Analyse symbolique de systèmes infinis basée sur les automates: Application à la vérification de systèmes paramétrés et dynamiques

Tayssir Touili

UNIVERSITÉ PARIS 7 - DENIS DIDEROT
UFR D'INFORMATIQUE

THÈSE

pour l'obtention du Diplôme de

DOCTEUR DE L'UNIVERSITÉ PARIS 7
SPÉCIALITÉ : INFORMATIQUE

présentée par

Tayssir TOUILI

Titre :

Analyse symbolique de systèmes infinis basée sur les automates : Application à la vérification de systèmes paramétrés et dynamiques

soutenue publiquement le 21 novembre 2003 devant le jury suivant :

André ARNOLD	Président du jury
Ahmed BOUAJJANI	Directeur de thèse
Jean-Éric PIN	Examinateur
Andreas PODELSKI	Rapporteur
Sophie TISON	Examinatrice
Pierre WOLPER	Rapporteur

A mes parents

A ma chère Nana

A tonton Skander et tata Khaoula

A tata Béchira

A Tayeb et Yassine

Remerciements

Je tiens à remercier très chaleureusement mon directeur de thèse, Monsieur Ahmed Bouajjani, pour sa disponibilité, son soutien constant, ses encouragements, son aide précieuse et ses conseils judicieux qui m'ont permis de mener efficacement à terme ce travail.

Mes plus vifs remerciements vont également à Monsieur André Arnold pour m'avoir fait l'honneur de présider le jury de ma thèse, Messieurs Andreas Podelski et Pierre Wolper pour en être les rapporteurs, ainsi que Monsieur Jean-Eric Pin et Madame Sophie Tison pour avoir accepté de participer à mon jury.

J'aimerais également remercier très sincèrement toutes les personnes avec qui j'ai travaillé ces dernières années. Merci en particulier à Anca Muscholl, Javier Esparza, Markus Muller-Olm, Bengt Jonsson, Parosh Abdulla, Agathe Merceron, Yassine Lakhnech, Monica Maidl, Peter Habermehl, Mihaela Sighireanu, Tomas Vojnar, Marcus Nilsson, Paul Gastin, et Delia Kesner.

Je remercie aussi tous mes collègues de bureau pendant ces trois années de thèse : Marc Boyer, Emmanuel Freund, Benjamin Lermann, Petr Matousek, Richard Mayr, Antoine Meyer, Denis Oddoux, et Tomas Vojnar.

Merci enfin à tous les autres membres du LIAFA. En particulier, mes remerciements vont à Eugène Asarin, Luc Boasson, Olivier Carton, Christian Choffrut, Berke Durak, Blaise Genest, Hugo Gimbert, Irène Guessarian, Isabelle Fagnot, Christiane Frougny, Yan Jurski, Inès Klimann, Matthieu Latapy, Sylvain Lombardy, Jean Mairesse, Clémence Magnien, Anne Micheli, Ha Duong Phan, Christophe Prieur, Dominique Rossin, Olivier Serre, Jean-Baptiste Yunès, Marc Zeitoun, et Wieslaw Zielonka. Sans oublier Noëlle Delgado, Laifa Ahmadi, Alain Courgey, et Zoubir Sami.

Table des matières

Chapitre 1

Introduction

L'automatisation a fait que les systèmes informatiques occupent une place de plus en plus importante dans la vie quotidienne. On les retrouve dans les systèmes de télécommunications, dans la navigation aérienne, dans les centrales nucléaires, etc. Les tâches qu'ils gèrent sont devenues de plus en plus complexes et critiques, dans le sens que la moindre défaillance peut entraîner de graves dégâts financiers et/ou humains.

Il est donc crucial pour les concepteurs de programmes informatiques de disposer de méthodes rigoureuses qui leur permettent de s'assurer que leurs applications remplissent bien les fonctions qui leur sont attribuées, et de détecter les éventuelles erreurs de leurs programmes avant de les mettre en service. Il est bien sûr préférable de détecter ces erreurs le plus tôt possible pour que leur correction soit plus simple et moins coûteuse.

1.1 Les différentes méthodes de validation

Il y a principalement deux techniques complémentaires pour valider les systèmes : *le test* et *la vérification*. Le test et la simulation consistent à tester le système en observant son comportement dans différentes situations. Ces méthodes ne sont pas exhaustives et ne permettent pas de considérer toutes les situations possibles. Elles peuvent donc aider à trouver des erreurs, mais ne garantissent pas la correction du système.

Quant aux méthodes de *vérification*, leur but est de vérifier *rigoureusement* qu'un système est correct dans *toutes* les situations. Ces méthodes nécessitent la définition de modèles mathématiques qui permettent une fidèle représentation du système et des propriétés à vérifier. Ces dernières années, différentes approches de vérification ont été développées :

– L'approche déductive (approche par preuve assistée) qui consiste à montrer, manuellement ou à l'aide de systèmes de déduction et d'aide à la preuve automatiques, que le système considéré satisfait les propriétés voulues. Cette approche consiste à poser le problème de la vérification comme la preuve d'un théorème et

de mener cette preuve en utilisant un système déductif. Des outils basés sur cette approche ont été développés. Nous pouvons citer à titre d'exemple Isabelle/HOL [NPW02], Coq [CH88, HKPM97], et PVS [ORS92]. Cette approche est générale et peut s'appliquer à différents systèmes. Seulement, son inconvénient majeur est qu'elle demande parfois une intervention non triviale et souvent fastidieuse du vérificateur.

– Le model-checking qui consiste à vérifier automatiquement que chaque exécution du programme satisfait la propriété. Cette approche a été introduite dans [QS82, CES83]. Les algorithmes de model-checking sont souvent basés sur (1) la représentation du programme par un modèle précis, qui est souvent un système de transitions étiquetées dont les nœuds représentent les différents états possibles du programme, et dont les arcs décrivent le passage du programme d'un état à un autre ; et (2) la représentation de la propriété par une formule de logique temporelle telles que les logiques temporelles linéaires [Pnu77, Wol86, Wol93], les logiques temporelles arborescentes CTL [CES83] et CTL* [Eme90], ou des calculs de points fixes (μ-calculs) propositionnels [Koz83]. Le model-checking revient alors à vérifier que le modèle du programme satisfait bien la formule de logique temporelle décrivant la propriété. Le résultat de l'analyse (entièrement automatique) est soit la garantie que le programme est correct, soit un contre-exemple montrant une mauvaise exécution en cas de détection d'une erreur. Ces contre-exemples sont particulièrement importants pour corriger les erreurs subtiles dans les systèmes complexes.

1.2 Le model-checking

1.2.1 Les systèmes finis

Cette thèse s'inscrit dans le cadre du model-checking. Dans le cas des systèmes finis (à nombre fini d'états), cette approche est maintenant bien établie. Elle a été largement étudiée et il existe beaucoup d'algorithmes [QS82, CES83, VW86, McM93, Hol94] qui permettent de répondre à la question : "*un système fini satisfait-il une certaine propriété ?*". Seulement, l'inconvénient majeur de cette approche est que la complexité d'un système fait qu'il peut avoir un nombre important d'états possibles. C'est ce qu'on appelle *le problème d'explosion combinatoire des états*. Pour contourner ce problème, différentes techniques réduisant l'espace d'états à considérer ont été adoptées :

– *La vérification à la volée* qui consiste à explorer l'espace des états dans un certain ordre en espérant qu'en cas d'erreur, celle-ci sera trouvée le plus tôt possible [FJJM92, JJ89]. Cette méthode ne permet pas toujours de réduire l'espace d'états exploré.

– *Les techniques d'ordre partiel* [Val91, GW93, God96] qui consistent à limiter l'explosion causée par les actions concurrentes. L'idée consiste à n'explorer qu'une partie des états du système, cette partie étant suffisante pour montrer la satisfaisabilité de la propriété considérée. Par exemple, si le système peut exécuter une ou plusieurs actions dans n'importe quel ordre, et si l'ordre n'affecte pas la satisfaisabilité de la propriété, alors il est inutile d'explorer tous les ordres

possibles ; et il suffit d'explorer un seul ordre arbitraire.

- *Les techniques d'abstraction* [CC77, GL93, Lon93] qui consistent à partitionner l'espace des états. Le modèle abstrait a un état abstrait pour chaque partition, et peut passer d'un état abstrait à un autre ssi les états concrets correspondants sont reliés dans le système réel. Ceci implique en général une perte d'information, et le modèle obtenu est une sur-approximation du système réel. Par exemple, le système abstrait peut utiliser moins de variables que le système concret, et remplacer les arcs où une variable enlevée est testée par le non-déterminisme pour simuler toutes les possibilités qui peuvent avoir lieu. Des erreurs artificielles (qui ne sont pas présentes dans le système concret) peuvent alors apparaître dans le modèle abstrait. Si de telles erreurs sont détectées, l'abstraction doit être raffinée.
- *Les méthodes symboliques* [BCM$^+$92, McM93] qui consistent à représenter de manière succinte les ensembles d'états en utilisant des structures de données dédiées à cet effet, au lieu de les énumérer tous. Dans cette approche, l'espace d'états à explorer n'est pas réduit. L'amélioration réside dans le coût de l'exploration. Parmi les structures de représentation symbolique bien connues, nous pouvons citer les BDDs [Bry92, McM93, BCM$^+$92].

Toutes ces techniques ont été implantées dans des outils de vérification performants qui ont été utilisés avec succès pour la vérification de beaucoup de systèmes industriels. A titre d'exemple, nous citons SPIN [Hol94], SMV [McM98], Murϕ [MDIS96], et CADP [JHA$^+$96].

1.2.2 Les systèmes infinis

Dans cette thèse, nous nous intéressons à la vérification des systèmes infinis (à nombre infini d'états). En effet, les systèmes finis ne sont pas suffisamment puissants pour modéliser certains aspects présents dans les protocoles et les systèmes logiciels tels que :

- La manipulation de variables non bornées telles que les compteurs, les horloges, etc.
- Les structures de contrôle complexes dûes par exemple à la création dynamique de processus et aux appels récursifs de procédures.
- La paramétrisation : beaucoup de systèmes sont définis avec des paramètres. Il est alors nécessaire de pouvoir vérifier leur version paramétrée. A titre d'exemple, plusieurs systèmes (tels que les protocoles d'exclusion mutuelle) sont conçus pour des réseaux pouvant comporter un nombre arbitraire de composantes. De tels systèmes sont appelés *systèmes paramétrés*. Pour valider ces protocoles, il faut vérifier qu'ils sont corrects *quelque soit* le nombre de composantes (qui est ici le paramètre) dans le réseau.

Malheureusement, l'inconvénient majeur de ces systèmes est que leur problème de model-checking n'est pas toujours décidable [EN94, Fin94, HKPV95, CFI96, ACaJT96, AJ96, Esp97a]. Ces dernières années plusieurs efforts ont alors été fournis pour essayer de définir des modèles et des formalismes pour décrire des sous-classes significatives de systèmes infinis, et de proposer des méthodes (semi-)algorihmiques qui permettent d'analyser ces modèles. Nous citons à titre d'exemple les systèmes tem-

porisés et hybrides [AD94, ACD90, HNSY94, ACH+95, AMP95], les systèmes paramétrés [WL89, CGJ95, KMM+97, ABJN99, JN00, PS00], les réseaux de Petri [Jan90, HRY91, Jan97, Esp97a, JM95, Fin94], les systèmes communicants par files d'attentes [PP91, AJ96, AK95, CFI96, BQ96, BG96, ABJ98, AAB99], les systèmes à pile [Cau92, BEM97, EHRS00], etc. Ces techniques sont principalement basées sur la combinaison de deux méthodes :

- *L'abstraction* qui consiste à considérer un système abstrait qui simule le système concret et qu'il est possible d'analyser. Dans beaucoup de cas, l'abstraction calcule un système fini qu'il est possible de valider par les méthodes de vérification présentées précédemment.
- Les *méthodes symboliques* (le model-checking symbolique) basées sur la représentation implicite des ensembles potentiellement infinis de configurations de manière finie en utilisant des structures de représentation symbolique. Par exemple, les DBM sont utilisés pour représenter les ensembles (infinis) de configurations des systèmes à compteurs ou à horloges [Dil89, Yov98], les polyhèdres pour représenter celles des systèmes hybrides [HRP94], les automates finis pour celles des systèmes paramétrés [KMM+97], etc.

1.3 Contributions de la thèse

L'objectif de ce travail est de définir un cadre général basé sur une approche symbolique qui permet la modélisation et la vérification algorithmique uniforme de plusieurs classes de systèmes infinis à l'aide d'automates et de systèmes de réécriture, et d'appliquer ce cadre à la vérification de deux classes de systèmes infinis : les systèmes paramétrés et les programmes récursifs parallèles avec création dynamique de processus.

1.3.1 Un cadre général de vérification basé sur les automates

Des articles tels que [KMM+97, BW98] ont suggéré l'utilisation des automates comme représentation symbolique pour l'analyse des systèmes infinis. Dans cette thèse, nous étendons cette idée vers un cadre général qui permet de raisonner de manière uniforme sur plusieurs classes de systèmes infinis. Nous montrons que ce cadre peut s'appliquer à la vérification des systèmes paramétrés et des programmes récursifs parallèles. Nous considérons le problème du calcul des ensembles des accessibles dans ce cadre, et nous proposons des algorithmes et des techniques d'accélération générales qui permettent le calcul de ces ensembles.

Plus précisément, nous représentons les configurations du système par des mots ou des arbres (de tailles arbitraires), et nous utilisons des classes d'automates de mots ou d'arbres (ou de manière équivalente des classes de langages) pour représenter les ensembles de configurations, et des classes de systèmes de réécriture de mots ou de termes (ou de manière équivalente, des classes de relations ou de transducteurs de mots ou d'arbres) pour représenter les actions du système, c-à-d. les relations de transition entre les configurations. Ensuite, les problèmes de vérification basés sur l'analyse d'accessibilité sont réduits au calcul des clôtures transitives de langages de mots ou

d'arbres par des systèmes de réécriture. C'est-à-dire au calcul de l'ensemble des accessibles $Post^*_{\mathcal{R}}(L)$, où L est un ensemble de configurations (représenté par un automate), et $\mathcal{R}$ est un système de réécriture qui représente les actions du système. Par exemple, dans ce cadre, la vérification d'une propriété de sûreté, exprimant que l'ensemble des "mauvaises" configurations ne doit pas être atteint au cours de l'exécution du système, revient à vérifier que l'intersection entre l'ensemble des accessibles $Post^*_{\mathcal{R}}(L_0)$ (où L_0 est un langage représentant les configurations initiales du système) et celui des mauvaises configurations L_{bad} est vide. C'est ce qu'on appelle l'analyse en avant. Symétriquement, nous pouvons faire une analyse en arrière qui consiste à calculer un langage représentant l'ensemble des prédécesseurs $Pre^*_{\mathcal{R}}(L_{bad})$ des mauvaises configurations, et à vérifier que son intersection avec les configurations initiales L_0 est vide. Ces deux approches étant symétriques, nous considérons l'analyse en avant pour la discussion qui suit. Pour pouvoir effectuer ces tests, les classes d'automates que nous considérons doivent satisfaire certaines propriétés de fermeture et de décidabilité. Plus précisément, nous devont être capables de décider le vide d'une intersection dans ces classes.

Cette approche offre un cadre de vérification général et uniforme qui peut s'appliquer à tous les systèmes dont les ensembles de configurations peuvent être représentés par des mots ou des arbres. D'ailleurs, plusieurs approches existantes spécifiques à des sous-classes particulières de systèmes peuvent être vues comme des instances de ce cadre général. Le *regular model-checking* correspond par exemple au cas où les ensembles de configurations sont représentés par des langages (automates) réguliers de mots ou d'arbres, et les transitions du système sont décrites par des relations régulières (ou transducteurs) de mots ou d'arbres.

Pour illustrer cette modélisation, considérons par exemple le "token passing protocol" qui est un protocole d'exclusion mutuelle défini sur un réseau linéaire pouvant contenir un nombre arbitraire de processus. Il y a un jeton qui circule de gauche à droite parmi les processus, et qui permet l'accès à la section critique. Chaque processus qui détient le jeton le passe (après avoir éventuellement passé à la section critique) à son voisin de droite. Initialement, le jeton est chez le processus le plus à gauche. Un processus a donc deux états : "0" s'il n'a pas le jeton, et "1" s'il l'a. Une configuration du système peut donc être représentée par un mot sur l'alphabet $\{1, 0\}$, où la lettre de la $i^{ème}$ position du mot correspond à l'état du processus numéro i du système. L'ensemble de configurations initiales du système peut donc être représenté par le langage régulier 10^* qui exprime que le premier processus a le jeton, et qu'à sa droite, il y a un nombre arbitraire de processus qui ne l'ont pas. De même, l'action du système qui consiste à faire circuler le jeton de la gauche vers la droite peut être représentée par la règle de réécriture $\mathcal{R} = 10 \rightarrow 01$ qui exprime que le processus qui a le jeton le perd (il passe de l'état 1 à l'état 0), alors que son voisin de droite le récupère (il passe de l'état 0 à l'état 1). L'ensemble des configurations accessibles est donné par $Post^*_{\mathcal{R}}(10^*)$ qui est égal à 0^*10^*. Tout le problème réside dans le calcul *automatique* de cet ensemble des accessibles, c-à-d., dans le calcul de $Post^*_{\mathcal{R}}(L)$ pour un système de réécriture $\mathcal{R}$ et un langage de mots ou d'arbres L (finiment représenté dans une classe d'automates). Ce calcul n'étant en général pas possible même dans le cas régulier (la relation de transition d'une machine de Turing est une relation régulière entre les mots), le problème principal revient à :

– Déterminer des classes significatives de relations et de langages pour lesquelles il est possible de calculer de manière effective $Post^*_{\mathcal{R}}(L)$ pour toute relation $\mathcal{R}$ et tout langage L des classes considérées.
– Trouver des semi-algorithmes généraux basés sur le calcul itératif de l'ensemble des accessibles, mais renforcés par des techniques d'accélération de calcul de point fixe [CC77]. En effet, il est facile de voir qu'une procédure itérative qui consiste à essayer de calculer l'ensemble des accessibles $Post^*_{\mathcal{R}}(L)$ en partant de L et en appliquant itérativement la relation $\mathcal{R}$ ne termine jamais (la procédure termine lorsqu'on aura calculé tous les successeurs, c-à-d. lorsque l'application de $\mathcal{R}$ ne rajoute aucune configuration à l'ensemble calculé). Pour voir ceci, reprenons l'exemple du "token passing protocol", et appliquons cette procédure itérative naïve à l'ensemble des configurations initiales. Nous aurons alors la séquence infinie

$$10^* \xrightarrow{\mathcal{R}} 010^* \xrightarrow{\mathcal{R}} 0010^* \xrightarrow{\mathcal{R}} 00010^*10^* \xrightarrow{\mathcal{R}} 000010^* \xrightarrow{\mathcal{R}} \cdots$$

et le calcul ne terminera jamais. D'où le besoin d'utiliser des techniques d'accélération qui permettent d'augmenter les chances de convergence du calcul itératif en ajoutant à chaque étape non seulement les successeurs par de simples transitions, mais aussi l'effet de *méta-transitions* [BW94] correspondant à des séquences de transitions.

