2}$. □

Ce test est linéaire dans le nombre de nœuds du graphe. On peut améliorer (un peu) la procédure en canonisant en parallèle les deux graphes mémoire. Dès que les mots construits sont différents, les deux graphes mémoire ne sont pas isomorphes. Sinon, ils sont isomorphes et la procédure les a canonisés.

Ce résultat est intéressant car les opérations sur les $\mathsf{ems}^\sharp$ requièrent souvent de tester l'isomorphisme de graphes mémoire.

6.4.5 Complément d'états mémoire symboliques abstraits

Les résultats sur les graphes mémoire étanches permettent de montrer que les ensembles $\mathsf{ems}^\sharp$-définissables sont clos par complémentation.

Nous montrons tout d'abord le résultat intermédiaire suivant.

Lemme 6.4.6. On note $\mathcal{G}_f$ l'ensemble des graphes mémoire fuite mémoire et $\mathcal{G}_{\neg f}$ son complémentaire. Les ensemble $\mathcal{G}, \mathcal{G}_f$ et $\mathcal{G}_{\neg f}$ sont $\mathsf{ems}^\sharp$-définissables.

Démonstration. On se sert du nombre fini de graphes mémoires étanches (lemme 6.4.2). On les note $G_1, \ldots, G_n$. Considérons les n ems$^\sharp$ atomiques canoniques : $\mathsf{ems}^\sharp_i = ((G_i, c_i, (\forall c.c > 0)), 1)$. Alors on vérifie que $[\![\mathcal{G}_f]\!] = \left[\!\!\left[\bigsqcup_i \mathsf{ems}^\sharp_i\right]\!\!\right]$. Donc $\mathcal{G}_f$ est ems$^\sharp$-définissable. On note maintenant $\mathsf{ems}^{\sharp'}_i = ((G_i, c_i, (\forall c.c > 0)), 0)$. Alors on vérifie que $[\![\mathcal{G}_{\neg f}]\!] = \left[\!\!\left[\bigsqcup_i \mathsf{ems}^{\sharp'}_i\right]\!\!\right]$. Donc $\mathcal{G}_{\neg f}$ est ems$^\sharp$-définissable. Enfin comme $\mathcal{G} = \mathcal{G}_f \cup \mathcal{G}_{\neg f}$, on déduit que $\mathcal{G}$ est ems$^\sharp$-définissable. □

Ceci nous amène au résultat sur la complémentation.

Théorème 6.4.5. *Les ensembles* ems$^\sharp$*-définissables sont clos par complémentation, et il existe une fonction récursive qui prend en entrée un* ems$^\sharp$ $EMS^\sharp$ *et retourne un* ems$^\sharp$ $EMS^{\sharp'}$ *tel que* $\left[\!\!\left[EMS^{\sharp'}\right]\!\!\right] = \mathcal{G} \setminus [\![EMS^\sharp]\!]$.

Démonstration. Nous raisonnons sur les ems$^\sharp$ atomiques. Soit un ensemble $X \subseteq \mathcal{G}$ de graphes mémoire tel qu'il existe un ems$^\sharp$ atomique $\mathsf{ems}^\sharp = (ems, b)$ tel que $[\![\mathsf{ems}^\sharp]\!] = X$. On note $ems = (G, c, \phi)$. On suppose que $b = 0$, c'est-à-dire que l'ensemble X ne contient aucune fuite mémoire. On désigne par $\bar{X}_1, \bar{X}_2$ et $\bar{X}_3$ les ensembles de graphes mémoire suivants : $\bar{X}_1 = \{G' | G' \text{est une fuite mémoire}\}$, $\bar{X}_2 = \{G' | G \not\preceq G' \text{ et } G' \text{ n'est pas une fuite mémoire}\}$ et $\bar{X}_3 = [\![(G, c, \neg\phi)]\!]$. Alors le complément de X, noté $\bar{X}$, est égal à $\bar{X} = \bar{X}_1 \cup \bar{X}_2 \cup \bar{X}_3$. On montre que $\bar{X}_1$ et $\bar{X}_2$ sont ems$^\sharp$-définissables en adaptant la preuve du lemme 6.4.6. L'ensemble $\bar{X}_3$ est ems$^\sharp$-définissable par définition. Donc l'ensemble $\bar{X}$ est ems$^\sharp$-définissable. La preuve dans le cas $f = 1$ est similaire. Ceci prouve le résultat pour les ems$^\sharp$ atomiques. Le résultat s'étend aux ems$^\sharp$ quelconques car les ensembles ems$^\sharp$-définissables sont clos par intersection. □

Ce résultat fournit une procédure effective qui prend en entrée un ensemble ems$^\sharp$-définissable codé par un ems$^\sharp$ et fournit en sortie le complément.