Nous proposons dans ce travail une technique d'accélération semi-algorithmique (dont la terminaison n'est pas garantie) générale qui peut s'appliquer de manière uniforme à l'analyse d'accessibilité de plusieurs classes de systèmes. Cette méthode, appelée *élargissement régulier*, est basée sur la comparaison des automates obtenus par applications successives d'une transformation $\mathcal{R}$ à un automate représentant L dans le but de détecter des croissances dans les structures des automates et deviner automatiquement un automate qui reconnaît la limite $Post^*_{\mathcal{R}}(L)$. Un test permet de déterminer si l'ensemble deviné contient tous les accessibles. Ce même test assure dans plusieurs cas que l'automate calculé représente *exactement* l'ensemble des accessibles. Cette méthode peut également être appliquée pour calculer itérativement les clôtures transitives de certaines relations régulières de mots ou d'arbres.

L'avantage de cette technique d'*élargissement régulier* est qu'elle peut être appliquée uniformément à l'analyse d'accessiblité de tous les systèmes qui peuvent être décrits dans le cadre du regular model-checking, et ce, quelle que soit la forme des relations qui les représentent. Nous illustrons ceci en montrant l'applicabillité de cette méthode dans l'analyse des systèmes paramétrés, et de sous classes de programmes récursifs parallèles. De plus, nous montrons qu'elle est assez puissante pour calculer *précisément* les clôtures transitives pour plusieurs classes significatives de langages et de relations pour lesquelles des algorithmes spécifiques existent dans la littérature.

Cette technique est présentée dans les articles [BJNT00, Tou01, BT02].

Dans ce travail, nous appliquons ce cadre général basé sur les automates à la vérification des (1) systèmes paramétrés, et des (2) programmes récursifs parallèles avec création dynamique de processus. Nos techniques d'élargissement s'appliquent avec succès dans ces cas. De plus, pour avoir des algorithmes d'analyse exacts, directs, et efficaces, nous définissons des classes de relations de mots (resp. d'arbres) et des classes de langages de mots (resp. d'arbres) qui permettent de modéliser ces systèmes,

et pour lesquelles $Post^*_{\mathcal{R}}(L)$ est effectivement calculable pour toute relation $\mathcal{R}$ et tout langage L des classes considérées ; ou, de manière plus générale, nous définissons des classes significatives de relations de mots (resp. d'arbres) pour lesquelles il est possible de caractériser les clôtures transitives $\mathcal{R}^*$ pour toute relation $\mathcal{R}$ de la classe. Nous utilisons, à chaque fois qu'il est possible, les langages réguliers pour analyser ces systèmes. Les programmes récursifs parallèles nous obligent de sortir du cadre régulier et de considérer des classes plus générales d'automates d'arbres.

1.3.2 Vérification des systèmes paramétrés

Rappelons que les systèmes paramétrés sont des systèmes pouvant avoir un nombre arbitraire de processus identiques, chaque processus étant un système fini. Un système paramétré peut être vu comme une famille infinie $S = \{S_n\}_{n=0}^{\infty}$ de systèmes $S_n = P\|P\|...\|P$ formés de n processus mis en parallèle. De tels systèmes apparaissent dans plusieurs contextes. Par exemple, les protocoles qui gèrent les réseaux doivent fonctionner quelque soit le nombre de composantes dans le réseau, etc. Le model-checking de ces systèmes est indécidable [AK86]. Il y a donc deux solutions pour attaquer ce problème : soit se restreindre à des sous-classes décidables, soit considérer des méthodes incomplètes et espérer que pour le système qui nous intéresse, l'une de ces méthodes va pouvoir s'appliquer.

Nous nous plaçons dans le cadre du regular model-checking, c-à-d. nous utilisons les langages réguliers de mots ou d'arbres pour représenter les ensembles de configurations ; et les relations régulières de mots ou d'arbres pour représenter les transitions du système. Pour analyser ces systèmes, nous pouvons appliquer notre méthode semi-algorithmique d'élargissement. De plus, nous proposons des algorithmes spécifiques qui permettent de traiter des classes de relations régulières qui permettent typiquement de modéliser les systèmes où les processus voisins communiquent :

1. Nous considérons tout d'abord les systèmes paramétrés linéaires. Nous considérons la classe des relations de semi-commutations, c-à-d., les relations qui peuvent être représentées par des règles de réécriture de la forme $ab \rightarrow ba$ qui réécrivent ab en ba. Ces règles sont présentes dans la modélisation des systèmes dont les processus communiquent avec leurs voisins (tel que le "token passing protocol" décrit plus haut). Il est bien connu que les langages réguliers ne sont pas fermés par ce genre de relations. Nous définissons alors une classe de langages réguliers de mots (la classe APC) qui apparaît naturellement dans la modélisation des systèmes paramétrés linéaires, et nous montrons que les APC sont effectivement fermés par semi-commutations. Nous montrons ensuite comment ce résultat peut être utilisé pour la vérification des réseaux paramétrés à topologies circulaires. Ces résultats sont publiés dans [BMT01].
2. Nous considérons ensuite les systèmes paramétrés arborescents. Nous définissons la classe des Well Oriented Systems, qui est une classe de relations d'arbres qui permet de modéliser la dynamique des systèmes paramétrés à topologie arborescente où les informations circulent des feuilles vers la racine et vice-versa. Nous donnons une construction qui permet de calculer $\mathcal{R}^*$ pour toute relation $\mathcal{R}$ de la classe. Ce résultat fait l'objet d'une partie de l'article [BT02].

1.3.3 Vérification des programmes récursifs parallèles

Nous considérons les programmes récursifs parallèles avec création dynamique de processus. Nous supposons que les types infinis des données ont déjà été abstraits vers des types finis par les méthodes classiques d'interprétation abstraite [CC77]. Même après une telle abstraction, le problème de la vérification de ces programmes est indécidable en général [Ric53]. Nous obtenons cette indécidabilité dès que les programmes contiennent synchronisation et récursivité [Ram00]. Nous devons alors trouver des modèles analysables qui permettent de représenter des classes significatives de ces programmes.

Ces dernières années, des classes de systèmes de réécriture ont été utilisées pour la modélisation de classes de programmes. Par exemple, les systèmes à pile ont été considérés pour l'analyse des programmes séquentiels [EK99, ES01], les réseaux de Petri pour les programmes concurrents non récursifs [BCR01, DBR02], les systèmes PA pour les programmes sans synchronisation [EK99, EP00], etc. Notre but dans cette thèse est d'étendre et de généraliser les approches existantes vers des modèles plus expressifs qui permettent de tenir compte à la fois de la récursivité, du parallélisme, et de la synchronisation ; et de proposer des méthodes d'analyses symboliques de ces modèles. Comme le problème de la vérification des programmes comprenant tous ces aspects est indécidable, nous sommes obligés de considérer des sur-approximations de l'ensemble des comportements possibles du programme. Dans ce cas, si ces comportements sont *sûrs*, les vrais comportements du système le sont aussi. Nous adoptons dans cette thèse deux approches : (1) nous considérons des modèles qui ont plus de comportements que le vrai programme, et nous proposons des méthodes d'analyse *exactes* de ces modèles. (2) Nous considérons des modèles dont les comportements sont *précisément* ceux du programme, et nous analysons ces systèmes par des techniques approximatives (puisque le problème de l'analyse exacte de ces systèmes est forcément indécidable) basées sur *l'interprétation abstraite* [CC77].

Analyse par calcul des accessibles des systèmes PRS Dans le cadre de la première approche, nous considérons la classe des systèmes de réécriture PRS (Process rewrite Systems) (un formalisme introduit par Mayr [May98]). Un PRS est un ensemble fini de règles de la forme $t \rightarrow t'$, où t et t' sont des termes construits à partir du processus nul "0", un nombre fini de variables (X), la composition séquentielle "$\cdot$", et la composition parallèle asynchrone "||". Les opérateurs "$\cdot$" et "||" sont respectivement associatif et associatif/commutatif. Plusieurs modèles bien connus peuvent être vus comme des sous-classes de PRS tels que les systèmes à pile qui correspondent aux systèmes où la composition parallèle est absente ; les réseaux de Petri où seule la composition parallèle est considérée ; les systèmes PA dont les règles ne contiennent que des variables dans leur partie gauche, c-à-d., elles sont de la forme $X \rightarrow t$; et les systèmes PAD où seule la composition séquentielle est autorisée dans les parties gauches des règles.

Nous proposons une traduction automatisable des programmes vers des PRS qui les modélisent. Les comportements du PRS obtenu sont en général des sur-approximations de ceux du vrai programme. Nous identifions une classe significative de programmes pour laquelle cette traduction est exacte. De même, nous mettons en évidence une

sous classe importante de programmes dont les PRS correspondants peuvent être analysés en oubliant l'associativité/commutativité du "||". Cette classe de programmes comprend la récursivité, le parallélisme, la création dynamique de processus, et la synchronisation. Nous proposons alors des méthodes d'analyse de toute la classe de PRS en oubliant les équivalences structurelles du "||".

Pour les programmes qui ne font pas partie de cette dernière classe, nous proposons une transformation qui permet de les abstraire vers des PADs. Les systèmes obtenus modélisent récursivité et retour de résultats de manière exacte, il n'y a que la synchronisation qui est approximée. Nous proposons alors des méthodes d'analyse exactes qui tiennent compte de l'associativité du "$\cdot$" et de l'associativité/commutativité du "||" pour les systèmes PAD.

Pour attaquer les problèmes d'accessibilité des PRS, il nous faut trouver de "bonnes" classes de représentation symbolique des termes de processus pour lesquelles on sache calculer les ensembles des accessibles. Ceci n'est pas une tâche facile puisque les réseaux de Petri est une sous-classe de PRS, et il est bien connu que l'ensemble des marquages accessibles des réseaux de Petri n'est pas définissable dans la logique de Presburger (il est possible de définir des relations exponentielles entre des variables entières en utilisant des réseaux de Petri [HP79]). Nous serons alors obligés de nous restreindre à des sous-classes de PRS. Pour attaquer ce problème nous proposons deux approches :

• D'abord, nous représentons les termes de processus par des arbres binaires et les ensembles réguliers de termes de processus par des automates d'arbres binaires. Dans ce cas, nous nous heurtons au problème de l'irrégularité de l'ensemble des accessibles à cause de l'associativité et/ou la commutativité des opérateurs. Nous adoptons alors une approche inspirée de [LS98, EP00] qui est basée sur (1) la considération d'équivalences plus fortes entre les termes obtenues en ignorant certaines propriétés d'associativité/commutativité des opérateurs "$\cdot$" et "||" ; et (2) le calcul de représentants des ensembles des accessibles modulo l'équivalence structurelle considérée, c-à-d., le calcul d'un ensemble qui contient au moins un terme de chaque classe d'équivalence, au lieu de calculer tout l'ensemble.

L'idée est basée sur quatres faits. Le premier est que comme nous l'avons mentionné précédemment, nous avons identifié une classe importante de programmes pour laquelle il n'est pas nécessaire de considérer l'associativité/commutativité du "||". Le second est que calculer des représentants de l'ensemble des successeurs (resp. prédécesseurs) est suffisant pour résoudre les problèmes $Post^*_{\mathcal{R}}(\mathcal{L}) \cap \mathcal{L}' = \emptyset$ et $Pre^*_{\mathcal{R}}(\mathcal{L}) \cap \mathcal{L}' = \emptyset$ si $\mathcal{L}'$ est fermé par associativité/commutativité. Le troisième est que calculer les accessibles pour des équivalences plus fortes est plus "facile" (par exemple, régulier et constructible). Et le dernier est que le calcul de représentants peut dans certains cas se ramener au calcul des accessibles modulo des équivalences plus fortes.

Par conséquent, notre but est de calculer les ensembles des accessibles ou leur représentants pour les systèmes PRS modulo les équivalences suivantes : (1) l'égalité entre les termes (=), (2) la relation $\sim_0$ qui tient compte de la neutralité du processus nul "0" par rapport à "$\cdot$" et "||", (3) la relation $\sim_s$ qui considère en plus l'associativité du "$\cdot$", et (4) l'équivalence $\sim$ qui considère aussi l'associativité/commutativité du "||". Etant donnée une de ces équivalences $\equiv$, nous dénotons par $Post^*_{\equiv}$ et $Pre^*_{\equiv}$ les relations

d'accessibilité modulo $\equiv$.

Nous donnons des constructions polynômiales de représentants réguliers des ensembles $Post^*_{\equiv}(\mathcal{L})$ et $Pre^*_{\equiv}(\mathcal{L})$ lorsque l'opérateur "||" n'est pas considéré associatif/commutatif, ces constructions sont valables pour toute la classe PRS. Nous utilisons ces résultats pour l'analyse d'une classe importante de programmes où récursivité, parallélisme, création dynamique de processus, et synchronisation sont considérés.

Lorsque toutes les équivalences sont considérées, nous nous restreignons à la classe PAD. Cette classe est plus générale que celle des systèmes à pile et celle des systèmes PA. L'intérêt principal de cette classe est qu'elle permet de modéliser les programmes parallèles où les procédures peuvent retourner leurs résultats aux procédures appelantes (ceci est possible avec les systèmes à pile, mais pas avec les systèmes PA). Nous montrons que les constructions précédentes permettent de calculer en temps polynômial un représentant de $Post^*_{\equiv}(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$, et un représentant de $Pre^*_{\equiv}(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$ qui est fermé par commutativité du "||". Sinon, dans le cas général où cette propriété de fermeture n'est pas satisfaite, nous n'avons pas réussi à construire un représentant régulier de cet ensemble. A la place, nous construisons un représentant non régulier représenté par un automate d'arbres à compteurs. Heureusement, nous montrons que les automates à compteurs que nous obtenons par notre construction ont de bonnes propriétés, dans les sens que le problème du vide de leur intersection avec les langages réguliers est décidable, ce qui permet de les utiliser comme structure de représentation dans notre cadre.

Ces résultats seront présentés dans [BT03b].

• Notre deuxième approche est basée sur le fait qu'un système PRS peut être vu comme l'union d'un système de réécriture préfixe (ou système de pile), c-à-d. un système dont les règles ne contiennent que l'opérateur "$\cdot$" ; et d'un système de réécriture de multiensembles (ou réseau de Petri), c-à-d. un système dont les règles ne contiennent que l'opérateur "||". Notre but est donc d'étendre et d'intégrer les approches pour l'analyse d'accessibilité des systèmes à pile et des réseaux de Petri pour avoir une procédure générale pour les systèmes PRS. Notre contribution principale est une procédure *générique* qui calcule des automates d'arbres représentant les ensembles des accessibles des systèmes PRS (ou des sur-approximations de ces ensembles) en invoquant des procédures d'analyse des systèmes de réécritures préfixe (système à pile) et celles d'analyses de systèmes de réécriture de multiensembles (réseaux de Petri). La procédure d'analyse des systèmes de réécritures préfixe pouvant être considérée comme fixe (par exemple [Cau92]), la généricité de notre construction vient du fait qu'elle est paramétrée par une procédure d'analyse des systèmes de réécriture de multiensembles qui utilise des ensembles semilinéaires (ou l'arithmétique de Presburger) pour représenter les ensembles de configurations (marquages). Cette procédure peut être soit exacte pour une sous classe de systèmes de réécriture de multiensembles, soit approximative (dans le cas général). Les instantiations de notre construction par des procédures d'accessibilité exactes (resp. approximatives) pour des sous classes de systèmes de réécriture de multiensembles constituent des procédures d'accessibilité exactes (resp. approximatives) pour les classes correspondantes de PRS (la combinaison de ces systèmes avec des systèmes de réécriture préfixe).

Pour définir notre procédure, nous définissons une classe d'automates d'arbres per-

mettant de représenter des ensembles (non réguliers) d'arbres ayant des largeurs non bornées qui sont fermés par commutations des fils de quelques nœuds (ceux correspondant à l'opérateur "||"). Cette classe d'automates satisfait toutes les propriétés de clôture et de décidabilité qui permettent de la considérer comme structure de représentation symbolique dans notre cadre.

Nous pouvons instancier notre procédure aux systèmes PAD pour obtenir un algorithme qui calcule l'ensemble exact des successeurs et des prédécesseurs. Ceci permet le model-checking des formules EF pour PAD. Comme mentionné plus haut, cette classe est plus générale que les systèmes à pile et que les systèmes PA. Ce résultat étend donc tous les résultats existants concernant des sous-classes de PRS.

De plus, nous montrons que notre construction peut être instanciée de différentes manières pour obtenir différentes procédures approchées d'analyse d'accessibilité qui peuvent être appliquées à toute la classe PRS. En fait, notre construction constitue un cadre unificateur où tous les résultats d'analyse d'accessibilité symbolique des réseaux de Petri et des automates à compteurs (ex. [CH78, BW94, DBR01, FL02]) peuvent être intégrées pour analyser les systèmes PRS.

Ces résultats font l'objet du rapport de recherche [BT03a].

Analyse des programmes par calcul d'abstractions des langages de chemins d'exécution Comme nous l'avons mentionné auparavant, les PRS ne permettent de modéliser la synchronisation que dans certaines situations. Nous considérons dans cette deuxième approche des modèles plus expressifs que PRS qui permettent de représenter de manière précise les comportements des programmes.

Nous considérons deux classes de programmes : Nous considérons d'abord les programmes constitués d'un nombre fixe de processus séquentiels concurrents qui peuvent se synchroniser. Nous modélisons ces programmes par des automates à pile communicants. Ensuite, nous étendons notre technique au cas avec création dynamique de processus. Nous modélisons alors les programmes par une algèbre de processus "SPA" qui étend les systèmes PA "à-la CCS" par des opérateurs de synchronisation et de restriction. Contrairement à PRS, ces modèles sont indécidables. Nous sommes donc obligés de considérer des techniques approximatives pour les analyser.

Pour résoudre le problème $Post^*_{\mathcal{R}}(\mathcal{L}) \cap \mathcal{L}' = \emptyset$, notre approche est basée sur le calcul d'abstractions (sur-approximations) des langages de chemins d'exécutions qui mènent de $\mathcal{L}$ à $\mathcal{L}'$, et tester si elles sont vides. Les techniques que nous proposons sont basées sur (1) la représentation des ensembles de configurations par des automates de mots ou d'arbres, (2) l'utilisation de ces automates pour poser un système de contraintes dont la plus petite solution caractérise l'ensemble des chemins d'exécution du programme, et (3) la résolution de ce système de contraintes dans un domaine abstrait. Nous considérons en particulier (1) les abstractions à chaînes finies (c-à-d. dont le domaine ne contient pas de chaînes infinies), et (2) les abstractions commutatives (c-à-d. dont les éléments sont des langages fermés par permutation des lettres de ses mots). Dans le premier cas, le système de contraintes peut être résolu par un calcul itératif, alors que dans le deuxième cas, nous utilisons l'algorithme de résolution de systèmes d'inégalités polynômiales dans les algèbres de Kleene commutatives de [HK99]. Nous proposons différents exemples d'abstractions qui rentrent dans ces deux cadres, et nous

montrons comment ces différentes abstractions fournissent des algorithmes d'analyse ayant différents coûts et différents degrés de précision.

L'avantage de notre approche est sa généricité. En effet, elle offre un cadre algébrique général et uniforme qui peut être instancié par différentes classes d'abstraction pour permettre des analyses dont le coût dépend de la précision désirée.

Ces résultats sont présentés dans les articles [BET03a, BET03b, BET03c].

1.4 Etat de l'art

1.4.1 Regular model-checking

L'idée d'utiliser les automates comme structure de représentation symbolique dans l'analyse d'accessibilité des systèmes infinis a été utilisée dans plusieurs contextes. Nous pouvons par exemple citer les systèmes manipulant des ensembles d'entiers exprimables par des formules de Presburger [BW00], les automates à pile [BEM97, FWW97, EHRS00], les systèmes manipulant des files d'attente avec ou sans perte [BG96, BGWW97, BH97, ABJ98, AAB99], etc. Par exemple, dans [ABJ98, AAB99], il est montré que les ensembles des configurations accessibles des systèmes à files d'attentes avec perte peuvent être effectivement caractérisés par des ensembles réguliers, ou plus précisément, par une sous-classe d'expressions régulières appelée SRE. Tous ces travaux sont cependant spécifiques à des classes particulières de systèmes.

Dans [KMM$^+$97, BW98, Bou01], l'utilisation des automates est proposée comme un cadre uniforme pour l'analyse des systèmes infinis. Dans [KMM$^+$97], ce cadre est proposé pour l'analyse des systèmes paramétrés. Seulement, cet article ne propose pas de techniques d'accélération. Les techniques d'accélération présentées dans [BW98] sont spécifiques à des classes particulières de systèmes qui n'incluent pas les systèmes paramétrés et les programmes récursifs parallèles. A notre connaissance, le premier article à avoir utilisé l'accélération dans le cadre des systèmes paramétrés est [ABJN99]. Les techniques considérées dans cet article sont spécifiques à des classes particulières de systèmes. Plus précisément, les auteurs définissent une classe de relations de mots, que nous appelons dans ce document *context-relations*, qui permet de modéliser certains protocoles d'exclusion mutuelle définis sur des réseaux paramétrés linéaires. Ils calculent la clôture transitive d'une relation de cette classe. Nous montrons au chapitre 9 que notre technique générale d'élargissement permet de simuler cette construction.

La méthode d'accélération définie dans [PS00] ne s'applique qu'aux relations qui préservent les structures, et qui ne réécrivent qu'une ou au plus deux positions à chaque fois. Nos techniques d'élargissement sont plus générales puisqu'elles peuvent s'appliquer à toutes les classes de relations régulières. Les méthodes d'accélération de cet articles ont aussi été utilisées dans [FP01], où des sous-classes d'automates à pile déterministes sont proposées comme cadre uniforme plus général que les automates réguliers pour l'analyse d'accessibilité des systèmes paramétrés linéaires.

Des semi-algorithmes généraux qui permettent, en cas de terminaison, le calcul des clôtures transitives des relations régulières sont présentés dans [BJNT00, JN00, DLS01, AJNd02, BLW03] pour le cas des mots, et dans [DLS01, AJMd02] dans le cas des arbres. Nous donnons dans le chapitre 9 une comparaison détaillée de ces méthodes

avec notre technique d'élargissement.

Il y a beaucoup de travaux sur les systèmes de réécritures de termes qui préservent la régularité. Il est indécidable de savoir si un système de réécriture de termes préserve effectivement la régularité [Gil91]. Plusieurs classes de systèmes de réécriture qui préservent effectivement la régularité ont été identifiées. Parmi ces classes nous citons les systèmes clos [Bra69, DT90], les systèmes monadiques linéaires à droite [Sal88], la classe plus générale des systèmes semi-monadiques linéaires [CDGV94], ou celle plus générale encore des systèmes *finite path overlapping* [TKS00]. L'avantage de notre technique d'élargissement réside dans le fait que ces résultats sont spécifiques à des classes de relations bien particulières, alors que notre méthode peut s'appliquer de manière uniforme à tous les systèmes réguliers pour essayer, en cas de terminaison, de calculer la clôture transitive. D'ailleurs, nous montrons que l'élargissement permet de simuler la construction donnée dans [CDGV94] pour les systèmes semi-monadiques linéaires. Il serait intéressant de voir si notre technique permet également de simuler la construction de [TKS00].

1.4.2 Vérification des systèmes paramétrés

Ces dernières années, il y a eu beaucoup de travaux sur ce thème. Tous ces travaux sont essentiellement basés sur l'une de ces cinq approches :

1. Trouver une constante "cut-off" k telle que si les systèmes contenant un nombre de processus inférieur ou égal à k satisfont la propriété, alors il en est de même des systèmes ayant un nombre plus grand de composantes. C'est l'approche utilisée par German et Sistla dans [GS92], Emerson et Namjoshi dans [EN95, EN96], Emerson et Kahlon dans [EK02], Pnueli et al. dans [PRZ01], et Bouajjani et al. dans [BHV03].
2. L'approche déductive basée sur la recherche d'invariants du système. Il y a deux sortes d'invariants [AJ98] :
   (a) Montrer que la propriété ϕ à satisfaire est un invariant du système, ce qui est équivalent à deviner une sur-approximation de l'ensemble des accessibles et à montrer qu'elle satisfait ϕ [PRZ01, APR$^+$01] ;
   (b) Trouver un *invariant de réseau* [WL89], c-à-d. un processus I dont le comportement "contient" les comportements de tous les systèmes quelle que soit leur taille ; ce qui veut dire que si I satisfait la propriété, alors tous les systèmes S_n de la famille la satisfont. Posons $Q \leq Q'$ si le comportement du processus Q est contenu dans celui du processus Q' ; alors l'invariant I doit satisfaire
   $$P \leq I \quad \text{et} \quad I||P \leq I$$
   Cette approche a été considérée dans [KM89, WL89, BCG89, SG89, HLR97, CGJ95, AJ98].
3. L'abstraction vers un système fini et l'utilisation des techniques classiques de vérification des systèmes finis pour l'analyse [BBLS00].
4. La preuve assistée qui consiste en l'utilisation de "theorem provers" tels que PVS [SOR93], ou l'utilisation de lemmes ou de fonctions d'abstraction fournies par l'utilisateur [JL98, GZ98, MP90].

5. Le regular model-checking. Toutes les techniques présentées dans [KMM+97, ABJN99, JN00, PS00, AJMd02, AJNd02] ont été appliquées aux systèmes paramétrés. Ces techniques ne couvrent pas les résultats de calcul de clôtures transitives décrits dans cette thèse.

Des problèmes liés aux clôtures des langages par semi-commutations ont été étudiés dans la communauté de la théorie des traces. Le chapitre 12 de [DGR95] donne une bonne vue d'ensemble sur les différents résultats trouvés. Le problème que nous considérons ici est différent. Notre but est d'identifier une sous-classe de langages réguliers qui est fermée par *tous* les systèmes de semi-commutations, alors que les résultats classiques de la théorie des traces ont pour but de donner pour une relation de semi-commutations $\mathcal{R}$ donnée, des conditions suffisantes sur les langages réguliers dont la clôture par $\mathcal{R}$ reste régulière. De plus, ces conditions sur les langages dépend souvent de la relation $\mathcal{R}$. La complexité de décider si un langage ω-régulier est fermé par commutations a été considérée dans [Mus96, PWW98].

Les langages APC ont été beaucoup étudiés en logique et en théorie des groupes. Comme nous allons le voir au chapitre 3, ils correspondent au niveau Σ_2 de la hiérarchie de la logique de premier ordre obtenue en comptant le nombre d'alternances des quantificateurs. Plus précisément, ces langages correspondent aux formules de la forme $\exists^*\forall^*\phi$, où ϕ est une formule sans quantificateurs [Tho82]. La classe APC a aussi une caractérisation algébrique. Elle correspond au niveau 3/2 de la hiérarchie de Straubing. De plus, elle correspond au plus haut niveau de la hiérarchie qui est connu d'être décidable [PW97].

1.4.3 Vérification des programmes récursifs parallèles

Ces dernières années, il y a eu beaucoup de travaux sur la vérification des programmes récursifs parallèles. Ces travaux se basent essentiellement sur deux approches complémentaires :

• *L'abstraction* (basée sur l'interprétation abstraite [CC77]). Dans cette approche, les informations concernant les flots de données sont représentées par des éléments d'un treillis approprié. L'analyse revient alors à des calculs de points fixes dans ces treillis. Dans ce cadre, beaucoup de travaux sont dédiés à la vérification des programmes séquentiels où le parallélisme est absent (l'analyse interprocédurale des programmes séquentiels). On peut se référer à la thèse de Knoop [Kno98] et les références qu'elle contient.

C'est après l'article de Taylor [Tay83] que l'analyse des programmes parallèles a commencé à se développer. Dans son article, Taylor a montré que beaucoup de problèmes de vérification sont NP-complets ou même PSPACE-complets même dans la présence du parallélisme, et il a proposé des algorithmes d'analyse généraux pour la résolution de différents problèmes.

Seulement, la majorité des travaux qui traitent le parallélisme ne considèrent pas la récursivité [Cor92, Mer91, NA98, BBS00, Yah01], alors que tous nos modèles (PRS, automates à pile communicants, et SPA) traitent la récursivité de manière exacte.

D'autres travaux considèrent les programmes où la récursivité et la création dynamique de processus sont permis, mais pas la synchronisation entre les composantes

parallèles [Mo02, SS01]. A notre connaissance, le seul travail qui considère à la fois les appels des procédures, la synchronisation, et la création dynamique de processus, est le travail de [DS91]. L'approche proposée dans cet article approxime en même temps l'effet de la synchronisation et celui de la récursivité, alors que tous les modèles que nous considérons dans cette thèse traitent de manière précise la récursion et n'approximent que la synchronisation. Une comparaison plus détaillée de nos techniques avec ces travaux est donnée aux chapitres 7 et 8.

Des références à d'autres travaux de vérification des programmes parallèles se trouvent dans [Rin01].

• *Les méthodes symboliques.* Le lien entre l'analyse statique des programmes et le model-checking a été remarqué dans [Ste91, Sch98, SS98]. Dans [BS95, EK99, ES01], les systèmes à pile ont été proposés comme un modèle d'analyse des programmes séquentiels. Les techniques de vérification des systèmes à pile [Wal96, BS97, BEM97, FWW97, EHRS00], ainsi que les outils dans lesquels ces techniques sont implantées (tels que Bebop [BR00] ou Moped [Sch02]) peuvent alors être utilisés pour l'analyse des programmes séquentiels.

Dans [BCR01, DBR02], le modèle des réseaux de Petri est proposé pour la vérification des programmes parallèles avec synchronisation et création dynamique de processus, mais sans récursion (les processus sont des systèmes finis). Cette approche est généralisée aux programmes avec des communications broadcast dans [FRSB02] en utilisant des réseaux de Petri avec des transitions de transfert. Les techniques d'analyse des réseaux de Petri peuvent donc être utilisées dans ce cadre.

Dans [EK99, EP00], l'algèbre de processus PA est proposée pour l'analyse des programmes sans communication ni entre les processus parallèles, ni entre les processus séquentiels (pas de retour de valeurs). Dans [LS98, EP00], des techniques d'analyse d'accessiblité de ces systèmes sont proposées. Ces techniques sont basées sur le calcul d'automates d'arbres réguliers reconnaissants les ensembles des accessibles.

Nos résultats étendent, généralisent, et uniformisent ces approches vers des modèles plus expressifs qui permettent de considérer des programmes où récursivité, parallélisme, et synchronisation sont présents.

Notre approche qui consiste à faire l'analyse d'accessibilité des PRS en calculant des représentants généralise celle de [LS98, EP00]. Ces travaux considèrent la construction des ensembles des accessibles des systèmes PA sans tenir compte des équivalences structurelles, et montrent que ceci permet d'analyser ces systèmes. Nous généralisons cette approche aux systèmes PRS. Nous identifions une classe importante de programmes que nous pouvons analyser de manière exacte en oubliant l'associativité/commutativité du "||". Ces programmes comprennent la récursivité, le parallélisme, la création dynamique de processus, ainsi que la synchronisation. Quant aux programmes qui ne sont pas dans cette classe, nous les traduisons vers des systèmes PAD, et nous proposons des algorithmes précis d'analyse de ces systèmes. Ceci généralise les résultats de [LS98, EP00] à la classe PAD qui englobe à la fois les systèmes PA et les systèmes à pile, ce qui permet de tenir compte des résultats de retour entre les procédures (ce qui ne peut pas être fait par PA) et du parallélisme (absent dans les systèmes à pile). Nos techniques couvrent les algorithmes relatifs aux automates à pile et aux systèmes PA. Nous donnons des comparaisons plus détaillées avec ces travaux dans les chapitres 5

et 6.

[Esp02] propose aussi une traduction des programmes vers les systèmes PRS sans donner des méthodes d'analyse de ces systèmes. Notre traduction ne traite pas la synchronisation de la même manière que dans [Esp02]. De plus, elle peut s'appliquer à tous les programmes, contrairement à la traduction proposée dans [Esp02]. Nous donnons au chapitre 4 une comparaison plus détaillée des deux transformations.

Dans [JRS03], une approche basée sur le calcul d'abstractions des langages de chemins d'exécutions de systèmes à piles a été utilisée dans le cadre de l'analyse interprocédurale des programmes séquentiels, pour résoudre le problème de propagation de constantes. La technique proposée dans cet article est similaire à l'approche que nous considérons pour la vérification des programmes comprenant un nombre fixe de composants séquentiels concurrents. Nous montrons dans le chapitre 7 que nos techniques s'appliquent aussi dans le cadre considéré dans [JRS03], et nous comparons notre algorithme de calcul des abstractions de chemins d'exécution à celui donné dans cet article.

1.4.4 Analyse des PRS

Les systèmes de réécriture PRS ont été introduits par Mayr dans sa thèse [May98]. Ce dernier a étudié la décidabilité et la complexité de différents problèmes de model-checking de ces systèmes par rapport à des formules de logique temporelle. En particulier, il a montré que le problème d'accessibilité entre deux termes donnés est décidable pour toute la classe de PRS (en considérant toutes les équivalences structurelles). Le problème que nous considérons ici est plus général puisque nous nous intéressons au calcul des ensembles des accessibles, ce qui permet en particulier de résoudre le problème d'accessibilité entre deux termes. Cependant, les résultats que nous proposons sont incomparables à ceux de [May98]. En effet, comme le problème de calculer les accessibles pour PRS est difficile (les PRS contiennent des réseaux de Petri), nous le résolvons pour toute la classe de PRS au cas où les équivalences structurelles du "||" ne sont pas considérées, et nous montrons que notre résultats peut être utilisé dans l'analyse d'une classe importante de programmes. Dans le cas où toutes les équivalences sont considérées, nous résolvons le problème pour la classe PAD, et nous montrons que cette classe est utile dans la modélisation de tous les programmes.

Comme nous l'avons évoqué précédemment, les systèmes de réécriture clos (qui ne comprennent pas de variables) préservent effectivement la régularité [Bra69, DT90]. Cependant, même si les systèmes PRS sont syntaxiquement des systèmes clos, ils ne le sont pas sémantiquement à cause de l'opérateur "$\cdot$" qui impose une stratégie de réécriture préfixe. Donc, même dans le cas où nous oublions les équivalences structurelles entre les termes, nos résultats ne sont pas couverts par [Bra69, DT90]. Par contre, les systèmes PRS sont équivalents aux systèmes de réécriture clos AC obtenus en combinant les systèmes clos avec les lois d'associativité/commutativité d'un opérateur [MR98]. Pour ces systèmes, le problème d'accessibilité entre deux termes est décidable [MR98]. Les auteurs ne proposent cependant pas de construction des ensembles des accessibles.

Beaucoup d'autres travaux sur des sous-classes de PRS portent sur l'étude de différentes équivalences (telle que la bisimulation) entre les structures des graphes

de transitions engendrés par ces classes. Nous citons à titre d'exemple [Hir93, HJM94, JM95, HJM96, Mol96, Sti96]. D'autres références peuvent être trouvées dans [May98].

1.5 Plan de la thèse

Au **chapitre 2.**, nous rappelons d'abord les définitions des notions d'automates et de relations que nous allons utiliser tout au long de cette thèse. Nous définissons ensuite le cadre de vérification général que nous proposons. Le reste de la thèse est ensuite divisé en quatre grandes parties : Le **chapitre 3.**, qui constitue la première partie, est dédié à l'analyse des systèmes paramétrés.

La deuxième partie est consacrée à la vérification des programmes récursifs parallèles en utilisant les systèmes PRS. Le **chapitre 4.** définit ce formalisme, et décrit la traduction que nous proposons des programmes vers ce modèle. Au **chapitre 5.**, nous présentons notre approche d'analyse des PRS par calcul de représentants. Nous donnons différentes constructions (pour la plupart polynômiales) de représentants des ensembles des accessibles. Notre deuxième approche d'analyse de PRS qui consiste à calculer tout l'ensemble des accessibles est présentée au **chapitre 6.**

Dans la troisième partie de cette thèse, nous présentons notre deuxième approche d'analyse des programmes récursifs parallèles, qui est basée sur le calcul d'abstractions des langages de chemins d'exécutions des programmes. Dans le **chapitre 7.**, nous considérons les programmes constitués d'un nombre fixe de processus séquentiels communicants. Nous montrons comment représenter ces programmes par des automates à pile communicants, et nous décrivons notre procédure générique de calcul des abstractions des langages de chemins. Au **chapitre 8.**, nous étendons ces techniques au cas avec création dynamique de processus.

Nos techniques d'élargissement sont décrites dans la dernière partie de ce document, au **chapitre 9.** Finalement, dans le **chapitre 10.**, nous présentons un bilan complet des travaux présentés dans cette thèse, et nous proposons quelques directions de recherches qui permettraient d'étendre ce travail.

En annexes, nous présentons des exemples de protocoles d'exclusion mutuelle définis sur des réseaux paramétrés linéaires. Nous montrons comment modéliser ces protocoles dans notre cadre, et comment appliquer notre méthode d'élargissement pour les vérifier.

Chapitre 2

Un cadre général pour la vérification des systèmes infinis

Dans ce chapitre, nous présentons un cadre général d'analyse d'accessibilité basé sur les automates et les relations de mots et d'arbres. Le cadre que nous proposons est général dans le sens qu'il peut s'appliquer uniformément à plusieurs classes de systèmes infinis, et que beaucoup de travaux existants traitant des classes bien particulières de systèmes peuvent être vus comme des instances de notre cadre. Dans cette thèse, nous appliquons ce cadre à la vérification (1) des systèmes paramétrés et (2) des programmes récursifs parallèles avec création dynamique de processus.

Nous commençons d'abord par rappeler les définitions et les propriétés des automates et des relations réguliers de mots et d'arbres, et nous présentons ensuite le cadre général que nous proposons.

2.1 Préliminaires

Dans cette section, nous introduisons et définissons les outils mathématiques de base et les diverses notations qui seront utilisés tout au long de ce document.

2.1.1 Langages et relations de mots

2.1.1.1 Langages de mots

Définition 2.1.1

- *Un **alphabet** est un ensemble fini de symboles (ou lettres).*
- *Un **mot** w sur un alphabet $\Sigma = \{a_1, ..., a_n\}$ est une séquence de lettres de cet alphabet.*
- *Pour tout mot w sur Σ, pour toute lettre $a \in \Sigma$, $|w|_a$ dénote le nombre d'occurrences de la lettre a dans w.*

- *Nous notons par ε le mot vide.*
- *$a_1 \dots a_n$ est un sous mot de w si w est de la forme $u_0 a_1 u_1 \dots u_{n-1} a_n u_n$, les u_i étant des mots.*
- *Étant donnés deux mots x et y , nous notons par $x.y$ ou xy leur concaténation.*
- *Nous notons par Σ^* l'ensemble de tous les mots sur l'alphabet Σ, et par Σ^+ l'ensemble de tous les mots non vides sur l'alphabet Σ.*

Définition 2.1.2 (Langage) *Nous appelons langage de Σ^* tout sous-ensemble de Σ^*. Nous identifions, quand il n'y a pas d'ambiguïté possible, les langages sur Σ ne comprenant qu'un seul mot par les mots de Σ. Nous écrivons alors a^* au lieu de $\{a\}^*$.*

Définition 2.1.3 (Clôture de Kleene) *Soit L un langage, nous définissons pour tout entier n, L^n par :$L^0 = \{\varepsilon\}$ et $\forall n > 0$; $L^n = L.L^{n-1}$.*

*Nous définissons les opérations de concaténation itérée * et + par :*

$$L^* = \bigcup_{i \geq 0} L^i \qquad et \qquad L^+ = \bigcup_{i > 0} L^i$$

L^* est la clôture de Kleene *de L, et L^+ sa* clôture positive*.*

Définition 2.1.4 (Miroir)
Étant donné un langage L, $Miroir(L) = \{a_1 ... a_n \in \Sigma^* \mid n \geq 1$; $a_n ... a_1 \in L\}$.