6.4.6 Résumé sur les états mémoire symboliques

Nous présentons à la figure 6.8 un comparatif des cadres symboliques définis par les états mémoire symboliques et les états mémoire symboliques abstraits.

Les deux approches permettent de vérifier les propriétés de violation mémoire faible et fuite mémoire faible. De plus l'aliasing quantitatif défini par Iosif et Bozga dans [BI05b] est également décidable [BI05a] dans les deux cas.

	ems	$\text{ems}^\sharp$
union, inclusion	oui	oui
intersection, vide	oui	oui
POST	*exact*	*approché*
complément	*non*	*oui*
$\mathcal{G}$ représentable	*non*	*oui*
type de graphes	graphe mémoire	graphe mém. canonique
isomorphisme de graphe	exponentiel	*linéaire*
violation mém. faible	décidable	décidable
fuite mém. faible	linéaire	*constant*
alias quantitatif	décidable	décidable

FIG. 6.8: Comparaison des ems et $\text{ems}^\sharp$

Comme en plus les $\text{ems}^\sharp$ ont de meilleurs propriétés algorithmiques que les ems, les états mémoires symboliques abstraits s'avèrent être exactement le bon cadre symbolique pour la vérification quantitative de systèmes à pointeurs.

6.5 Conclusion

Nous nous sommes concentrés dans ce chapitre sur la vérification de programmes à allocation dynamique de mémoire. Plus précisément nous avons défini un cadre symbolique (cf. chapitre 2) pour les systèmes à pointeurs. Les systèmes à pointeurs modélisent des programmes séquentiels manipulant des listes simplement chaînées, cycliques ou non, et pouvant allouer et désallouer arbitrairement des cellules mémoire.

Les états mémoire symboliques disposent de toutes les opérations nécessaires au calcul symbolique de l'ensemble d'accessibilité (l'union, le calcul des successeurs et l'inclusion) et on peut définir également l'intersection et le test du vide. Enfin on peut décider sur un état mémoire symbolique les propriétés de violation mémoire faible et fuite mémoire faible et toutes les propriétés définies par la logique d'alias à compteurs.

Nous proposons enfin une abstraction du calcul des successeurs dans les systèmes à pointeurs. D'une part l'ensemble d'accessibilité abstrait, s'il est

calculable, permet encore de décider les propriétés de fuite mémoire faible, violation mémoire faible et alias à compteurs pour l'ensemble d'accessibilité concret. D'autre part, les états mémoire symboliques abstraits ont de meilleurs propriétés algorithmiques : nombre borné de graphes mémoire sous-jacents et isomorphisme de graphes décidable en temps linéaire. Les états mémoires symboliques abstraits s'avèrent donc être exactement le bon cadre symbolique pour la vérification quantitative de systèmes à pointeurs.

Actuellement notre approche permet de vérifier un invariant d'un système à pointeurs et de lancer un calcul de point fixe itératif en avant. Cependant comme expliqué au chapitre 2, un tel calcul termine dans des cas bien particuliers. Aussi le développement de techniques d'accélération pour améliorer la convergence pratique du calcul est souhaitable.

Conclusion

Bilan

Nous avons étudié dans cette thèse le problème de la vérification de systèmes infinis par accélération du calcul de l'ensemble d'accessibilité. Nos résultats sont à trois niveaux : des résultats théoriques généraux sur l'accélération et l'étude de la composition automatique d'accélérations pour la vérification des systèmes hétérogènes ; des résultats de nature algorithmique sur deux types de données particuliers, les compteurs et les pointeurs ; enfin des résultats expérimentaux avec l'implantation des techniques d'accélération dédiées aux compteurs dans l'outil de vérification Fast.

Nous avons proposé un cadre théorique unifiant la plupart des techniques d'accélérations connues. Ce cadre propose des points-clés pour comparer plus facilement les différentes méthodes et donne des repères pour le développement de nouvelles techniques d'accélération. Nous avons aussi contribué à la mise en œuvre des théorèmes d'accélération plate dans une procédure de calcul de point fixe. La notion d'applatissement permet de cerner exactement les systèmes dont l'ensemble d'accessibilité est calculable par accélération plate. Nous proposons différents algorithmes complets pour le calcul d'accessibilité des systèmes applatissables, et des optimisations indépendantes du type de données considéré.

Nous nous sommes intéressés à la vérification de systèmes hétérogènes sous l'angle de la composition automatique d'accélérations. Nos principaux résultats sont d'identifier la classe des systèmes faiblement hétérogènes et de définir une classe d'accélérations composables sur ces systèmes. Le résultat de composition est effectif modulo quelques hypothèses sur les accélérations.