Définition 2.1.5 (Langages réguliers)
Soit Σ un alphabet, la classe des **langages réguliers** *d'alphabet Σ est la plus petite classe qui a les propriétés suivantes :*

i) le langage $\emptyset$ est régulier ;

ii) le langage réduit au seul mot vide est régulier ;

iii) pour chaque lettre a de Σ, le langage réduit au seul mot a est régulier ;

iv) si L et L' sont réguliers, il en est de même pour $L \cup L'$, $L.L'$, et L^*.

Définition 2.1.6 (Expression régulière)
- *Une expression régulière sur un alphabet Σ est une expression dont les opérandes sont des lettres de Σ ou le symbole ε et les opérateurs sont pris dans l'ensemble $\{+,.,*\}$, où "+" et "." sont binaires et "*" est unaire.*
- *Une expression régulière décrit un langage régulier sur Σ quand on identifie chaque lettre a de Σ au langage $\{a\}$; ε au mot vide et que l'on interprète "+" comme l'union ; "." comme la concaténation des langages et "*" comme la concaténation itérée.*

Définition 2.1.7 (Automate fini de mots) *Un automate fini de mots est un quintuplet $\mathcal{A}=(Q,I,\Sigma,\delta,F)$ où :*
- *Q est un ensemble fini d'états,*
- *$I \subseteq Q$ est l'ensemble des états initiaux,*
- *Σ est un alphabet fini,*

– $\delta \subseteq Q \times (\Sigma \cup \{\epsilon\}) \times Q$, *est un ensemble fini appelé ensemble des transitions de* $\mathcal{A}$. *Nous pouvons aussi considérer* δ *comme une application de* $Q \times \Sigma \cup \{\epsilon\}$ *vers* Q.
– $F \subseteq Q$ *est l'ensemble des états finaux (accepteurs) de* $\mathcal{A}$.

$\mathcal{A}$ *est dit déterministe si pour tout* $q \in Q$, $\delta(q, \epsilon)$ *n'existe pas, et pour tout* $a \in \Sigma$, *si* $\delta(q, a)$ *existe alors il est réduit à un unique élément.*

Définition 2.1.8 *Soit* $\mathcal{A}$=(Q,I,Σ,δ,F) *un automate fini. Un* ***chemin*** *de* $\mathcal{A}$ *ayant pour longueur n (où* $n \geq 0$*) est une suite de transitions* $((q_i, a_i, q_{i+1}))_{i=0,\ldots,n-1}$. *Ce chemin mène de l'état* q_0 *à l'état* q_n *avec l'étiquette* $a_0 \ldots a_{n-1}$.

Un mot w est reconnu (ou accepté) par $\mathcal{A}$ *s'il existe un chemin étiqueté par w qui mène d'un état initial de I à un état final de F. Des arcs étiquetés par* ϵ *peuvent apparaître au cours de ce chemin, même si les* ϵ *n'apparaissent pas dans w. L'ensemble des mots reconnus par* $\mathcal{A}$ *est noté* $\mathcal{L}(\mathcal{A})$ *: le langage reconnu par* $\mathcal{A}$.

Théorème 2.1.1 *Un langage de mots est régulier si et seulement si il est reconnu par un automate fini de mots.*

Les langages réguliers satisfont toutes les bonnes propriétés de clôture et de décidabilité.

Proposition 2.1.1 *[HU79] La classe des langages de mots réguliers est effectivement fermée par union, intersection, et complémentation. En plus, le problème du vide de ces automates peut être résolu en un temps linéaire.*

2.1.1.2 Relations de mots

Définition 2.1.9 (Relation de mots) *Nous appelons "relation sur* Σ^**" toute partie de* $\Sigma^* \times \Sigma^*$. *Si R est une relation sur* Σ^*, R^{-1} *est la relation définie par :*

$$R^{-1} = \{(w, w') \in \Sigma^* \times \Sigma^* \mid (w', w) \in R\}.$$

Nous notons par Id_{Σ^*} la relation identité sur Σ^* c-à-d., $Id_{\Sigma^*} = \{(w, w) \in \Sigma^* \times \Sigma^*\}$. Une relation R sur Σ préserve les longueurs si pour tout $(w, w') \in R$, w et w' ont la même longueur.

Nous définissons $copy(L)$ la relation qui recopie les mots de L par :

Notation 2.1.1 (*copy(L)*) *Soit un langage* $L \subseteq \Sigma^*$, $copy(L) = \{(w, w) \in \Sigma^* \times \Sigma^* \mid w \in L\}$.

Définition 2.1.10 *Soient* $w \in \Sigma^*$ *et R une relation sur* Σ^*. *Nous notons*

$$R(w) = \{w' \in \Sigma^* \mid (w, w') \in R\}.$$

Cette notion est étendue aux langages de manière standard comme suit : Soit L un langage régulier sur Σ *;*

$$R(L) = \{w' \in \Sigma^* \mid \exists w \in L, (w, w') \in R\}.$$

Définition 2.1.11 (Composition) *La composition des relations sur Σ^* est l'opération sur $\mathcal{P}(\Sigma^* \times \Sigma^*)$ définie par : $\forall R_1, R_2 \in \mathcal{P}(\Sigma^* \times \Sigma^*) : R_2 \circ R_1 = \{(x,y) \in \Sigma^* \times \Sigma^* \mid \exists z \in \Sigma^*; (x,z) \in R_1$ et $(z,y) \in R_2\}$.*

Définition 2.1.12 (Clôture transitive) *Soit R une relation, pour tout $n \geq 0$; nous définissons inductivement la relation R^n par : $R^0 = Id_{\Sigma^*}$ et $\forall n > 0$; $R^n = R \circ R^{n-1}$. L'union $\bigcup_{0 \leq i \leq j} R^i$ est notée par $R^{\leq j}$. Les relations R^* et R^+ sont définies par :*

$$R^* = \bigcup_{i \geq 0} R^i \qquad \text{et} \qquad R^+ = \bigcup_{i > 0} R^i$$

R^* (resp. R^+) est la clôture réflexive transitive (resp. transitive) de R.

Définition 2.1.13 (Transducteur de mots) *Nous appelons transducteur de mots tout quintuplet $\mathcal{T} = (Q, I, \Sigma, \delta, F)$ où :*

- *Q est un ensemble fini d'états,*
- *$I \subseteq Q$ est l'ensemble des états initiaux,*
- *Σ est un alphabet fini,*
- *$\delta \subseteq Q \times \big((\Sigma \cup \{\epsilon\}) \times \Sigma^*\big) \times Q$ est un ensemble fini appelé ensemble des transitions de $\mathcal{T}$,*
- *$F \subseteq Q$ est l'ensemble des états finaux de $\mathcal{T}$.*

Un chemin de $\mathcal{T}$ est une suite de transitions $\Big(\big(q_i, (a_i, b_i), q_{i+1}\big)\Big)_{i=0,\ldots,n-1}$ telles que les a_i sont dans $\Sigma \cup \{\epsilon\}$, et les b_i sont dans Σ^*. Ce chemin mène de l'état q_0 à l'état q_n avec l'étiquette $(a_0 \cdots a_n, b_0 \cdots b_n)$. Si $q_0 \in I$ et $q_n \in F$, cette paire est dite reconnue par le transducteur. Le langage de $\mathcal{T}$, $\mathcal{L}(\mathcal{T})$ est l'ensemble de telles paires.

Un transducteur peut être vu comme un automate avec entrées/sorties. Il lit un mot en entrée, et en produit un autre en sortie. Une transition de la forme $\big(q, (a, w), q'\big)$, où $a \in \Sigma \cup \{\epsilon\}$ et $w \in \Sigma^*$, exprime que si le transducteur lit la lettre $a \in \Sigma$ (ou ne lit rien du tout si $a = \epsilon$) en entrée, il écrit le mot w en sortie. Le langage (ou la relation) reconnu(e) par un transducteur correspond à l'ensemble des paires de mots (w_1, w_2) telles que le transducteur produit w_2 en sortie s'il lit w_1 en entrée. Une telle relation est appelée *relation régulière* :

Définition 2.1.14 (Relation régulière) *Une relation de mots est dite* **régulière** *si elle est reconnue par un transducteur de mots.*

Etant données deux relations régulières, il est facile de construire un transducteur qui reconnaît leur composition :

Définition 2.1.15 (Composition de deux transducteurs)
Soient $\mathcal{T}_1 = (Q_1, I_1, \Sigma, \delta_1, F_1)$ et $\mathcal{T}_2 = (Q_2, I_2, \Sigma, \delta_2, F_2)$ deux transducteurs de mots. Nous définissons le transducteur $\mathcal{T}_2 \circ \mathcal{T}_1$ par : $\mathcal{T}_2 \circ \mathcal{T}_1 = (Q_1 \times Q_2, I_1 \times I_2, \Sigma, \delta, F_1 \times F_2)$ tel que δ est l'ensemble des transitions $\big((q_1, q_2), (a, w), (q_1', q_2')\big)$, où $a \in \Sigma \cup \{\epsilon\}$ et $w \in \Sigma^*$ sont tels qu'il existe $u \in \Sigma^*$ t.q. $\big(q_1, (a, u), q_1'\big) \in \delta_1$ et $\big(q_2, (u, w), q_2'\big)$ est un chemin de $\mathcal{T}_2$.

Il est alors facile de voir que :

Théorème 2.1.2 *Soient R_1 et R_2 deux relations régulières de de Σ^*. Soient $\mathcal{T}_1 = (Q_1, I_1, \Sigma, \delta_1, F_1)$ et $\mathcal{T}_2 = (Q_2, I_2, \Sigma, \delta_2, F_2)$ deux transducteurs qui leur correspondent. Alors la relation $R_2 \circ R_1$ est reconnue par le transducteur $\mathcal{T}_2 \circ \mathcal{T}_1$.*

Définition 2.1.16 (Itération d'un transducteur) *Soit $\mathcal{T}$ un transducteur de mots, $\mathcal{T}^n$ est défini inductivement par : $\mathcal{T}^1 = \mathcal{T}$, et $\mathcal{T}^n = \mathcal{T} \circ \mathcal{T}^{n-1}$.*

Il est clair que $\mathcal{T}^n$ reconnaît R^n, où R est la relation reconnue par $\mathcal{T}$.

De même, étant donnés un langage régulier de mots L et une relation régulière R, il est facile de construire un automate qui reconnaît $R(L)$:

Définition 2.1.17 (Produit d'un transducteur et d'un automate) *Soient $\mathcal{A} = (Q_1, I_1, \Sigma, \delta_1, F_1)$ un automate de mots et $\mathcal{T} = (Q_2, I_2, \Sigma, \delta_2, F_2)$ un transducteur de mots. Nous définissons l'automate $\mathcal{T}(\mathcal{A})$ dont l'ensemble des états initiaux (resp. finaux) est $I_1 \times I_2$ (resp. $F_1 \times F_2$), et comprenant les transitions de la forme $\big((q_1, q_2), w, (q_1', q_2')\big)$[1] telles que :*

- *$\big(q_2, (\epsilon, w), q_2'\big) \in \delta_2$ et $q_1 = q_1'$; ou*
- *il existe $a \in \Sigma \cup \{\epsilon\}$ tel que (q_1, a, q_1') est un chemin de δ_1 et $\big(q_2, (a, w), q_2'\big) \in \delta_2$.*

Nous obtenons alors que :

Théorème 2.1.3 *Soient $\mathcal{A} = (Q_1, I_1, \Sigma, \delta_1, F_1)$ (resp. $\mathcal{T} = (Q_2, I_2, \Sigma, \delta_2, F_2)$) un automate reconnaissant un langage régulier L, (resp. un transducteur reconnaissant une relation régulière R), alors $R(L)$ est reconnu par l'automate $\mathcal{T}(\mathcal{A})$.*

2.1.2 Langages et relations d'arbres

2.1.2.1 Termes et relations d'arbres

Termes Soit Σ un alphabet muni d'une fonction *arité* : $\Sigma \to \mathbb{N}$. Pour $k \geq 0$, Σ_k est l'ensemble des éléments d'arité k. Notons que les ensembles Σ_k ne sont pas forcément disjoints. Les éléments de Σ_0 sont appelés des *constantes*.

Soit $\mathcal{X} = \{x_1, x_2, \ldots\}$ un ensemble dénombrable de symboles appelés *variables*. L'ensemble $T_\Sigma[\mathcal{X}]$ des termes sur Σ et $\mathcal{X}$ est défini inductivement de la manière suivante :

- Si $f \in \Sigma_0$, alors $f \in T_\Sigma[\mathcal{X}]$,
- Si $x \in \mathcal{X}$, alors $x \in T_\Sigma[\mathcal{X}]$,
- Si $k \geq 1$, $f \in \Sigma_k$, et $t_1, \ldots, t_k \in T_\Sigma[\mathcal{X}]$, alors $f(t_1, \ldots, t_k)$ est dans $T_\Sigma[\mathcal{X}]$.

Nous écrivons T_Σ à la place de $T_\Sigma[\emptyset]$. Les termes de T_Σ sont appelés *termes clos*. Un terme t dans $T_\Sigma[\mathcal{X}]$ est *linéaire* si chaque variable apparaît au plus une fois dans t. Un terme dans $T_\Sigma[\mathcal{X}]$ peut être vu comme un arbre étiqueté où un nœud interne

[1] Pour simplifier la présentation, nous donnons cette définition informelle des transitions de $\mathcal{T}(\mathcal{A})$. Pour donner une définition rigoureuse, il faut considérer un ensemble d'états auxilliaires Q, et si $w = a_1 \cdots a_n$, il faut considérer les transitions $\big((q_1, q_2), a_1, p_1\big)$, (p_i, a_{i+1}, p_{i+1}), et $\big(p_{n-1}, a_n, (q_1', q_2')\big)$, où les p_i sont dans Q.

ayant n fils est étiqueté par un symbole de Σ_n, et les feuilles sont étiquetées par des variables et des constantes. Par conséquent, si t est le terme $f(t_1, \ldots, t_k)$, nous posons $racine(t) = f$. Dans ce document, nous confondons en général un terme de $T_\Sigma[\mathcal{X}]$ et l'arbre qui lui correspond.

Un *contexte* C est un terme linéaire de $T_\Sigma[\mathcal{X}]$. Soient $t_1, \ldots, t_n$ des termes de T_Σ, $C[t_1, \ldots, t_n]$ dénote le terme obtenu en remplaçant dans le contexte C l'occurrence de la variable x_i par le terme t_i, et ce pour chaque i tel que $1 \leq i \leq n$. Nous notons parfois C par $C[x_1, \ldots, x_n]$ pour exprimer que C est un contexte ayant n variables.

Nous définissons la frontière $Front$ d'un terme clos comme suit :

- Si $f \in \Sigma_0$, alors $Front(f) = f$,
- Si $k \geq 1$, $f \in \Sigma_k$, et $t_1, \ldots, t_k \in T_\Sigma$, alors

$$Front\big(f(t_1, \ldots, t_k)\big) = Front(t_1) \cdots Front(t_k).$$

La frontière d'un terme est la séquence obtenue en concaténant les étiquettes des différentes feuilles du terme.

Relations d'arbres Nous définissons maintenant les relations d'arbres :

Définition 2.1.18 (Relation d'arbres) *Nous appelons "relation sur T_Σ" toute partie de $T_\Sigma \times T_\Sigma$. Si R est une relation sur T_Σ, R^{-1} est la relation d'arbres définie par :*

$$R^{-1} = \{(t, t') \in T_\Sigma \times T_\Sigma \mid (t', t) \in T\}.$$

Nous notons par Id_{T_Σ} la relation identité sur T_Σ c-à-d., $Id_{T_\Sigma} = \{(t, t) \in T_\Sigma \times T_\Sigma\}$.

Définition 2.1.19 *Soient $t \in T_\Sigma$ et R une relation sur T_Σ. Nous notons*

$$R(t) = \{t' \in T_\Sigma \mid (t, t') \in R\}.$$

Cette notion est étendue aux langages de manière standard comme suit : Soit L un langage régulier d'arbres sur T_Σ,

$$R(L) = \{t' \in T_\Sigma \mid \exists t \in L, (t, t') \in R\}.$$

La composition de deux relations d'arbres est définie comme précédemment pour le cas des mots. De même, la clôture réflexive-transitive d'une relation R est définie par $R^* = \bigcup_{n \geq 0} R^n$, où R^n dénote la composition de R n fois.

2.1.2.2 Automates et transducteurs d'arbres

Dans cette section, nous définissons les automates d'arbres, un formalisme qui permet de représenter de manière finie des ensembles réguliers de termes. La théorie des automates d'arbres est une extension directe de la théorie des automates de mots : un mot peut être vu comme un terme sur un alphabet dont tous les symboles sont unaires. Dans ce travail, nous utilisons principalement des automates d'arbres ascendants non déterministes :

Définition 2.1.20 *Un* **automate d'arbres ascendant** *est un quadruplet* $\mathcal{A} = (Q, \Sigma, F, \delta)$ *où* Q *est un ensemble fini d'états,* Σ *est un alphabet muni d'une fonction d'arité,* $F \subseteq Q$ *est un ensemble d'états finaux, et* δ *est un ensemble de règles de la forme*

$$f\big(q_1, \ldots, q_n\big) \to q \tag{2.1}$$
$$a \to q \tag{2.2}$$
$$q \to q' \tag{2.3}$$

où $a \in \Sigma_0$, $n \geq 1$, $f \in \Sigma_n$, *et* $q_1, \ldots, q_n, q, q' \in Q$.

Si Q *est fini,* $\mathcal{A}$ *est appelé un* automate d'arbres ascendant fini.

Dorénavant, puisqu'aucune confusion n'est possible, nous utiliserons juste le terme *"automate d'arbres"* au lieu de *"automate d'arbres ascendant".*

Soit t un terme clos. Une exécution ascendante de $\mathcal{A}$ sur t est définie comme suit : d'abord, un état est associé à chaque feuille en appliquant les règles (2.2), ensuite, pour chaque nœud, nous devons recueillir les états qui ont été attribués aux fils, et associer un état au nœud lui-même en appliquant les règles (2.1). Plus précisément, si pendant le processus d'attribution des états les sous-termes $t_1, \ldots, t_n$ sont annotés par les états $q_1, \ldots, q_n$, et si la règle $f\big(q_1, \ldots, q_n\big) \to q$ est dans δ, alors le terme $f(t_1, \ldots, t_n)$ est annoté par q. Les règles (2.3) sont appelées les ϵ-règles. Elles permettent d'annoter par l'état q' un nœud qui est déjà annoté par l'état q. Un terme t est accepté si $\mathcal{A}$ arrive à la racine de t avec un état final.

Formellement, nous définissons la relation $\to_\delta$ induite par $\mathcal{A}$ comme suit : Soient t et t' deux termes de $T_{\Sigma \cup Q}$, alors $t \to_\delta t'$ si et seulement s'il existe un contexte $C \in T_{\Sigma \cup Q}[\mathcal{X}]$, et

- n termes clos $t_1, \ldots, t_n \in T_\Sigma$, et une règle $f(q_1, \ldots, q_n) \to q$ dans δ tels que $t = C[f\big(q_1(t_1), \ldots, q_n(t_n)\big)]$ et $t' = C[q\big(f(t_1, \ldots, t_n)\big)]$, ou
- une règle $a \to q$ dans δ, telle que $t = C[a]$ et $t' = C[q]$, ou
- une règle $q \to q'$ dans δ, telle que $t = C[q]$ et $t' = C[q']$.

Soit $\xrightarrow{*}_\delta$ le clôture réflexive-transitive de $\to_\delta$. Un terme t est accepté par un état $q \in Q$ ssi $t \xrightarrow{*}_\delta q$. Soit L_q l'ensemble des termes acceptés par q. Le langage accepté par l'automate $\mathcal{A}$ est $\mathcal{L}(\mathcal{A}) = \bigcup\{L_q \mid q \in F\}$. Un langage d'arbres est régulier s'il est accepté par un automate d'arbres fini.

Proposition 2.1.2 *[CDG+97] La classe des langages d'arbres réguliers est effectivement fermée par union, intersection, et complémentation. En plus, le problème du vide de ces automates peut être résolu en un temps linéaire.*

2.1.2.3 Transducteurs d'arbres

Nous définissons ci-dessous les transducteurs d'arbres, un formalisme qui permet de représenter finiment des relations régulières :

Définition 2.1.21 *Un* **transducteur d'arbres ascendant** *est un quintuplet* $\mathcal{T} = (Q, \Sigma, \Sigma', F, \delta)$ *où* Q *est un ensemble fini d'états,* Σ *et* Σ' *(les ensembles des symboles d'entrée et de sortie) sont des alphabets munis de fonctions d'arité,* $F \subseteq Q$ *est l'ensemble des états finaux, et* δ *est un ensemble de règles de la forme :*

$$f\big(q_1(x_1), \ldots, q_n(x_n)\big) \to q(u), u \in T_{\Sigma'}[\{x_1, \ldots, x_n\}] \tag{2.4}$$

$$q(x) \to q'(u), u \in T_{\Sigma'}[\{x\}] \tag{2.5}$$

$$a \to q(u), u \in T_{\Sigma'} \tag{2.6}$$

où $a \in \Sigma_0$, $n \geq 1$, $f \in \Sigma_n$, $x, x_1, \ldots, x_n \in \mathcal{X}$, *et* $q_1, \ldots, q_n, q, q' \in Q$.

Dans ce document, nous utilisons juste le terme *"transducteur d'arbres"* au lieu de *"transducteur d'arbres ascendant".*

Etant donné un terme t en entrée, $\mathcal{T}$ procède comme suit : il commence par remplacer quelques feuilles en appliquant les règles (2.6). Par exemple, si une feuille est étiquetée par a et la règle $a \to q(u)$ est dans δ, alors a est remplacée par $q(u)$. Ensuite, la substitution se fait de manière ascendante jusqu'à atteindre la racine de l'arbre. Si la règle $f\big(q_1(x_1), \ldots, q_n(x_n)\big) \to q(u)$ est dans δ, alors $\mathcal{T}$ remplace une occurrence du sous-arbre $f\big(q_1(t_1), \ldots, q_n(t_n)\big)$ par le terme $q(u[x_1 \leftarrow t_1, \ldots, x_n \leftarrow t_n])$, où chaque occurrence de la variable x_i dans u est remplacée par t_i. De même, si $q(x) \to q'(u)$ est dans δ, alors $\mathcal{T}$ remplace un terme de la forme $q(t)$ par $q'(u[x \leftarrow t])$, où chaque occurrence de la variable x dans u est remplacée par t. Le calcul continue jusqu'à la racine de t.

Plus précisément, nous définissons la relation $\to_\delta$ induite par $\mathcal{T}$ comme suit : Soient t et t' deux termes de $T_{\Sigma \cup Q}$, alors $t \to_\delta t'$ si et seulement s'il existe un contexte $C \in T_{\Sigma \cup Q}[\mathcal{X}]$, et

- n termes clos $t_1, \ldots, t_n \in T_\Sigma$, et une règle $f(q_1(x_1), \ldots, q_n(x_n)) \to q(u)$ dans δ tels que $t = C[f\big(q_1(t_1), \ldots, q_n(t_n)\big)]$, et $t' = C\Big[q\big(u[x_1 \leftarrow t_1], \ldots, u[x_n \leftarrow t_n]\big)\Big]$, ou
- une règle $a \to q(u)$ dans δ, telle que $t = C[a]$, et $t' = C[q(u)]$, ou
- une règle $q(x) \to q'$ dans δ, telle que $t = C[q(v)]$, et $t' = C\Big[q'\big(u[x \leftarrow v]\big)\Big]$.

Soit $\overset{*}{\to}_\delta$ la clôture réflexive-transitive de $\to_\delta$. Le transducteur $\mathcal{T}$ définit la relation suivante entre les termes : $R_\mathcal{T} = \{(t, t') \in T_\Sigma \times T_{\Sigma'} \mid \exists q \in F, t \overset{*}{\to}_\delta q(t')\}$.

Définition 2.1.22 *Un transducteur est* linéaire *si toutes les parties droites de ses règles sont linéaires (aucune variable n'apparaît plus d'une fois). Une relation d'arbres est* **régulière** *si elle peut être représentée par un transducteur linéaire.*

Nous nous restreignons aux transducteurs d'arbres linéaires puisqu'ils sont fermés par composition, ce qui n'est pas le cas pour les transducteurs d'arbres généraux [Eng75, CDG+97] :

Définition 2.1.23 (Composition de deux transducteurs linéaires) *Soient* $\mathcal{T}_1 = (Q_1, \Sigma, \Sigma', F_1, \delta_1)$ *et* $\mathcal{T}_2 = (Q_2, \Sigma', \Sigma'', F_2, \delta_2)$ *deux transducteurs d'arbres linéaires. Nous définissons le transducteur* $\mathcal{T}_2 \circ \mathcal{T}_1$ *par* $\mathcal{T}_2 \circ \mathcal{T}_1 = (Q_1 \times Q_2, \Sigma, \Sigma'', F_1 \times F_2, \delta)$, *où* δ *est l'ensemble des règles suivantes :*

- $f\big((q_1,p_1)(x_1),\ldots,(q_n,p_n)(x_n)\big) \to (q,q')(t)$ *si :*
  - $f\big(q_1(x_1),\ldots,q_n(x_n)\big) \to q(u[x_{i_1},\ldots,x_{i_k}]) \in \delta_1$, *et*
  - $u[q'_1(x_{i_1}),\ldots,q'_k(x_{i_k})] \xrightarrow{*}_{\delta_2} q'(t)$,

  où $i_1,\ldots,i_k$ *sont des indices disjoints de* $\{1,\ldots,n\}$, *et* $p_k = q'_j$ *si* $k = i_j$ *et* $p_k \in Q_2$ *sinon ;*
- $(q_1,q_2)(x) \to (q'_1,q'_2)(t)$ *si* $q_1(x) \to q'_1\big(u(x)\big) \in \delta_1$ *et* $u\big(q_2(x)\big) \xrightarrow{*}_{\delta_2} q'_2(t)$ *;*
- $(q_1,q_2)(x) \to (q'_1,q'_2)(t)$ *si* $q_1(x) \to q'_1(u) \in \delta_1$, $u \in T_{\Sigma'}$, *et* $u \xrightarrow{*}_{\delta_2} q'_2(t)$, *et ce pour tout* $q_2 \in Q_2$ *;*
- $a \to (q_1,q_2)(u)$ *si* $a \to q_1(u) \in \delta_1$ *et* $u \xrightarrow{*}_{\delta_2} q_2(t)$.

Soit $\mathcal{T}$ un transducteur linéaire, nous définissons comme précédemment l'itération $\mathcal{T}^n$ par la composition du transducteur $\mathcal{T}$ n fois.

Théorème 2.1.4 *Soient* $\mathcal{T}_1 = (Q_1,\Sigma,\Sigma',F_1,\delta_1)$ *et* $\mathcal{T}_2 = (Q_2,\Sigma',\Sigma'',F_2,\delta_2)$ *deux transducteurs d'arbres linéaires. Soient* R_1 *et* R_2 *les relations reconnues par ces transducteurs. Alors la relation* $R_2 \circ R_1$ *est reconnue par le transducteur linéaire* $\mathcal{T}_2 \circ \mathcal{T}_1$.

Preuve : La preuve est donnée dans [DLS01]. Elle est valable pour le cas des transducteurs descendants, mais elle peut être adaptée de manière directe au cas ascendant. □

De plus, les transducteurs linéaires préservent la régularité :

Définition 2.1.24 (Composition d'un transducteur linéaire et d'un automate d'arbres) *Soit* $\mathcal{T} = (Q_2,\Sigma,\Sigma',F_2,\delta_2)$ *un transducteur d'arbres linéaire et* $\mathcal{A} = (Q_1,\Sigma,F_1,\delta_1)$ *un automate d'arbres, nous définissons l'automate d'arbres* $\mathcal{T}(\mathcal{A})$ *comme suit :* $\mathcal{T}(\mathcal{A}) = (Q_1 \times Q_2,\Sigma',F_1 \times F_2,\delta)$, *où* δ *est telle que :*

- δ *contient la dérivation* $u\big((q_{i_1},q'_{i_1}),\ldots,(q_{i_k},q'_{i_k})\big) \xrightarrow{*}_{\delta} (q,q')$ *si*[2] *:*
  - $f\big(q_1(x_1),\ldots,q_n(x_n)\big) \to q \in \delta_1$, *et*
  - $f\big(q'_1(x_1),\ldots,q'_n(x_n)\big) \to q'(u[x_{i_1},\ldots,x_{i_k}]) \in \delta_2$ *;*
- $(q_1,q_2) \to (q'_1,q_2) \in \delta$ *si* $q_1 \to q'_1 \in \delta_1$ *;*
- δ *contient la dérivation* $u\big((q_1,q_2)\big) \xrightarrow{*}_{\delta} (q_1,q'_2)$ *si* $q_2(x) \to q'_2\big(u(x)\big) \in \delta_2$ *;*
- δ *contient la dérivation* $u \xrightarrow{*}_{\delta} (q_1,q'_2)$ *si* $q_2(x) \to q'_2(u) \in \delta_2$, *et* $u \in T_{\Sigma'}$ *;*
- δ *contient la dérivation* $u \xrightarrow{*}_{\delta} (q,q')$ *si* $a \to q \in \delta_1$ *et* $a \to q'(u) \in \delta_2$.

Théorème 2.1.5 (Composition d'un transducteur linéaire et d'un automate d'arbres) *Soit* $\mathcal{T}$ *un transducteur d'arbres linéaire et* L *un langage d'arbres régulier. alors,* $R_{\mathcal{T}}(L)$ *est reconnu par l'automate d'arbres* $\mathcal{T}(\mathcal{A})$.

Preuve :

Il est facile de montrer que dans $\mathcal{T}(\mathcal{A})$ nous avons :

$$L_{(q_1,q_2)} = \{u \in T_{\Sigma'} \mid \exists t \in T_\Sigma, t \in L_{q_1} \text{ et } t \xrightarrow{*}_{\delta_2} q_2(u)\}$$

□

[2] Ici aussi, nous donnons une définition informelle pour simplifier la présentation ; Pour obtenir cette dérivation, nous devons considérer des états auxiliaires qui permettent d'annoter les noeuds internes du terme u.

Nous définissons ci-dessous les *réétiquetages*, une sous-classe de transducteurs d'arbres linéaires qui préserve la structure de l'arbre en entrée.

Définition 2.1.25 (Réétiquetage) *Un transducteur d'arbres est appelé* **réétiquetage** *si ses règles sont de la forme*

$$f\big(q_1(x_1),\ldots,q_n(x_n)\big) \to q\big(g(x_1,\ldots,x_n)\big) \tag{2.7}$$
$$a \to q(b) \tag{2.8}$$
$$q(x) \to q'(x) \tag{2.9}$$

où $f, g \in \Sigma_n$ *et* $a, b \in \Sigma_0$.

La relation définie par un réétiquetage est appelée *relation de réétiquetage*. Notons que les relations de réétiquetages préservent la structure de l'arbre en entrée. En effet, un réétiquetage $(Q, \Sigma, \Sigma', F, \delta)$ peut être vu comme un automate d'arbres sur l'alphabet produit $\Sigma \times \Sigma'$, où $(\Sigma \times \Sigma')_n = \Sigma_n \times \Sigma'_n$. Les règles (2.7) peuvent être représentées par $f/g(q_1,\ldots,q_n) \to q$, les règles (2.8) par $a/b \to q$, et les règles (2.9) par $q \to q'$ (ici nous représentons la paire (a, b) par a/b). Par conséquent, si R est une relation de réétiquetage, et t et t' sont deux termes tels que $t' \in R^*(t)$, alors les termes t et t' peuvent être vus comme deux étiquetages différents du même arbre u. Considérons le terme t/t', un autre étiquetage de l'arbre u défini comme suit : Un nœud n est étiqueté par f/g si n est étiqueté par f (resp. par g) dans t (resp. dans t'). La relation R^* peut être vue comme l'ensemble de tels termes $\{t/t' \mid t' \in R^*(t)\}$ sur $\mathcal{T}_{\Sigma\times\Sigma'}$.

2.1.3 Ensembles semilinéaires, arithmétique de Presburger, et images de Parikh

2.1.3.1 Ensembles semilinéaires

Un *ensemble linéaire* est un ensemble de vecteurs d'entiers de la forme

$$\{\vec{x} \in \mathbb{Z}^m \mid \exists k_1,\ldots,k_n \in \mathbb{Z}, \vec{x} = \vec{v}_0 + k_1\vec{v}_1 \cdots + k_n\vec{v}_n\},$$

où les $\vec{v}_i$ sont des vecteurs dans $\mathbb{Z}^m$. Un *ensemble semilinéaire* est une union finie d'ensembles linéaires.

Théorème 2.1.6 *[GS66] Les ensembles semilinéaires sont effectivement fermés par union, intersection, et complémentation.*

2.1.3.2 Images de Parikh

Etant donné un mot w sur un alphabet $\Sigma = \{a_1,\ldots,a_n\}$, l'image de Parikh de w est le vecteur d'entiers $Parikh(w) = (|w|_{a_1},\ldots,|w|_{a_n})$. Cette définition peut être généralisée aux langages sur Σ de manière standard. Il est bien connu que l'image de Parikh d'un langage hors-contexte est un ensemble semilinéaire [GS66].

2.1.3.3 Arithmétique de Presburger

L'arithmétique de Presburger est la logique de premier ordre sur les entiers munie de l'addition et de l'ordre linéaire standard entre les entiers. Les termes de l'arithmétique de Presburger sont donnés par :

$$t ::= 0 \mid 1 \mid x \mid t - t \mid t + t,$$

où x est une variable appartenant à un ensemble de variables $\mathcal{V}$. L'ensemble des formules de Presburger est défini par :

$$\varphi ::= t \leq t \mid \neg\varphi \mid \varphi \vee \varphi \mid \exists x.\varphi$$

Nous utilisons de manière standard les abbréviations suivantes : $\wedge, \Rightarrow$, et $\forall$.

Etant donnée une formule φ, nous dénotons par $FV(\varphi)$ l'ensemble de ses variables libres. Soit $FV(\varphi) = \{x_1, \ldots, x_n\}$. Un vecteur $\vec{u} = (u_1, \ldots, u_n) \in \mathbb{Z}^n$ satisfait φ, si $\varphi(\vec{u})$ est vraie, où $\varphi(\vec{u})$ est l'expression obtenue en remplaçant dans φ les occurrences des variables x_i par u_i. Nous écrivons alors $\vec{u} \models \varphi$. Chaque formule φ définit un ensemble de vecteurs d'entiers $[\![\varphi]\!] = \{\vec{u} \in \mathbb{Z}^n \mid \vec{u} \models \varphi\}$. Une formule de Presburger φ est dite satisfaisable si $[\![\varphi]\!] \neq \emptyset$.

Il est facile de montrer par induction structurelle sur la formule et en utilisant le théorème 2.1.6 que l'ensemble des vecteurs qui satisfont une formule de Presburger est semilinéaire :

Théorème 2.1.7 *[Har78] Pour toute formule de Presburger φ, $[\![\varphi]\!]$ est un ensemble semilinéaire. De plus, chaque ensemble semilinéaire $\mathcal{S}$ peut être caractérisé par une formule de Presburger φ telle que $\mathcal{S} = [\![\varphi]\!]$.*

Dans la suite de ce document, nous confondons un ensemble semilinéaire et la formule de Presburger qui lui correspond. Nous déduisons du théorème précédent que le problème de la satisfaisabilité d'une formule de Presburger est décidable. Plus précisément, si $|\varphi|$ est le nombre d'opérateurs dans φ, nous avons :

Théorème 2.1.8 *[Pre29, Opp78, FR93] Décider la satisfaisabilité d'une formule de Presburger nécessite au moins un temps non déterministe double exponentiel ($2^{2^{O(|\varphi|)}}$), et peut être résolu en $2^{2^{2^{O(|\varphi|)}}}$ temps et $2^{2^{O(|\varphi|)}}$ espace. Ce problème est NP-complet si la formule est de la forme $\exists x_1 \cdots \exists x_n.\phi$, où ϕ est une formule sans quantificateurs.*

2.2 Un cadre général basé sur les automates

L'utilisation des automates comme structure de représentation dans l'analyse symbolique des systèmes infinis a été proposée par plusieurs articles tels que [KMM+97, BW98]. Nous proposons dans cette section d'étendre cette idée vers un cadre d'analyse général qui permet de faire l'analyse d'accessibilité de plusieurs systèmes infinis de manière uniforme.

2.2.1 Modélisation et problème d'accessibilité

Nous considérons dans cette thèse le problème d'analyse d'accessibilité des systèmes infinis. Comme pour ces systèmes nous devons manipuler en général des ensembles infinis de configurations, nous avons besoin de structures de représentation symboliques qui permettent de représenter de manière finie des ensembles potentiellement infinis de configurations.

Nous considérons ici les systèmes dont les configurations peuvent être modélisées par des mots ou des arbres, et nous proposons d'utiliser des classes de langages (ou d'automates) de mots ou d'arbres pour représenter de manière finie les ensembles infinis de configurations, et les classes de relations (ou de transducteurs) de mots ou d'arbres pour décrire de manière finie comment un ensemble potentiellement infini de configurations peut évoluer en exécutant les actions du système.