Pour les systèmes à compteurs, des algorithmes d'accélération étaient déjà développés. Nous nous sommes concentrés sur l'efficacité pratique des techniques d'accélération et d'applatissement. Nous avons proposé de nou-

veaux algorithmes d'accélération pour les systèmes manipulant des translations convexes et des translations positives. Ces algorithmes ont à la fois une meilleure complexité théorique et de meilleures performances pratiques que les algorithmes d'accélération généraux. Finalement nous avons montré de nouveaux résultats sur les techniques de réduction dédiées aux systèmes à compteurs, notamment qu'elles permettent de s'affranchir du cadre exact de l'accélération plate et de calculer des ensembles d'accessibilité de systèmes non applatissables.

Les résultats sur les compteurs ont été implantés dans l'outil de vérification FAST. Les points forts de FAST sont qu'il manipule des systèmes très expressifs, qu'il termine sur de nombreuses études de cas et qu'il peut s'adapter facilement à d'autres types de calcul (par exemple co-accessibilité, couverture) où ses performances restent honorables. Nous avons discuté les détails d'implantation de FAST et décrit les vérifications paramétrées du protocole TTP et du protocole CES.

Enfin nous avons tenté d'adapter les techniques symboliques à la vérification de programmes à allocation dynamique. Nos résultats sont de trois ordres. Tout d'abord nous avons identifié la classe des systèmes à pointeurs, nous leur avons donné une sémantique précise en termes de graphes mémoire et nous avons exprimé les propriétés à vérifier dans notre formalisme. Ensuite nous avons défini une représentation symbolique adaptée : les états mémoire symboliques. Enfin nous avons proposé une abstraction correcte vis-à-vis des propriétés qui nous intéressent et qui permet de manipuler la sous-classe des états mémoire symboliques sans fuite mémoire. Cette sous-classe a de meilleures propriétés algorithmiques que la classe générale.

Perspectives

La vérification par accélération pose encore de nombreux problèmes.

De manière générale le développement, l'implantation et l'expérimentation de techniques d'accélérations dédiées à de nouveaux domaines seraient souhaitables pour mieux évaluer l'impact réel des techniques d'accélération. Le développement d'algorithmes d'accélération pour les systèmes à pointeurs, l'implantation des cqdd et leur évaluation pratique sont des pistes intéressantes.

L'apport de l'accélération sur les systèmes à compteurs est maintenant

bien établi, et des implantations efficaces existent. On peut donc envisager d'aller plus loin, et se poser comme objectif la vérification par accélération de systèmes manipulant de l'ordre d'une centaine de variables. Trois pistes semblent particulièrement prometteuses : optimiser les bibliothèques d'automates binaires (représentation et opérations), améliorer la recherche de cycles et enfin relâcher le paradigme du calcul exact en mariant abstraction et accélération.

Finalement les résultats de composition d'accélérations s'intègrent naturellement dans un outil générique de vérification quantitative de systèmes hétérogènes. Un tel outil pourrait proposer une interface générique des données et les opérations de composition pour construire à la demande des algorithmes d'accélération adaptés au système à analyser.

Bibliographie

[AAB99] P. A. Abdulla, A. Annichini, and A. Bouajjani. Symbolic verification of lossy channel systems : Application to the bounded retransmission protocol. In *Proc. 5th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'99), Amsterdam, The Netherlands, March 1999*, volume 1579 of *Lecture Notes in Computer Science*, pages 208–222. Springer, 1999.

[AAB00] A. Annichini, E. Asarin, and A. Bouajjani. Symbolic techniques for parametric reasoning about counter and clock systems. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 419–434. Springer, 2000.

[ABS01] A. Annichini, A. Bouajjani, and M. Sighireanu. TReX : A tool for reachability analysis of complex systems. In *Proc. 13th Int. Conf. Computer Aided Verification (CAV'2001), Paris, France, July 2001*, volume 2102 of *Lecture Notes in Computer Science*, pages 368–372. Springer, 2001.

[ACBJ04] P. A. Abdulla, A. Collomb-Annichini, A. Bouajjani, and B. Jonsson. Using forward reachability analysis for verification of lossy channel systems. *Formal Methods in System Design*, 25(1) :39–65, 2004.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, Pei-Hsin Ho, X. Nicollin, A. Olivero, J. Sifakis, and S. Yovine. The algorithmic analysis of hybrid systems. *Theoretical Computer Science*, 138(1) :3–34, 1995.

[AD94] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183–235, 1994.

[Alv] ALV homepage. `http://www.cs.ucsb.edu/~bultan/composite/`.

[BB02] C. Bartzis and T. Bultan. Efficient symbolic representations for arithmetic constraints in verification. Technical Report ucsb cs :TR-2002-16, University of California, Santa Barbara, Computer Science, 2002.