Nous considérons les problèmes d'accessibilité suivants : Etant donnés deux ensembles de configurations $\mathcal{L}$ et $\mathcal{L}'$ représentés dans la classe de langages considérée, et une relation $\mathcal{R}$ (de la classe des relations considérée) qui représente la dynamique du système ; décider si

$$Post^*_{\mathcal{R}}(\mathcal{L}) \cap \mathcal{L}' = \emptyset, \tag{2.10}$$

où $Post^*_{\mathcal{R}}(\mathcal{L})$ représente l'ensemble de tous les successeurs de $\mathcal{L}$ par les actions de $\mathcal{R}$ ($Post^*_{\mathcal{R}}(\mathcal{L})$ est par définition égal à $\mathcal{R}^*(\mathcal{L})$) ; ou de manière équivalente, si

$$Pre^*_{\mathcal{R}}(\mathcal{L}') \cap \mathcal{L} = \emptyset, \tag{2.11}$$

où $Pre^*_{\mathcal{R}}(\mathcal{L})$ représente l'ensemble de tous les prédécesseurs de $\mathcal{L}$ par $\mathcal{R}$. Ceci revient à déterminer s'il existe une configuration de $\mathcal{L}$ qui peut atteindre une configuration de $\mathcal{L}'$ par application itérative de $\mathcal{R}$.

Plusieurs problèmes de vérification peuvent être exprimés de cette façon. Par exemple, comme décrit dans l'introduction de la thèse, pour vérifier qu'un système satisfait une propriété de sûreté (une propriété exprimant que toutes les configurations accessibles par le système sont sûres), nous pouvons :

- soit calculer l'ensemble des configurations que le système peut atteindre en partant de ses configurations initiales, et vérifier qu'il est inclus dans l'ensemble des configurations *sûres* : c'est l'analyse en avant,
- soit calculer l'ensemble des prédécesseurs des configurations *non sûres* (dangereuses) (c-à-d., l'ensemble des configurations à partir desquelles une configuration non sûre est atteinte), et vérifier que l'intersection de cet ensemble et de l'ensemble des configurations initiales est vide : c'est l'analyse en arrière.

Si nous représentons un système par un triplet $(\mathcal{L}_0, \mathcal{R}, \mathcal{L}_{bad})$, où $\mathcal{L}_0$ et $\mathcal{L}_{bad}$ représentent respectivement l'ensemble des configurations initiales du système, et celui des mauvaises configurations ; et $\mathcal{R}$ est une relation de transition entre les configurations qui décrit les différentes actions faisant passer le système d'une configuration à une autre ; l'analyse en avant revient à vérifier que

$$Post^*_{\mathcal{R}}(\mathcal{L}_0) \cap \mathcal{L}_{Bad} = \emptyset, \tag{2.12}$$

et l'analyse en arrière revient à vérifier que

$$Pre^*_{\mathcal{R}}(\mathcal{L}_{Bad}) \cap \mathcal{L}_0 = \emptyset. \tag{2.13}$$

2.2.2 Choix des classes de langages et de relations

Le problème qui se pose alors est : quelles classes de relations (ou de transducteurs) et de langages (ou d'automates) faut-il considérer ? La classe la plus naturelle à considérer est la classe des réguliers puisqu'elle vérifie toutes les bonnes propriétés de fermeture par les opérations booléennes et de décidabilité du problème du vide, ce qui permet de décider la satisfaisabilité des tests (2.10) et (2.11). Seulement, les réguliers ne sont malheureusement pas assez expressifs pour pouvoir représenter tous les systèmes intéressants. En effet, les actions de communication entre les différentes composantes du système introduisent souvent de l'irrégularité. Elles ne peuvent souvent pas être représentées par des relations régulières de mots ou d'arbres ; et même dans le cas où une telle représentation est possible, les relations $\mathcal{R}$ obtenues ne préservent pas toujours la régularité, dans le sens que si $\mathcal{L}$ est un langage régulier, $Post^*_{\mathcal{R}}(\mathcal{L})$ (qui est par définition égal à $\mathcal{R}^*(\mathcal{L})$) ne l'est pas forcément. Nous sommes alors parfois obligés d'aller au-delà des réguliers et de considérer des classes de langages plus expressives. Une manière naturelle d'obtenir de telles classes, c-à-d., des classes plus expressives que les réguliers et qui restent finiment représentables, consiste à combiner les automates de mots ou d'arbres classiques avec des contraintes de Presburger. Les CQDDs [BH97] sont des exemples de telles classes. Dans cette thèse, nous définissons de cette manière des classes d'automates d'arbres que nous utilisons pour analyser les programmes récursifs parallèles avec création dynamique de processus.

Ces classes considérées doivent cependant satisfaire certaines contraintes. En effet, pour pouvoir tester la satisfaisabilité du test (2.10), une exigence minimale consiste à considérer trois classes de langages (finiment représentables par des classes d'automates) $\mathcal{C}_1$, $\mathcal{C}_2$, et $\mathcal{C}_3$ qui ne sont pas forcément les mêmes, telles que $\mathcal{L}$, $Post^*_{\mathcal{R}}(\mathcal{L})$, et $\mathcal{L}'$ soient respectivement représentables dans $\mathcal{C}_1$, $\mathcal{C}_2$, et $\mathcal{C}_3$; et telles qu'en plus il est possible de décider le vide d'une intersection de deux langages appartenant respectivement à $\mathcal{C}_2$ et $\mathcal{C}_3$. Il est souvent naturel de considérer une seule classe de langages $\mathcal{C}$ dédiée à la représentation des ensembles de configurations, c-à-d., de représenter $\mathcal{L}$, $\mathcal{L}'$, et $Post^*_{\mathcal{R}}(\mathcal{L})$ dans la même classe $\mathcal{C}$. Dans ce cas, pour pouvoir tester (2.10), on doit savoir décider le vide d'une intersection de deux langages de $\mathcal{C}$. Mais, dans beaucoup de cas, il n'est pas possible de représenter $Post^*_{\mathcal{R}}(\mathcal{L})$ dans cette même classe $\mathcal{C}$. Il faut alors caractériser cet ensemble de configurations dans une classe $\mathcal{C}'$ telle que l'on sache décider le vide de l'intersection de deux langages appartenant à $\mathcal{C}$ et $\mathcal{C}'$, respectivement. Des exigences similaires peuvent être formulées si nous voulons tester la satisfaisabilité de (2.11).

2.2.3 Calcul des accessibles

Outre ce problème de trouver de bonnes classes de représentation symbolique, le second problème principal qui se pose pour pouvoir résoudre les problèmes (2.10) et (2.11) est de pouvoir calculer l'ensemble des successeurs $Post^*_{\mathcal{R}}(\mathcal{L})$, ou celui des prédécesseurs $Pre^*_{\mathcal{R}}(\mathcal{L})$ pour un langage $\mathcal{L}$ donné. Nous considérons dans la discussion qui suit le problème de calculer $Post^*_{\mathcal{R}}(\mathcal{L})$, le calcul de $Pre^*_{\mathcal{R}}(\mathcal{L})$ étant symétrique.

Le calcul de $Post^*_{\mathcal{R}}(\mathcal{L})$ n'est en général pas possible puisque la relation de transition d'une machine de Turing est une relation régulière entre les mots. Il faut donc soit

trouver des algorithmes de calcul de cet ensemble spécifiques pour des classes particulières de langages et de relations ; soit trouver des semi-algorithmes qui s'appliquent à tous les systèmes mais dont la terminaison n'est pas garantie.

Dans le premier cas, il s'agit de trouver des classes de relations de mots (resp. d'arbres) $\mathcal{C}_R$ et des classes de langages de mots (resp. d'arbres) $\mathcal{C}$ et $\mathcal{C}'$ comme décrites précédemment, c-à-d., pour lesquelles nous savons décider si l'intersection d'un langage de $\mathcal{C}$ avec un langage de $\mathcal{C}'$ est vide ou non ; telles que pour tout langage $\mathcal{L}$ de la classe considérée $\mathcal{C}$ et toute relation $\mathcal{R}$ de la classe $\mathcal{C}_R$, $Post^*_{\mathcal{R}}(\mathcal{L})$ est effectivement caculable et représentable dans la classe de langages $\mathcal{C}'$.

Dans le cas de l'approche semi-algorithmique, une méthode naïve consiste à essayer de calculer l'ensemble des accessibles $Post^*_{\mathcal{R}}(\mathcal{L})$ itérativement en considérant la séquence croissante $(X_i)_{i\geq 0}$ telle que

$$\begin{aligned} X_0 &= \mathcal{L} \\ X_{i+1} &= X_i \cup Post_{\mathcal{R}}(X_i) \end{aligned}$$

Le calcul termine quand nous arrivons à un indice k tel que $X_{k+1} \subseteq X_k$. Dans ce cas, $Post^*_{\mathcal{R}}(\mathcal{L})$ est égal à X_k. Cette procédure commence à partir de l'ensemble de configurations initiales, et rajoute itérativement à cet ensemble les configurations atteintes après une étape d'application de $\mathcal{R}$. Le calcul s'arrête quand aucune nouvelle configuration ne peut être rajoutée. Pour que ces itérations puissent être appliquées, la classe $\mathcal{C}$ doit satisfaire des conditions supplémentaires : elle doit être effectivement fermée par union et $Post_{\mathcal{R}}$. De même, pour que cette procédure puisse converger, le test d'inclusion doit être décidable dans $\mathcal{C}$.

Seulement, pour les systèmes infinis, cette procédure naïve ne termine pas dans presque tous les cas intéressants. Pour augmenter les chances de terminaison de cette procédure, nous appliquons le principe d'*accélération* qui consiste à accélérer le calcul du point fixe en suivant principalement deux approches :

– L'*élargissement* tel que défini dans la communauté de l'interprétation abstraite [CC77] qui consiste à appliquer la procédure ci-dessus en essayant de deviner la limite à partir des premiers ensembles calculés. Ceci se fait en général en comparant les langages calculés aux premières étapes pour essayer de détecter des croissances et extrapoler à chaque fois qu'une croissance est détectée. Plus précisément, ce principe consiste à définir un opérateur d'élargissement ∇ et à calculer la séquence croissante $(Z_i)_{i\geq 0}$ telle que

$$\begin{aligned} Z_0 &= \mathcal{L} \\ Z_{i+1} &= Z_i \cup \nabla\big(Z_i, Post_{\mathcal{R}}(Z_i)\big) \end{aligned}$$

Le but de l'opérateur ∇ est de deviner les croissances et d'extrapoler. Telles que définies dans [CC77], les techniques d'élargissement convergent toujours, et calculent le plus souvent une sur-approximation de l'ensemble des accessibles. De plus, Cousot et al. ont introduit ce principe de manière abstraite et l'ont appliqué au cas où les configurations sont des ensembles d'entiers ou de réels [CH78]. Ils n'ont pas défini cet opérateur dans notre cadre où les configurations sont des langages de mots ou d'arbres. Nous proposons dans le chapitre 9 des

techniques d'extraplation sur les automates réguliers de mots et d'arbres appelées *élargissement régulier* qui définissent cet opérateur ∇ dans notre cadre. A la différence du principe introduit dans [CC77], la méthode que nous proposons ne termine pas toujours puisque l'extrapolation ne se fait pas aveuglément à chaque fois qu'une croissance est détectée. Nous utilisons un test qui permet de décider si l'ensemble calculé est un invariant ou pas, et l'extrapolation n'a lieu que si le test réussit. Ce test permet aussi dans certains cas de garantir que l'ensemble calculé est *exactement* égal à l'ensemble des accessibles. Nous montrons à travers des exemples que notre technique permet de traiter de manière exacte plusieurs cas intéressants. De même, nous montrons qu'elle peut simuler plusieurs constructions de $Post^*_{\mathcal{R}}(\mathcal{L})$ spécifiques à des classes de langages et de relations particulières. L'avantage de notre technique réside dans le fait qu'elle peut s'appliquer de manière uniforme à pratiquement tous les systèmes modélisés par des langages et des relations réguliers. Certes, la terminaison n'est pas toujours garantie, mais dans tous les cas que nous avons considérés, cette méthode s'est bien comportée.

– L'accélération par ajout de *méta-transitions* [BW94] qui consiste à ajouter à chaque étape, quand c'est possible, non seulement les successeurs immédiats par application d'une transition, mais aussi des *méta-transitions* permettant de calculer les successeurs obtenus par application d'un nombre arbitraire de transitions (ou de séquences de transitions). Cette technique d'accélération consiste à utiliser des algorithmes spécifiques pour calculer les successeurs par des sous-ensembles de transitions de $\mathcal{R}$ pour aider la procédure ci-dessus à converger. Typiquement, si $\mathcal{R}$ est donnée sous forme d'un ensemble de relations $\mathcal{R} = \mathcal{R}_1 \cup \cdots \cup \mathcal{R}_n$, il s'agit de regrouper les $\mathcal{R}_i$ en $m+1$ relations $\mathcal{R} = \mathcal{R}' \cup \mathcal{R}'_1 \cup \cdots \cup \mathcal{R}'_m$ telles que pour chaque i, $1 \leq i \leq m$ et pour chaque ensemble $\mathcal{L}$ de $\mathcal{C}$, $Post^*_{\mathcal{R}'_i}(\mathcal{L})$ est effectivement calculable et représentable dans $\mathcal{C}$ (dans ce cas, $\mathcal{C}$ est dite effectivement fermée par $\mathcal{R}'_i$), c-à-d., que l'on ait des algorithmes spécifiques qui permettent de calculer ces ensembles. Et donc le principe de l'accélération dans ce cas consiste à calculer la séquence croissante $(Y_i)_{i\geq 0}$ telle que

$$\begin{aligned} Y_0 &= \mathcal{L} \\ Y_{i+1} &= Y_i \cup Post_{\mathcal{R}'}(Y_i) \cup Post^*_{\mathcal{R}'_1}(Y_i) \cup \cdots \cup Post^*_{\mathcal{R}'_m}(Y_i) \end{aligned}$$

Observons que dans ce cas, il est important que les ensembles $Post^*_{\mathcal{R}'_i}(\mathcal{L})$ soient également dans la classe $\mathcal{C}$ pour que l'on puisse appliquer à chaque étape ces algorithmes spécifiques aux relations $\mathcal{R}'_i$. Le calcul termine quand nous arrivons à un indice k tel que $Y_{k+1} \subseteq Y_k$. Dans ce cas, $Post^*_{\mathcal{R}}(\mathcal{L})$ est égal à Y_k. Cette procédure ne termine pas toujours, mais elle a plus de chances de converger que la procédure précédente qui calcule les ensembles X_i.

Pour maximiser les chances de convergence de notre semi-algorithme, nous pouvons combiner ces deux approches, ce qui revient à calculer l'effet exact d'une méta-transition à chaque fois qu'on sait le faire, et extrapoler à chaque fois qu'une croissance est détectée, et que notre test d'invariance est satisfait. Ceci revient à calculer

la séquence croissante $(W_i)_{i\geq 0}$ telle que

$$\begin{aligned} W_0 &= \mathcal{L} \\ W_{i+1} &= W_i \cup \nabla\big(W_i, Post_{\mathcal{R}'}(W_i) \cup Post^*_{\mathcal{R}'_1}(W_i) \cup \cdots \cup Post^*_{\mathcal{R}'_m}(W_i)\big) \end{aligned}$$

2.2.4 Applications de ce cadre général

Ce cadre général peut s'appliquer de manière uniforme à tous les systèmes dont les configurations peuvent être représentées par des mots ou des arbres. D'ailleurs, plusieurs travaux existants spécifiques à des systèmes bien particuliers qui utilisent des structures de représentation symboliques basées sur des classes d'automates finis, éventuellement combinés avec des contraintes de Presburger, peuvent être vus comme des instances de ce cadre. A titre d'exemple, nous pouvons citer les travaux de Boigelot et al. sur les systèmes à files d'attente parfaits (sans pertes) qui considèrent des structures de représentation (QDD) basées sur les automates finis [BG96, BGWW97]. Bouajjani et Habermehl ont proposé des structures plus précises (CQDD), basées sur une combinaison d'automates finis avec des contraintes arithmétiques linéaires [BH97]. Pour les systèmes à files d'attente avec perte, Abdulla et al. utilisent une classe particulière d'expressions régulières (SRE) [ABJ98, AAB99]. [FP01] propose une classe d'automates à pile déterministes pour l'analyse des systèmes paramétrés. Les ensembles de configurations des protocoles cryptographiques peuvent être modélisés par des classes d'automates d'arbres [Mon02, GL00, CCM01, GK00]. Les algorithmes de calcul des ensembles des accessibles des automates à pile présentés par exemple dans [BEM97, FWW97, EHRS00] rentrent aussi dans ce cadre.

Le cas particulier où les langages réguliers sont utilisés comme représentation symbolique correspond au *regular model checking* [KMM+97, BW98, ABJN99, BJNT00]. Ce cadre a, par exemple, été appliqué de manière intense à la vérification des systèmes paramétrés. Notre méthode générale d'*élargissement régulier* contribue au développement de ce cadre. Nous citons également ci-dessous des algorithmes spécifiques que nous proposons et qui peuvent être utilisés pour calculer des méta-transitions dans ce cadre régulier.

Nous proposons dans ce travail d'appliquer ce cadre général à la vérification des (1) systèmes paramétrés, et des (2) programmes récursifs parallèles avec création dynamique de processus. Nous nous plaçons, à chaque fois qu'il est possible, dans le cadre du regular model checking ; c-à-d. nous utilisons les langages réguliers pour représenter les ensembles de configurations de ces systèmes à chaque fois que nous pouvons le faire. Les programmes récursifs parallèles nous obligent de sortir du cadre régulier et de considérer des classes plus générales de langages d'arbres. Nous définissons alors de nouvelles classes d'automates d'arbres qui combinent les automates d'arbres ascendants finis avec les contraintes de Presburger, et qui ont de bonnes propriétés de fermeture et de décidabilité qui permettent de résoudre les problèmes (2.10) et (2.11).

Dans le cas des systèmes paramétrés linéaires, une configuration peut être représentée par un mot dont la $i^{ème}$ lettre correspond à l'état du $i^{ème}$ processus. Nous représentons alors les ensembles de configurations potentiellement infinis par des langages réguliers de mots (voir [KMM+97] et le chapitre 3), et les transformations du système par des

relations régulières entre les mots. De même, pour les réseaux paramétrés à topologie arborescente, une configuration peut être représentée par un terme (ou un arbre) dont l'étiquette d'un nœud correspond à l'état du processus correspondant à ce nœud. Nous représentons alors un ensemble de configurations par un langage régulier d'arbres, et les actions du systèmes par une relation de réétiquetage (voir [KMM$^+$97] et le chapitre 3). Nous montrons dans le chapitre 9 que notre technique générale d'élargissemenent régulier peut s'appliquer dans ce cadre pour calculer de manière exacte les ensembles des accessibles. De plus, nous proposons des algorithmes spécifiques à des classes particulières de langages et de relations réguliers qui permettent de modéliser ces systèmes paramétrés :

2.2.4.1 Vérification des systèmes paramétrés

Dans le chapitre 3, nous considérons la classe des relations de semi-commutations, c-à-d., les relations qui sont une union de relations de la forme

$$copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$$

Ces relations permettent de modéliser les communications entre les processus voisins dans les systèmes paramétrés linéaires. Il est bien connu que les langages réguliers ne sont pas fermés par ce genre de relations. Nous définissons alors une classe de langages réguliers de mots (la classe APC) qui apparaît naturellement dans la modélisation des systèmes paramétrés linéaires, et nous montrons que les APC sont effectivement fermés par semi-commutations. Puisque dans ce cas les classes $\mathcal{C}$ et $\mathcal{C}'$ sont toutes les deux égales à la classe APC, nous montrons comment nous pouvons utiliser notre algorithme pour le calcul de méta-transitions (lors de l'application de la procédure itérative donnée précédemment) pour analyser un contrôleur d'ascenceur décrit par une relation de la forme $\mathcal{R} \cup \mathcal{R}'$ où $\mathcal{R}$ est une relation de semi-commutations et $\mathcal{R}'$ est une relation quelconque.