[BB03] C. Bartziz and T. Bultan. Efficient image computation in infinite state model checking. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 249–261. Springer, 2003.

[BB04] C. Bartzis and T. Bultan. Widening arithmetic automata. In *Proc. 16th Int. Conf. Computer Aided Verification (CAV 2004), Boston, Massachussetts , July 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 321–333. Springer, 2004.

[BBR97] B. Boigelot, L. Bronne, and S. Rassart. Improved reachability analysis method for strongly linear hybrid systems. In *Proc. 9th Int. Conf. Computer Aided Verification (CAV '97), Haifa, Israel, June 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 167–178. Springer, 1997.

[BC96] A. Boudet and H. Comon. Diophantine equations, Presburger arithmetic and finite automata. In *Proc. 21st Int. Coll. on Trees in Algebra and Programming (CAAP'96), Linköping, Sweden, April 1996*, volume 1059 of *Lecture Notes in Computer Science*, pages 30–43. Springer, 1996.

[BCvH+03] J. Billington, S. Christensen, K. van Hee, E. Kindler, O. Kummer, L. Petrucci, R. Post, C. Stehno, and M. Weber. The Petri Net Markup Language : Concepts, technology and tools. In *Proc. 24th Int. Conf. Application and Theory of Petri Nets (ICATPN'2003), Eindhoven, The Netherlands, June 2003*, volume 2679 of *Lecture Notes in Computer Science*, pages 483–505. Springer, 2003.

[BEF+00] A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems, and P. Wolper. An efficient automata approach to some problems on context-free grammars. *Information Processing Letters*, 74(5–6) :221–227, 2000.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability analysis of pushdown automata : Application to model-checking. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, July 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 135–150. Springer, 1997.

[BF99] B. Bérard and L. Fribourg. Reachability analysis of (timed) Petri nets using real arithmetic. In *Proc. 10th Int. Conf. Concurrency Theory (CONCUR'99), Eindhoven, The Netherlands, August 1999*, volume 1664 of *Lecture Notes in Computer Science*, pages 178–193. Springer, 1999.

[BF00] J-P. Bodeveix and M. Filali. Experimenting acceleration methods for the validation of infinite state systems. In *Proc. 20th IEEE Int. Conf. in Distributed Computing Systems (IDCS'00), Taipei, ROC, Taiwan, April 2000*, 2000.

[BF04] S. Bardin and A. Finkel. Composition of accelerations to verify infinite heterogeneous systems. In *Proc. of the 2nd International Symposium on Automated Technology for Verification and Analysis (ATVA'2004), Taipei, Taiwan, November 2004*, volume 3299 of *Lecture Notes in Computer Science*, pages 248–262. Springer, 2004.

[BFL+] S. Bardin, A. Finkel, J. Leroux, L. Petrucci, and L. Worobel. *FAST user manual*.

[BFL04] S. Bardin, A. Finkel, and J. Leroux. Faster acceleration of counter automata. In *Proc. 10th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2004) Barcelona, Spain, March 2004*, volume 2988 of *Lecture Notes in Computer Science*, pages 576–590. Springer, 2004.

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST : Fast Acceleration of Symbolic Transition systems. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 118–121. Springer, 2003.

[BFLS05] S. Bardin, A. Finkel, J. Leroux, and P. Schnoebelen. Flat acceleration in symbolic model checking. In *Proc. of the 3rd International Symposium on Automated Technology for Verification and Analysis (ATVA'2005), Taipei, Taiwan, October 2005*, Lecture Notes in Computer Science. Springer, 2005.

[BFN04] S. Bardin, A. Finkel, and D. Nowak. Toward symbolic verification of programs handling pointers. In Ramesh Bharadwaj, editor, *Proceedings of the 3rd International Workshop on Automated Verification of Infinite-State Systems (AVIS'04)*, Electronic Notes in Theoretical Computer Science, Barcelona, Spain, 2004. Elsevier Science Publishers.

[BGP97] T. Bultan, R. Gerber, and W. Pugh. Symbolic model-checking of infinite state systems using Presburger arithmetic. In *Proc. 9th Int. Conf. Computer Aided Verification (CAV'97), Haifa, Israel, June 1997*, volume 1254 of *Lecture Notes in Computer Science*, pages 400–411. Springer, 1997.

[BGP99] T. Bultan, R. Gerber, and W. Pugh. Model-checking concurrent systems with unbounded integer variables : symbolic representations, approximations, and experimental results. *ACM Transactions on Programming Languages and Systems*, 21(4) :747–789, 1999.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *Proc. 4th Int. Symp. Static Analysis, (SAS'97), Paris, France, September 1997*, volume 1302 of *Lecture Notes in Computer Science*, pages 172–186. Springer, 1997.