Dans ce même chapitre, nous définissons la classe des Well Oriented Systems, qui est une classe de relations de réétiquetage qui permet de modéliser la dynamique des systèmes paramétrés à topologie arborescente où les informations circulent des feuilles vers la racine et vice-versa. Nous donnons une construction qui permet de calculer une relation de réétiquetage représentant $\mathcal{R}^*$ pour toute relation $\mathcal{R}$ de la classe.

2.2.4.2 Vérification des programmes récursifs parallèles

En ce qui concerne les programmes récursifs avec création dynamique de processus parallèles, nous considérons deux approches qui peuvent être vues toutes les deux comme des instances du cadre général que nous proposons dans cette section.

La première approche consiste à modéliser ces programmes par des systèmes de réécriture particuliers dénommés PRS (pour Process Rewrite Systems), et à calculer les ensembles des accessibles de ce modèle. Plus précisément, dans cette modélisation, une configuration décrivant la structure de contrôle du programme est représentée par un terme construit à partir du processus nul "0", de constantes "X", et des compositions séquentielles et parallèles "$\cdot$" et "$||$" qui sont respectivement associatif et associatif/commutatif. Dans ce cas, les actions d'un programme sont modélisées par

les dits PRS. Pour faire l'analyse d'acessibilité de ces systèmes, nous proposons deux méthodes :

1. La première consiste à représenter les ensembles de termes par des langages réguliers d'arbres binaires. Dans ce cas, la régularité n'est plus préservée par application du système PRS, et ce à cause de la commutativité/associativité des opérateurs séquentiels et parallèles. Nous étudions alors la "limite de régularité" de ces systèmes. Pour ce faire, nous nous contentons parfois de calculer des représentants des ensembles des accessibles, c-à-d., des ensembles qui contiennent au moins un terme de chaque classe de l'ensemble des accessibles. Nous référons au chapitre 5 pour comprendre l'utilitée de calculer de tels ensembles. Nous proposons alors des algorithmes polynômiaux qui permettent de calculer des automates réguliers d'arbres qui reconnaissent : (1) les ensembles des accessibles quand les équivalences structurelles entre les termes ne sont pas considérées, (2) des représentants de ces ensembles si seule l'associativité/commutativité du "||" n'est pas considérée, (3) des représentants de l'ensemble des successeurs si toutes les équivalences sont considérées et que le système est un PAD (une sous classe significative de PRS). Lorsque toutes les équivalences sont prises en compte, nous ne pouvons malheureusemnet pas caractériser des représentants des ensembles des accessibles pour toute la classe PRS. Nous nous sommes donc restreints à la sous-classe PAD (qui est déjà assez significative). Dans ce cas, calculer des représentants de l'ensemble des prédécesseurs nous fait sortir du cadre régulier. Nous introduisons alors une classe d'automates d'arbres à compteurs (les 0-CTA) qui vérifie la bonne propriété que le problème du vide de l'intersection d'un langage 0-CTA avec un langage régulier d'arbres est décidable, ce qui permet d'utilser cette classe pour résoudre le problème (2.11). Nous donnons un algorithme qui calcule un 0-CTA qui reconnaît un représentant de l'ensemble de prédécesseurs dans le cas PAD. Les détails sur cette approche se trouvent au chapitre 5.
2. La deuxième méthode consiste à exploiter les propriétés d'associativité/commutativité des opérateurs séquentiels et parallèles pour représenter un terme par un arbre à largeur non bornée. Nous introduisons donc une classe d'automates que nous appelons les CH-automates pour représenter des ensembles de termes représentés par ce genre d'arbres. Ces automates peuvent être vus comme une extension des automates réguliers avec des contraintes de Presburger. Nous montrons que les langages reconnus par ces automates sont effectiement fermés par les opérations boléennes, et que le problème du vide est décidable pour cette classe d'automates, ce qui la rend utilisable dans notre cadre. Nous donnons un algorithme qui permet de calculer un CH-automate qui reconnaît les ensembles des accessibles pour une sous classe importante de PRS (qui inclut strictement PAD), et qui peut être appliqué pour calculer des surapproximations des ensembles des accessibles pour toute la classe de PRS. Les détails se trouvent au chapitre 6.

La deuxième approche consiste à résoudre les problèmes (2.10) et (2.11) en calculant des abstractions des ensembles de traces d'actions qui mènent de $\mathcal{L}$ à $\mathcal{L}'$ dans

des modèles qui sont plus puissants que PRS, et qui permettent de modéliser la synchronisation dans les programmes de manière plus précise que dans PRS. Les modèles que nous considérons sont (1) les automates à pile communicants. Dans ce cas, nous représentons les ensembles de configurations d'un automate à pile de manière symbolique par un automate de mots fini [BEM97, EHRS00] ; et (2) les SPA, pour Synchronized PA qui sont des systèmes qui étendent PA à-la CCS avec des opérateurs de synchronisation et de restriction. Dans ce cas, nous représentons, les ensembles de configurations par des langages réguliers d'arbres binaires, comme fait dans le cas des systèmes PRS. Les détails de cette approche se trouvent aux chapitres 7 et 8.

2.3 Conclusion

Nous avons défini un cadre général pour raisonner de manière uniforme sur plusieurs types de systèmes infinis. En effet, les mots et les arbres sont des structures de données très communes et peuvent être utilisées de manière naturelle pour représenter les configurations de plusieurs classes de systèmes.

Dans ce cadre, la vérification des systèmes se fait par une analyse d'accessibilité. Nous avons montré comment ceci peut être utilisé par exemple pour la vérification des propriétés de sûreté. Mais notre cadre peut également être appliqué à la vérification des propriétés de vivacité pour les systèmes paramétrés, c-à-d., les propriétés exprimant qu'une "bonne chose" se produit nécessairement lors de l'exécution du système [BJNT00].

Première partie

Analyse des systèmes paramétrés

Chapitre 3

Analyse des systèmes paramétrés

Un système paramétré est un système pouvant contenir un nombre arbitraire (non borné) de processus. Il peut être vu comme une famille infinie $S = \{S_n\}_{n=0}^{\infty}$ de systèmes $S_n = P_1 \| P_2 \| ... \| P_n$ formés de n processus identiques mis en parallèle. Le problème de la vérification d'un système paramétré peut être formulé comme suit : Etant donnée une famille $S = \{S_n\}_{n=0}^{\infty}$, vérifier que pour tout n, le système S_n est correct.

Par conséquent, vérifier (2.12) pour un système paramétré S revient à tester que

$$\forall n \geq 0, Post^*_{\mathcal{R}_n}(\mathcal{L}_0^n) \cap \mathcal{L}_{Bad}^n = \emptyset, \tag{3.1}$$

où $\mathcal{L}_0^n$, $\mathcal{L}_{Bad}^n$, et $\mathcal{R}_n$ sont respectivement l'ensemble des configurations initiales, l'ensemble des "mauvaises" configurations, et la relation de transition du système S_n. Nous voyons alors que nous avons affaire à une infinité de problèmes à résoudre. Puisque les transformations $\mathcal{R}_n$ ne s'appliquent qu'aux configurations de taille n (en préservant leur taille), et que les configurations de deux systèmes de tailles différentes ont forcément des tailles différentes, les transformations $\mathcal{R}_n$ ne peuvent être appliquées qu'aux configurations du système S_n. Donc, résoudre (3.1) revient à vérifier que

$$Post^*_{\mathcal{R}}(\mathcal{L}_0) \cap \mathcal{L}_{Bad} = \emptyset, \tag{3.2}$$

où $\mathcal{R} = \bigcup_{n \geq 0} \mathcal{R}_n$, $\mathcal{L}_0 = \bigcup_{n \geq 0} \mathcal{L}_0^n$, et $\mathcal{L}_{Bad} = \bigcup_{n \geq 0} \mathcal{L}_{Bad}^n$.

Pour attaquer ce problème, nous devons tout d'abord, comme décrit au chapitre précédent, trouver de "bonnes" structures de représentation qui permettent de représenter finiment ces ensembles infinis de configurations et de transformations. Nous adoptons dans ce chapitre la modélisation de [KMM+97] qui, selon que la topologie du système paramétré en question est linéaire ou arborescente, utilise les langages réguliers de mots ou d'arbres pour représenter les ensembles infinis de configurations, et les relations régulières de mots ou de réétiquetage d'arbres pour représenter les ensembles infinis de transformations. Ensuite, comme expliqué précédemment, le problème d'accessibilité (3.2) peut être résolu de deux manières : (1) Si la relation $\mathcal{R}$ est quelconque,

nous pouvons appliquer notre méthode d'accélération générale basée sur l'élargissement et espérer que le calcul va terminer. Nous présentons cette technique au chapitre 9 de ce document. (2) Sinon, si la relation $\mathcal{R}$ peut être décomposée en l'union de sous-relations $\mathcal{R} = \mathcal{R}' \cup \mathcal{R}'_1 \cup \cdots \mathcal{R}'_m$ telles que l'on ait des algorithmes spécifiques qui permettent de claculer les clôtures transitives par les $\mathcal{R}'_i$; nous pouvons utiliser ces algorithmes pour le calcul de méta-transitions qui peuvent aider la convergence du calcul itératif de l'ensemble des accessibles.

Le but de ce chapitre est de mettre en évidence des classes de relations et de langages réguliers qui soient significatives pour la modélisation des systèmes paramétrés, et pour lesquelles il est possible de proposer des algorithmes qui permettent de calculer $\mathcal{R}^*(\mathcal{L})$[1] pour toute relation $\mathcal{R}$, et tout langage $\mathcal{L}$ des classes considérées. Ou, de manière plus générale, de trouver des classes de relations régulières telles que l'on sache caractériser par un transducteur fini la relation $\mathcal{R}^*$.

Dans la première section, nous considérons les systèmes paramétrés linéaires. Nous montrons d'abord comment les langages et les relations réguliers de mots peuvent modéliser de tels systèmes. Ensuite, nous considérons la classe des relations de semi-commutations, qui sont les relations de la forme

$$\bigcup copy(\Sigma^*)(ab, ba)copy(\Sigma^*).$$

Ces relations permettent de modéliser les systèmes où chaque processus communique avec son voisin. Comme ces relations ne préservent pas la régularité, nous introduisons une classe de langages réguliers de mots (la classe APC) et nous montrons qu'elle est effectivement fermée par semi-commutations. Nous retrouvons cette classe de manière naturelle dans la modélisation de la plupart des systèmes paramétrés que nous considérons. Nous montrons que cette classe est effectivement fermée par union et intersection, ce qui est, comme expliqué au chapitre précédent, important pour que l'algorithme de clôture par semi-commutations puisse être utilisé de manière répétée pour calculer des méta-transitions au cours de la procédure itérative du calcul des accessibles. Nous montrons comment notre résultat peut être utilisé pour réaliser l'analyse d'accessibilité d'un contrôleur d'ascenceur.

A la fin de cette section, nous considérons les systèmes paramétrés à topologies circulaires. Nous représentons les ensembles de configurations de tels systèmes par des langages réguliers de mots circulaires, et nous montrons comment notre résultat précédent peut être étendu pour calculer la clôture circulaire d'une APC par semi-commutations. Les résultats de cette partie ont été publiés dans [BMT01].

Dans la dernière section, nous considérons les systèmes paramétrés à topologies arborescentes. Nous montrons d'abord comment représenter les ensembles de configurations et de transformations de ces systèmes par des langages et des relations de réétiquetage réguliers d'arbres. Ensuite, nous introduisons la classe de relations de réétiquetage des Well Oriented Systems. Cette classe apparaît naturellement dans la modélisation des systèmes où les processus communiquent avec leur père et leurs fils. Nous montrons que pour toute relation de réétiquetage $\mathcal{R}$ dans cette classe, $\mathcal{R}^*$ est une relation de réétiquetage effectivement calculable. Ce résultat a fait l'objet d'une partie de l'article [BT02].

[1] $Post^*_{\mathcal{R}}(\mathcal{L})$ est par définition égal à $\mathcal{R}^*(\mathcal{L})$

3.1 Systèmes paramétrés linéaires

3.1.1 Modélisation

Dans le cas où les processus sont disposés linéairement, une configuration du système S_n peut être vue comme un mot $e_1...e_n$ de Σ^n (Σ étant l'ensemble des états locaux de chaque processus), où e_i représente l'état du processus i. Et donc, une configuration du système $S = \{S_n\}_{n=0}^{\infty}$ peut être considérée comme un mot de Σ^* [KMM+97]. Par exemple, considérons un système formé de trois processus dont chacun peut être soit dans l'état a soit dans l'état b. Le mot aba correspond à une configuration où les processus 1 et 3 sont dans l'état a et le processus 2 dans l'état b. Par conséquent, nous pouvons utiliser les langages réguliers pour représenter un ensemble de configurations de la famille $S = \{S_n\}_{n=0}^{\infty}$. Ainsi, l'expression régulière a^*ba^* représente l'ensemble des configurations où il y a exactement un seul processus dans l'état b et un nombre arbitraire de processus dans l'état a.

De même, puisque les actions d'un programme font passer le système d'une configuration à une autre de même taille, une action peut être modélisée par une relation régulière qui préserve la longueur (ou de manière équivalente, une règle de réécriture) sur Σ^*. Ainsi, dans la relation

$$copy\big((a+b)^*\big)(a,b)copy\big((a+b)^*\big),$$

a (resp. b) dénote l'état de la composante modifiée du système avant (resp. après) l'application de l'action du programme.

Nous illustrons cette modélisation par l'exemple d'un contrôleur d'ascenceur. Des exemples de protocoles d'exclusion mutuelle tels que le *bakery algorithm* et les protocoles de Dijkstra et de Szymanski figurent à la section 9.6 et en annexes.

3.1.1.1 Un contrôleur d'ascenceur

Nous considérons un contrôleur d'ascenceur qui a le comportement suivant : A chaque instant, les usagers peuvent arriver à n'importe quel étage et déclarer leur intention de monter ou de descendre. L'ascenceur est initialement au rez-de chaussée, et fait continuellement des va-et-vients entre le rez-de chaussée et le dernier étage : Quand il est au rez-de chaussée, il monte jusqu'au dernier étage, et quand il arrive à ce dernier étage, il redescend, etc.

Quand il est en train de monter (resp. de descendre), l'ascenceur prend toutes les personnes qui souhaitent monter (resp. descendre) et ignore les autres. Il prendra ces derniers quand il descendra (resp. montera).

Ce système est paramétré puisqu'il peut y avoir un nombre arbitraire d'étages. Si le nombre d'étages est n, une configuration du système peut être représentée par un mot de la forme $\#x_1 \cdots x_j y x_{j+1} \cdots x_n \#$, où $y \in \{A\uparrow, A\downarrow\}$ et $x_i \in \{\bot, p\uparrow, p\downarrow, p\uparrow\downarrow\}$, pour $i \in \{1, \ldots, n\}$. Le symbole correspondant à x_i représente l'état du $i^{ème}$ étage : $x_i = \bot$ s'il n'y a personne qui attend à cet étage, $x_i = p\uparrow$ (resp. $x_i = p\downarrow$) si à l'étage i il n'y a que des personnes qui attendent pour monter (resp. pour descendre), et $x_i = p\uparrow\downarrow$ si à cet étage il y a à la fois des gens qui désirent monter et d'autres qui veulent descendre. Le symbole correspondant à y permet de déterminer la position de l'ascenceur : Dans

la configuration ci-dessus, $y = A\uparrow$ (resp. $y = A\downarrow$) exprime que l'ascenceur est à l'étage $j + 1$ (resp. j), et qu'il est en train de monter (resp. de descendre).

L'ensemble de toutes les configurations initiales (pour un nombre arbitraire d'étages) est l'ensemble $L_0 = \#A\uparrow \perp^*\#$, signifiant que l'ascenceur est initialement au rez-de chaussée, et qu'il n'y a encore personne aux différents étages.

La dynamique du système peut être modélisée par les relations régulières (3.3)-(3.16), où $\Sigma = \{\#, \perp, p\uparrow, p\downarrow, p\uparrow\downarrow, A\uparrow, A\downarrow\}$.

Les relations 3.3, 3.4, 3.5, et 3.6 représentent des substitutions modélisant l'arrivée des usagers. Appelons les "request". Les relations 3.7 et 3.8 (resp. 3.12 et 3.13) sont des semi-commutations modélisant les montées (resp. les descentes) de l'ascenceur. Appelons ces actions move-up (resp. move-down). Les relations 3.9 et 3.10 (resp. 3.14 et 3.15) représentent l'action de prendre à un certain étages les gens qui désirent monter (resp. ceux qui veulent descendre). Nous appelons ces actions take-up (resp. take-down). Enfin, les relations 3.11 et 3.16 représentent les actions de passer de la phase ascendante à la phase descendante (action up2down), et vice-versa (action down2up).

$$copy(\Sigma^*)(\perp, p\uparrow)copy(\Sigma^*) \tag{3.3}$$
$$copy(\Sigma^*)(\perp, p\downarrow)copy(\Sigma^*) \tag{3.4}$$
$$copy(\Sigma^*)(p\uparrow, p\uparrow\downarrow)copy(\Sigma^*) \tag{3.5}$$
$$copy(\Sigma^*)(p\downarrow, p\uparrow\downarrow)copy(\Sigma^*) \tag{3.6}$$
$$copy(\Sigma^*)(A\uparrow \perp, \perp A\uparrow)copy(\Sigma^*) \tag{3.7}$$
$$copy(\Sigma^*)(A\uparrow p\downarrow, p\downarrow A\uparrow)copy(\Sigma^*) \tag{3.8}$$
$$copy(\Sigma^*)(A\uparrow p\uparrow, \perp A\uparrow)copy(\Sigma^*) \tag{3.9}$$
$$copy(\Sigma^*)(A\uparrow p\uparrow\downarrow, p\downarrow A\uparrow)copy(\Sigma^*) \tag{3.10}$$
$$copy(\Sigma^*)(A\uparrow \#, A\downarrow \#)copy(\Sigma^*) \tag{3.11}$$
$$copy(\Sigma^*)(\perp A\downarrow, A\downarrow \perp)copy(\Sigma^*) \tag{3.12}$$
$$copy(\Sigma^*)(p\uparrow A\downarrow, A\downarrow p\uparrow)copy(\Sigma^*) \tag{3.13}$$
$$copy(\Sigma^*)(p\downarrow A\downarrow, A\downarrow \perp)copy(\Sigma^*) \tag{3.14}$$
$$copy(\Sigma^*)(p\uparrow\downarrow A\downarrow, A\downarrow p\uparrow)copy(\Sigma^*) \tag{3.15}$$
$$copy(\Sigma^*)(\#A\downarrow, \#A\uparrow)copy(\Sigma^*) \tag{3.16}$$

3.1.2 Calcul des clôtures par semi-commutations

Nous nous intéressons dans cette section aux relations de semi-commutations, c-à-d., aux relations de la forme

$$\bigcup copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$$

telles que $a \neq b$. Ces relations peuvent également être représentées par des règles de réécriture de la forme $ab \to ba$. Nous retrouvons ces relations naturellement lors de la modélisation des systèmes paramétrés où les processus voisins communiquent entre eux. Par exemple, dans le modèle du contrôleur d'ascenceur, nous voyons que les

relations 3.7, 3.8, 3.12, et 3.13 sont des semi-commutations qui représentent un genre de communication entre le "processus" *ascenceur* et les différents "processus" *étages*. Il est donc important, pour l'analyse des systèmes paramétrés, d'avoir des algorithmes qui calculent la fermeture d'un langage régulier par ce genre de relations. Seulement, il n'est pas difficile de voir que les semi-commutations ne préservent pas la régularité, et donc que la relation $\mathcal{R}^*$ ne peut en général pas être représentée par un transducteur fini de mots. Considérons par exemple l'ensemble $\mathcal{L} = (ab)^*$ et la semi-commutation $\mathcal{R} = \{ba \rightarrow ab\}$. Il s'en suit que $\mathcal{R}^*(\mathcal{L})$ est l'ensemble non régulier de tous les mots ayant le même nombre de a et de b, et tels que tous leurs préfixes ont au moins autant de a que de b.