[BH99] A. Bouajjani and P. Habermehl. Symbolic reachability analysis of FIFO-channel systems with nonregular sets of configurations. *Theoretical Computer Science*, 221(1–2) :211–250, 1999.

[BHJ03] B. Boigelot, F. Herbreteau, and S. Jodogne. Hybrid acceleration using real vector automata. In *Proc. 15th Int. Conf. Computer Aided Verification (CAV'2003), Boulder, CO, USA, July 2003*, volume 2725 of *Lecture Notes in Computer Science*, pages 193–205. Springer, 2003.

[BHMV94] V. Bruyère, G. Hansel, C. Michaux, and R. Villemaire. Logic and p-recognizable sets of integers. *Bull. Belg. Math. Soc.*, 1(2) :191–238, March 1994.

[BHMV05] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying programs with dynamic 1-selector-linked structures in regular model checking. In *Proc. 11th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2005), Edinburgh, UK, April 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 13–29. Springer, 2005.

[BI05a] M. Bozga and R. Iosif. On decidability within the arithmetic of addition and divisibility. In *Proc. 8th Int. Conf. Foundations of Software Science and Computation Structures (FOSSACS'2005), Edinburgh, UK, April 2005*, volume 3441 of *Lecture Notes in Computer Science*, pages 425–439. Springer, 2005.

[BI05b] M. Bozga and R. Iosif. Quantitative verification of programs with lists. In *Proc. of the NATO Advanced Research Workshop*

Verification of Infinite-State systems with Applications to Security (VISSAS'05), Timisoara, Romania, March 2005, NATO Science Series. IOS Press, 2005.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV 2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 403–418. Springer, 2000.

[BM99] A. Bouajjani and R. Mayr. Model checking lossy vector addition systems. In *Proc. 16th Ann. Symp. Theoretical Aspects of Computer Science (STACS'99), Trier, Germany, March 1999*, volume 1563 of *Lecture Notes in Computer Science*, pages 323–333. Springer, 1999.

[BM02] A. Bouajjani and A. Merceron. Parametric verification of a group membership algorithm. In *Proc. 7th Int. Symp. Formal Techniques in Real-Time and Fault-Tolerant Systems (FTRTFT'2002), Oldenburg, Germany, September 2002*, volume 2469 of *Lecture Notes in Computer Science*, pages 311–330. Springer, 2002.

[BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation rewriting and algorithmic verification. In *Proc. 16th IEEE Symp. Logic in Computer Science (LICS 2001), Boston, MA, USA, June 2001*, pages 399–408. IEEE Comp. Soc. Press, 2001.

[Boi98] B. Boigelot. *Symbolic Methods for Exploring Infinite State Spaces*. PhD thesis, Université de Liège, 1998.

[Boi03] B. Boigelot. On iterating linear transformations over recognizable sets of integers. *Theoretical Computer Science*, 309(2) :413–468, 2003.

[Bou01] A. Bouajjani. Languages, rewriting systems, and verification of infinite-state systems. In *Proc. 28th Int. Coll. Automata, Languages, and Programming (ICALP'2001), Crete, Greece, July 2001*, volume 2076 of *Lecture Notes in Computer Science*, pages 24–39. Springer, 2001.

[BP04] S. Bardin and L. Petrucci. From PNML to counter systems for accelerating Petri nets with FAST. In Ekkart Kindler, editor, *Proceedings of the Workshop on Interchange Format for Petri Nets*, pages 26–40, Bologna, Italy, June 2004.

[Bra] Brain homepage. `http://www.cs.man.ac.uk/~voronkov/BRAIN/\-index.html`.

[Bry92] R. E. Bryant. Symbolic boolean manipulation with ordered binary-decision diagrams. *ACM Computing Surveys*, 24(3) :293–318, 1992.

[BW94] B. Boigelot and P. Wolper. Symbolic verification with periodic sets. In *Proc. 6th Int. Conf. Computer Aided Verification (CAV'94), Stanford, CA, USA, June 1994*, volume 818 of *Lecture Notes in Computer Science*, pages 55–67. Springer, 1994.

[BZ83] D. Brand and P. Zafiropulo. On communicating finite-state machines. *Journal of the ACM*, 30(2) :323–342, 1983.

[CC77] P. Cousot and R. Cousot. Abstract interpretation : A unified lattice model for static analysis of programs by construction or approximation of fixpoints. In *Proc. 4th ACM Symp. Principles of Programming Languages, Los Angeles, CA, USA*, pages 238–252, January 1977.

[CFP96] G. Cécé, A. Finkel, and S. Purushothaman Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1) :20–31, 1996.

[CGP99] E. M. Clarke, O. Grumberg, and D. A. Peled. *Model Checking*. MIT Press, 1999.

[CH78] P. Cousot and N. Halbwachs. Automatic discovery of linear restraints among variables of a program. In *Conference Record of the Fifth annual ACM Symposium on Principles of Programming Languages*, pages 84–96. ACM, ACM, January 1978.

[CJ98] H. Comon and Y. Jurski. Multiple counters automata, safety analysis, and Presburger arithmetic. In *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 268–279. Springer, 1998.

[CJ99] H. Comon and Y. Jurski. Timed automata and the theory of real numbers. In *Proc. Conf. on Concurrency theory*, volume 1664 of *Lecture Notes in Computer Science*, pages 242–257, Eindhoven, 1999. Springer.

[Del] Home Page – Giorgio Delzanno. `http://www.disi.unige.it/person/DelzannoG/`.

[Del00a] G. Delzanno. Automatic verification of parameterized cache coherence protocols. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 53–68. Springer, 2000.

[Del00b] G. Delzanno. Verification of consistency protocols via infinite-state symbolic model checking : A case study. In *Proc. IFIP Joint Int. Conf. Formal Description Techniques & Protocol Specification, Testing, and Verification (FORTE-PSTV'00), Pisa, Italy, October 2000*, volume 183 of *IFIP Conference Proceedings*, pages 171–186. Kluwer Academic, 2000.

[DFS98] C. Dufourd, A. Finkel, and P. Schnoebelen. Reset nets between decidability and undecidability. In *Proc. 25th Int. Coll. Automata, Languages, and Programming (ICALP'98), Aalborg, Denmark, July 1998*, volume 1443 of *Lecture Notes in Computer Science*, pages 103–115. Springer, 1998.

[DFV04] C. Darlot, A. Finkel, and L. Van Begin. About Fast and TReX accelerations. In *Proceedings of the 4th International Workshop on Automated Verification of Critical Systems (AVoCS'04)*, Electronic Notes in Theoretical Computer Science, London, UK, August-September 2004. Elsevier Science Publishers.

[DRV04] G. Delzanno, J.-F. Raskin, and L. Van Begin. Covering sharing trees : a compact data structure for parameterized verification. *Journal of Software Tools for Technology Transfer*, 5(2–3) :268–297, 2004.

[EFM99] J. Esparza, A. Finkel, and R. Mayr. On the verification of broadcast protocols. In *Proc. 14th IEEE Symp. Logic in Computer Science (LICS'99), Trento, Italy, July 1999*, pages 352–359. IEEE Comp. Soc. Press, 1999.

[EN98] E. A. Emerson and K. S. Namjoshi. On model checking for non-deterministic infinite-state systems. In *Proc. 13th IEEE Symp. Logic in Computer Science (LICS'98), Indianapolis, IN, USA, June 1998*, pages 70–80. IEEE Comp. Soc. Press, 1998.

[Fas] Fast homepage. `http://www.lsv.ens-cachan.fr/fast/`.

[Fin87] A. Finkel. A Generalization of the Procedure of Karp and Miller to Well Structured Transition Systems. In *Proc. 14th Int. Coll. Automata, Languages, and Programming (ICALP'87), Karlsruhe, FRG, July 1987*, volume 267 of *Lecture Notes in Computer Science*, pages 499–508. Springer, 1987.

[Fin93] A. Finkel. The minimal coverability graph for Petri nets. In *Proc. 12th International Conference on Applications and Theory of Petri Nets (APN'91), Gjern, Denmark, 1991*, volume 674 of *Lecture Notes in Computer Science*, pages 210–243. Springer, 1993.

[FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations : Applications to broadcast protocols. In *Proc. 22nd Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2002), Kanpur, India, December 2002*, volume 2556 of *Lecture Notes in Computer Science*, pages 145–156. Springer, 2002.

[FO97a] L. Fribourg and H. Olsén. A decompositional approach for computing least fixed-points of Datalog programs with Z-counters. *Constraints*, 2(3/4) :305–335, 1997.

[FO97b] L. Fribourg and H. Olsén. Proving safety properties of infinite state systems by compilation into Presburger arithmetic. In *Proc. 8th Int. Conf. Concurrency Theory (CONCUR'97), Warsaw, Poland, July 1997*, volume 1243 of *Lecture Notes in Computer Science*, pages 213–227. Springer, 1997.

[FPS03] A. Finkel, S. Purushothaman Iyer, and G. Sutre. Well-abstracted transition systems : Application to FIFO automata. *Information and Computation*, 181(1) :1–31, 2003.

[Fri00] L. Fribourg. Petri nets, flat languages and linear arithmetic. Invited lecture. In M. Alpuente, editor, *Proc. 9th Int. Workshop on Functional and Logic Programming (WFLP'2000), Benicassim, Spain, September 2000*, pages 344–365, 2000. Proceedings published as Ref. 2000.2039, Universidad Politécnica de Valencia, Spain.

[FS00] A. Finkel and G. Sutre. Decidability of reachability problems for classes of two counters automata. In *Proc. 17th Ann. Symp. Theoretical Aspects of Computer Science (STACS'2000), Lille, France, February 2000*, volume 1770 of *Lecture Notes in Computer Science*, pages 346–357. Springer, 2000.

[GRV04] G. Geeraerts, J-F. Raskin, and L. Van Begin. Expand, Enlarge and Check : new algorithms for the coverability problem of WSTS. In *Proc. 24th Conf. Found. of Software Technology and Theor. Comp. Sci. (FST&TCS'2004), Chennai, India, December 2004*, volume 3328 of *Lecture Notes in Computer Science*, pages 287–298. Springer, 2004.

[GRV05] G. Geeraerts, J-F. Raskin, and L. Van Begin. Expand, enlarge and check... made efficient. In S. K. Rajjamani and K. Etessami, editors, *Poceedings of 17th International Conference on Computer Aided Verification – (CAV 2005)*, number 3576 in Lecture Notes in Computer Science, pages 394–404. Springer, 2005. to appear.

[Hin01] M. Hind. Pointer analysis : haven't we solved this problem yet ? In *ACM SIGPLAN–SIGSOFT workshop on Program analysis for software tools and engineering : June 18–19, 2001, Snowbird, Utah, USA : PASTE'01*, pages 54–61. ACM Press, 2001. Invited talk.

[Iba78] O. H. Ibarra. Reversal-bounded multicounter machines and their decision problems. *Journal of the ACM*, 25 :116–133, 1978.

[ISD+00] O. H. Ibarra, J. Su, Z. Dang, T. Bultan, and R. Kemmer. Counter machines : Decision problems and applications. In *Proc. 25th Int. Symp. Math. Found. Comp. Sci. (MFCS'2000), Bratislava, Slovakia, August 2000*, volume 1893, pages 426–435. Springer, 2000.

[ISD+02] O. H. Ibarra, Jianwen Su, Zhe Dang, T. Bultan, and R. A. Kemmerer. Counter machines and verification problems. *Theoretical Computer Science*, 289(1) :165–189, 2002.

[JEK+90] J.R. Burch, E.M. Clarke, K.L. McMillan, D.L. Dill, and L.J. Hwang. Symbolic Model Checking : 10^{20} States and Beyond. In *Proc. 5th IEEE Symp. Logic in Computer Science (LICS'90), Philadelphia, PA, USA, June 1990*, pages 1–33. IEEE Computer Society Press, 1990.

[KG94] H. Kopetz and G. Grünsteidl. A time trigerred protocol for fault-tolerant real-time systems. In *IEEE computer*, volume January, pages 14–23, 1994.

[Kla04] F. Klaedtke. On the automata size for Presburger arithmetic. In *Proceedings of the 19th Annual IEEE Symposium on Logic in Computer Science (LICS 2004)*, pages 110–119. IEEE Comp. Soc. Press, 2004.

[KMM+01] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1–2) :93–112, 2001.

[KMS02] N. Klarlund, A. Møller, and M. I. Schwartzbach. MONA implementation secrets. *Int. J. of Foundations Computer Science*, 13(4) :571–586, 2002.

[LARSW00] T. Lev-Ami, T. Reps, M. Sagiv, and R. Wilhelm. Putting static analysis to work for verification : A case study. *ACM SIGSOFT Software Engineering Notes*, 25(5) :26–38, 2000.

[Las] Lash homepage. `http://www.montefiore.ulg.ac.be/~boigelot/research/lash/`.

[LAS00] T. Lev-Ami and M. Sagiv. Tvla : A system for implementing static analysis. In *Proc. 7th Int. Symp. Static Analysis (SAS'2000), Santa Barbara, CA, USA, June 2000*, volume 1824 of *Lecture Notes in Computer Science*, pages 280–281. Springer, 2000.

[LB02] L. Liu and J. Billington. Tackling the infinite state space of a multimedia control protocol service specification. In *Proc. 23rd Int. Conf. Application and Theory of Petri Nets (ICATPN'2002), Adelaide, Australia, June 2002*, volume 2360 of *Lecture Notes in Computer Science*, pages 273–293. Springer, 2002.

[Ler03a] J. Leroux. The affine hull of a binary automaton is computable in polynomial time. In *Proc. 5th Int. Workshop on Verification of Infinite State Systems (INFINITY 2003), Marseille, France, Sep. 2003*, volume 98 of *Electronic Notes in Theor. Comp. Sci.*, pages 89–104. Elsevier Science, 2003.

[Ler03b] J. Leroux. *Algorithmique de la vérification des systèmes à compteurs. Approximation et accélération. Implémentation de l'outil Fast.* PhD thesis, Ecole Normale Supérieure de Cachan, Laboratoire Spécification et Vérification. CNRS UMR 8643, décembre 2003.

[Ler05] J. Leroux. A Polynomial-Time Presburger Criterion and Synthesis for Number Decision Diagrams. In *Proc. 20th Annual IEEE Symposium on Logic in Computer Science (LICS'05), Chicago, USA, June 2005*. IEEE Comp. Soc. Press, 2005. to appear.

[LS04] J. Leroux and G. Sutre. On flatness for 2-dimensional vector addition systems with states. In *Proc. 15th Int. Conf. Concurrency Theory (CONCUR'04), London, UK, August-September 2004*, volume 3170 of *Lecture Notes in Computer Science*, pages 402–416. Springer, 2004.

[LS05] J. Leroux and G. Sutre. Flat counter automata almost everywhere ! Technical report, IRISA, Vertecs Project, Campus de Beaulieu, Rennes, France and LaBRI, Univ. de Bordeaux & CNRS UMR 5800, Talence, France, 2005. Draft version to be submitted.

[May81] E. W. Mayr. Persitence of Vector Replacement Systems is decidable. *Acta Informatica*, 15 :309–318, 1981.

[May03] R. Mayr. Undecidable problems in unreliable computations. *Theoretical Computer Science*, 297(1–3) :337–354, 2003.

[Mer74] P. M. Merlin. *A study of the recoverability of computing systems.* PhD thesis, Department of Information and Computer Science, University of California, 1974.

[Mon] MONA homepage. `http://www.brics.dk/mona/index.html`.

[MS77] A. Mandel and I. Simon. On finite semigroups of matrices. *Theoretical Computer Science*, 5(2) :101–111, October 1977.

[MS01] A. Møller and M. Schwartzbach. The pointer assertion logic engine. In *Proceedings of the ACM SIGPLAN'01 Conference on Programming Language Design and Implementation (PLDI-01)*, volume 36.5 of *ACM SIGPLAN Notices*, pages 221–231. ACMPress, June 20–22 2001.

[Pac87] J. K. Pachl. Protocol description and analysis based on a state transition model with channel expressions. In *Proc. 7th IFIP WG6.1 Int. Workshop on Protocol Specification, Testing, and Verification (PSTV '87), Zurich, Switzerland, May 1987*, pages 207–219. North-Holland, 1987.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parameterized verification. In *Proc. 12th Int. Conf. Computer Aided Verification (CAV'2000), Chicago, IL, USA, July 2000*, volume 1855 of *Lecture Notes in Computer Science*, pages 328–343. Springer, 2000.

[Rev90] P. Z. Revesz. A closed form for Datalog queries with integer order. In *Proc. 3rd Int. Conf. Database Theory (ICDT'90), Paris, France, December 1990*, volume 470 of *Lecture Notes in Computer Science*, pages 187–201. Springer, 1990.

[RV02] T. Rybina and A. Voronkov. Brain : Backward reachability analysis with integers. In *Proc. 9th Int. Conf. Algebraic Methodology and Software Technology (AMAST'2002), Saint-Gilles-les-Bains, Reunion Island, France, September 2002*, volume 2422 of *Lecture Notes in Computer Science*, pages 489–494. Springer, 2002.

[SRW02] M. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *ACM Transactions on Programming Languages and Systems*, 24(3) :217–298, May 2002.

[WB98] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. Computer Aided Verification (CAV'98), Vancouver, BC, Canada, June-July 1998*, volume 1427 of *Lecture Notes in Computer Science*, pages 88–97. Springer, 1998.

[WB00] P. Wolper and B. Boigelot. On the construction of automata from linear arithmetic constraints. In *Proc. 6th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2000), Berlin, Germany, March-April 2000*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19. Springer, 2000.

[YKTB01] T. Yavuz-Kahveci, M. Tuncer, and T. Bultan. A library for composite symbolic representations. In *Proc. 7th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2001), Genova, Italy, April 2001*, volume 2031 of *Lecture Notes in Computer Science*, pages 52–66. Springer, 2001.

[Yu97] S. Yu. Regular languages. In A. Salomaa and Grzegorz Rozenberg, editors, *Handbook of Formal Languages*, volume 1, Word Language Grammar, pages 41–110. Springer-Verlag, 1997.