Nous introduisons alors une sous-classe de langages réguliers : la classe APC, qui est une classe significative et importante dans la modélisation des systèmes paramétrés. Elle apparaît de manière naturelle dans les modèles correspondant à tous les systèmes paramétrés linéaires que nous considérons dans cette thèse. Nous donnons un algorithme qui calcule pour tout langage APC $\mathcal{L}$ et toute relation de semi-commutations $\mathcal{R}$, un langage APC correspondant à $\mathcal{R}^*(\mathcal{L})$. Comme mentionné dans l'introduction de ce chapitre, le fait que $\mathcal{R}^*(\mathcal{L})$ soit aussi un langage APC et que les APC soient fermés par union est important pour que notre algorithme puisse être utilisé à plusieurs reprises pour calculer des méta-transitions lors du calcul itératif de l'ensemble des accessibles. Nous illustrons ceci dans l'analyse du contrôleur d'ascenceur donné ci-dessus.

3.1.2.1 Alphabetic Pattern Constraints

Nous définissons dans ce qui suit la classe *Alphabetic Pattern Constraints (APC)*, et nous montrons qu'elle est fermée par union, intersection, et concaténation, mais pas par complémentation.

Définition 3.1.1 *Soit Σ un alphabet fini. Une* expression atomique *sur Σ est soit une lettre a de Σ, soit une* expression étoile *$(a_1 + a_2 + \cdots + a_n)^*$, où $a_1, a_2, \ldots, a_n \in \Sigma$.*

Un produit p sur Σ^* est la concaténation (qui peut être vide) $e_1 e_2 \cdots e_n$ d'expressions atomiques $e_1, \ldots, e_n$ sur Σ. Nous utilisons ϵ pour représenter le produit vide.

Une "Alphabetic Pattern Constraint" (APC) sur Σ^* est une expression de la forme $p_1 + \cdots + p_n$, où $p_1, \ldots, p_n$ sont des produits sur Σ^*.

Dans le reste de cette section, nous ne faisons pas de distinction entre une expression régulière et le langage qu' elle définit. Cependant, les entrées des algorithmes que nous proposons seront des expressions APC.

Notation 3.1.1 *La longueur de $p = e_1 \cdots e_n$, dénotée $l(p) = n$, est le nombre d'expressions atomiques dans p. Soit une expression APC $e = \sum_i p_i$, la* longueur *de e est définie par $l(e) = \max_i l(p_i)$. La* taille *d'une expression est la somme des longueurs de ses produits. Pour un langage $\mathcal{L} \subseteq \Sigma^*$, nous dénotons par $\alpha(\mathcal{L})$ l'ensemble des lettres de Σ apparaissant dans les mots de $\mathcal{L}$.*

Propriétés de clôture et résultats de complexité Il est facile de voir que la classe des langages APC n'est pas fermée par complémentation. En effet, considérons par exemple l'alphabet $\Sigma = \{a, b\}$ et le langage APC $\Sigma^* aa\Sigma^* + \Sigma^* bb\Sigma^* + b\Sigma^* + \Sigma^* a$. Son complémentaire est $(ab)^*$ qui n'est pas un langage APC.

La proposition suivante donne quelques propriétés de fermeture de la classe APC.

Proposition 3.1.1 *APC est fermée par concaténation, union, et intersection.*

Preuve : APC est par définition fermée par concaténation et union.

Soient p_1 et p_2 deux produits sur Σ. L'expression APC correspondant à $p_1 \cap p_2$ peut être calculée inductivement comme suit :

$p_1 \cap p_2 =$

$$\begin{cases} \emptyset & \text{si } p_1 = \emptyset \text{ ou } p_2 = \emptyset \\ \varepsilon & \text{si } p_1 = \varepsilon \text{ et } \varepsilon \in p_2 \\ & \text{ou } p_2 = \varepsilon \text{ et } \varepsilon \in p_1 \\ a(p'_1 \cap p'_2) & \text{si } p_1 = ap'_1 \text{ et } p_2 = ap'_2 \\ \emptyset & \text{si } p_1 = ap'_1, p_2 = bp'_2, \text{ et } a \neq b \\ a(p'_1 \cap p_2) & \text{si } p_1 = ap'_1 \text{ et } p_2 = A^* p'_2, \text{ t.q. } a \in A \\ p_1 \cap p'_2 & \text{si } p_1 = ap'_1 \text{ et } p_2 = A^* p'_2, \text{ t.q. } a \notin A \\ C^*(p_1 \cap p'_2) + C^*(p'_1 \cap p_2) & \text{si } p_1 = A^* p'_1, p_2 = B^* p'_2, \text{ et } C = A \cap B \neq \emptyset \\ (p_1 \cap p'_2) + (p'_1 \cap p_2) & \text{si } p_1 = A^* p'_1, p_2 = B^* p'_2, \text{ et } A \cap B = \emptyset \end{cases}$$

□

Remarque 3.1.1 *L'union et la concaténation sont des operations polynômiales, tandis que le calcul de l'intersection de deux langages APC produit une expression de taille exponentielle. En effet, l'exemple suivant montre que le pire-cas est exponentiel. Considérons les produits $p_n = b^*(ab^*)^n$ et $q_n = (a^*b)^n a^*$, chacun est de taille $2n + 1$. Toute expression APC repésentant $p_n \cap q_n$ est de taille exponentielle, puisque $p_n \cap q_n = \{w \in (a + b)^* \mid |w|_a = |w|_b = n\}$.*

Nous pouvons montrer les résultats de complexité suivants pour la classe APC :

Théorème 3.1.1 *[BMT01] Les problèmes suivants sont PSPACE-complets :*

– *décider si une expression APC sur Σ^* est égale à Σ^* ;*
– *décider l'inclusion entre des langages dans APC.*

De plus, décider si un langage régulier, donné par une expression régulière ou un automate non déterministe, est un langage APC est aussi PSPACE-complet. Ce même problème est NLOGSPACE-complet quand l'entrée est un automate déterministe.

Caractérisations des APCs Nous avons défini la classe APC de manière naturelle en observant les modèles des systèmes paramétrés que nous avons considéré. Il s'est avéré que cette classe de langage existe déjà dans la littérature sous d'autres appellations, et qu'elle a été étudiée de manière intense dans le cadre de la théorie des groupes. En effet APC correspond à l'ensemble des expressions définissant les langages de niveau 3/2 dans la hiérarchie de Straubing-Thérien [PW97]. Dans cette hiérarchie, le niveau 0 correspond à $\{\Sigma^*, \emptyset\}$ et les niveaux suivants sont définis inductivement comme suit : le niveau $n + 1/2$ est la clôture polynômiale du niveau n et le niveau $n+1$ est la clôture booléenne du niveau $n+1/2$, où la clôture polynômiale d'une classe $\mathcal{C}$ de Σ^+ est l'ensemble des langages de Σ^+ qui sont l'union finie de langages de la forme $L_0 a_0 L_1 \cdots a_n L_n$ t.q. les a_i sont des lettres et les L_i sont des éléments de $\mathcal{C}$. La hiérarchie de Straubing-Thérien est stricte.

Il existe aussi un lien avec la Σ_n-hiérarchie de la logique de premier ordre. Cette hiérarchie est obtenue en comptant le nombre de blocs alternants de quantificateurs existentiels et universels. Le niveau Σ_n représente les formules avec n alternances de blocs de quantificateurs, commençant par un bloc existentiel. Thomas [Tho82] a montré que le niveau $n + 1/2$ dans la hiérarchie de Straubing-Thérien correspond précisément au niveau Σ_{n+1}. Ainsi, il s'en suit que chaque langage dans APC peut être défini par une formule de la forme $\phi = \exists^* \forall^* \psi$ où ψ est sans quantificateurs.

Egalement, APC correspond au fragment suivant de la logique temporelle linéaire (LTL) :

$$\phi ::= A \mid \phi \vee \phi \mid \phi \wedge \phi \mid A \mathcal{U} \phi \mid \Box A \mid \bigcirc \phi$$

où A est de la forme $A = a_1 \vee \cdots \vee a_n$, et $\bigcirc, \mathcal{U}$, et $\Box$ sont respectivement les opérateurs *next, until*, et *always*. En effet, chaque APC peut être décrite par une formule de ce fragment puisqu'un produit de la forme $\Sigma_0^* a_0 \Sigma_1^* a_1 \cdots a_{n-1} \Sigma_n^*$ est équivalent à la formule

$$\Sigma_0 \mathcal{U}(a_0 \wedge \bigcirc(\Sigma_1 \mathcal{U}(a_1 \wedge \bigcirc \cdots (\Sigma_{n-1} \mathcal{U}(a_{n-1} \wedge \bigcirc \Box \Sigma_n)) \cdots)$$

Pour l'autre direction, nous pouvons raisonner par induction structurelle sur les formules en utilisant le fait que APC est fermée par union, intersection, et concaténation (Proposition 3.1.1).

3.1.2.2 Fermeture des APC par semi-Commutations

Nous montrons dans ce qui suit le principal résultat de cette section, qui énonce que les APC sont effectivement fermés par semi-commutations :

Théorème 3.1.2 *Pour toute expression APC $\mathcal{L}$ et toute relation de semi-commutations $\mathcal{R}$, $\mathcal{R}^*(\mathcal{L})$ appartient à APC et peut être effectivement calculée. En plus, la longueur de l'expression calculée est dans $\mathcal{O}(l(\mathcal{L})|\Sigma|)$.*

Le reste de cette section est consacré à la description de l'algorithme sous-jacent à ce théorème. Puisque chaque langage APC $\mathcal{L}$ est une union finie de produits, sa clôture $\mathcal{R}^*(\mathcal{L})$ est l'union des clôtures de ses produits. Par conséquent, il suffit de montrer comment calculer effectivement $\mathcal{R}^*(p)$ pour un certain produit donné p. Pour ceci, nous utilisons l'opération de $\mathcal{R}-$shuffle définie ci-dessous. L'idée est de calculer $\mathcal{R}^*(e_1 \cdots e_n)$ inductivement, c-à-d., calculer d'abord $\mathcal{R}^*(e_2 \cdots e_n)$, et utiliser le fait que $\mathcal{R}^*(e_1) = e_1$. L'étape d'induction consiste à calculer $\mathcal{R}^*(eL)$, pour une expression APC fermée par $\mathcal{R}$ L et une expression atomique e. Cette opération est aussi faite inductivement. Pour ces calculs, nous avons besoin de certaines notations et définitions :

Nous associons à chaque relation de semi-commutations $\mathcal{R}$ une relation binaire irreflexive $\rho_{\mathcal{R}}$ sur les lettres de Σ, c-à-d., un sous ensemble de $\Sigma \times \Sigma \setminus \{(a,a) \mid a \in \Sigma\}$ telle que

$$\mathcal{R} = \bigcup_{(a,b)\in\rho_{\mathcal{R}}} copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$$

La relation $\rho_{\mathcal{R}}$ peut être étendue aux ensembles comme suit : Pour chaque ensembles $X, Y \subseteq \Sigma$:

$$(X, Y) \in \rho_{\mathcal{R}} \text{ si } X \times Y \subseteq \rho_{\mathcal{R}} .$$

Nous dénotons par $\delta_{\mathcal{R}}$ la valeur

$$\delta_{\mathcal{R}} = \max_{a\in\Sigma}\{|Y| \mid Y \subseteq \Sigma \text{ t.q. } (Y, a) \in \rho_{\mathcal{R}}\} .$$

Nous supposons que $\rho_{\mathcal{R}} \neq \emptyset$, ce qui veut dire que $\delta_{\mathcal{R}} > 0$.

Etant donnés deux mots x et y de Σ^*, le $\mathcal{R}-$**shuffle** de x et y, dénoté by $x \sqcup\!\sqcup_{\mathcal{R}} y$, est l'ensemble des mots de la forme $x_1y_1 \cdots x_ny_n$ avec $x = x_1 \cdots x_n$, $y = y_1 \cdots y_n$, $x_i, y_i \in \Sigma^*$ pour tout $1 \leq i \leq n$ et tels que $(\alpha(x_i), \alpha(y_j)) \in \rho_{\mathcal{R}}$ pour tout $j < i$.

L'opération de $\mathcal{R}-$shuffle s'étend aux ensembles $X, Y \subseteq \Sigma^*$ comme suit :

$$X \sqcup\!\sqcup_{\mathcal{R}} Y = \{x \sqcup\!\sqcup_{\mathcal{R}} y \mid x \in X, y \in Y\}.$$

Notons que pour tout $x, y \in \Sigma^*$, nous avons $\mathcal{R}^*(xy) = \mathcal{R}^*(x) \sqcup\!\sqcup_{\mathcal{R}} \mathcal{R}^*(y)$. Le lemme suivant montre comment calculer $\mathcal{R}^*(\mathcal{L}\mathcal{K})$ quand $\mathcal{L}$ et $\mathcal{K}$ sont déjà $\mathcal{R}$-clos (un langage $\mathcal{L}$ et $\mathcal{R}$-clos si $\mathcal{R}^*(\mathcal{L}) = \mathcal{L}$).

Lemme 3.1.1 *Soient $\mathcal{L}$ et $\mathcal{K}$ deux ensembles fermés par $\mathcal{R}$. Nous avons alors*

$$\mathcal{R}^*(\mathcal{L}\mathcal{K}) = \mathcal{L} \sqcup\!\sqcup_{\mathcal{R}} \mathcal{K}.$$

Puisque toute expression atomique est $\mathcal{R}$-close, nous avons :

Lemme 3.1.2 *Soient $e_1, e_2, \ldots, e_n$ des expressions atomiques, et soit $p = e_1e_2 \cdots e_n$ un produit, nous avons :*

$$\mathcal{R}^*(p) = e_1 \sqcup\!\sqcup_{\mathcal{R}} (e_2 \sqcup\!\sqcup_{\mathcal{R}} (\cdots (e_{n-1} \sqcup\!\sqcup_{\mathcal{R}} e_n) \cdots)).$$

Le lemme précédent permet de calculer $\mathcal{R}^*(p)$ récursivement. Le lemme 3.1.3 et la proposition 3.1.2 sont les cas de base de notre algorithme.

Lemme 3.1.3 *Soit E un sous ensemble de Σ et soit une lettre $a \in \Sigma$, nous avons :*

$$E^* \sqcup\!\sqcup_{\mathcal{R}} a = \mathcal{R}^*(E^* a) = E^* a E'^* \text{ et } a \sqcup\!\sqcup_{\mathcal{R}} E^* = \mathcal{R}^*(aE^*) = E''^* a E^*,$$

où $E' = \{x \in E \mid (x, a) \in \rho_{\mathcal{R}}\}$ et $E'' = \{x \in E \mid (a, x) \in \rho_{\mathcal{R}}\}$.

Preuve : Le fait que $E^* \sqcup\!\sqcup_{\mathcal{R}} a = \mathcal{R}^*(E^* a)$ et $a \sqcup\!\sqcup_{\mathcal{R}} E^* = \mathcal{R}^*(aE^*)$ est dû au lemme 3.1.1, puisque E^* et a sont clos par $\mathcal{R}$. Les autres égalités sont directes puisque les symboles de E' sont précisément les lettres de E qui sont capables de passer à travers la lettre a, et les symboles de E'' sont précisément les lettres de E que la lettre a peut traverser.

□

Exemple 3.1.1 *Considérons le produit $p = (e + f + g)^* d$, et la relation de semi-commutation $\mathcal{R}_1$ définie par la relation binaire $\rho_{\mathcal{R}_1} = \{(f, d), (g, d)\}$. D'aprés le lemme précédent, nous avons :*

$$\mathcal{R}_1^*(p) = (e + f + g)^* \sqcup\!\sqcup_{\mathcal{R}_1} d = (e + f + g)^* d (f + g)^*.$$

La proposition suivante est le plus important résultat technique nécessaire pour la preuve du théorème 3.1.2. Elle montre que la $\mathcal{R}$-clôture d'un produit qui consiste en la concaténation de deux expressions étoiles est une APC. En particulier, notons que la longueur des produits dans l'expression donnée ci-dessous est bornée par une constante n qui est polynômiale en $|\Sigma|$ et $|\mathcal{R}|$. Plus préciséments, $n \leq \delta_{\mathcal{R}} \leq |\Sigma|$.

Proposition 3.1.2 *Soient E et F deux sous ensembles de Σ, nous avons*

$$E^* \sqcup\!\sqcup_{\mathcal{R}} F^* = \mathcal{R}^*(E^* F^*) = \sum E^*(E_1 + F_1)^* \cdots (E_n + F_n)^* F^*,$$

où $n \leq \delta_{\mathcal{R}}$, et la somme est prise sur tous les sous ensembles E_i et F_i de Σ qui satisfont les conditions suivantes :

- $\emptyset \neq E_n \subsetneq \cdots \subsetneq E_1 \subseteq E$,
- $\emptyset \neq F_1 \subsetneq \cdots \subsetneq F_n \subseteq F$,
- $(E_i, F_j) \in \rho_{\mathcal{R}}$ *pour tout* $1 \leq j \leq i \leq n$.

Preuve : La première égalité peut être obtenue comme précédemment à partir du lemme 3.1.1 puisque E^* et F^* sont clos par $\mathcal{R}$.

Considérons alors la seconde égalité. Il est clair que $E^*(E_1+F_1)^* \cdots (E_n+F_n)^* F^* \subseteq \mathcal{R}^*(E^* F^*)$ si E_i et F_i satisfont $(E_i, F_j) \in \rho_{\mathcal{R}}$ pour tout $j \leq i$.

Réciproquement, soit $w \in E^* \sqcup\!\sqcup_{\mathcal{R}} F^* = \mathcal{R}^*(E^* F^*)$. Nous pouvons écrire $w = u_1 v_1 u_2 v_2 \cdots u_m v_m$ avec $u_i \in E^*, v_i \in F^*$, et tels que l'on a $(\alpha(u_i), \alpha(v_j)) \in \rho_{\mathcal{R}}$ pour tout $j < i$. Nous pouvons clairement supposer que $u_i, v_j \neq \epsilon$ pour tout $i \neq 1$ et $j \neq m$.

Nous définissons inductivement les séquences $(k_i)_{1 \leq i \leq n}$, $(E_i)_{1 \leq i \leq n}$, et $(F_i)_{1 \leq i \leq n}$:

- $k_1 = 1$, $k_i = \min\{j \mid k_{i-1} < j < m\,, v_j \notin F_{i-1}^*\}$ $(i > 1)$,
- $E_i = \alpha(u_{k_i+1} \cdots u_m)$,
- $F_i = \{y \in F \mid (E_i, y) \in \rho_{\mathcal{R}}\}$.

Par définition, nous avons $E_{i+1} \subsetneq E_i \subseteq E$, et $F_i \subsetneq F_{i+1} \subseteq F$ pour tout i. En plus, $(E_i, F_i) \in \rho_{\mathcal{R}}$ pour tout i, donc $(E_i, F_j) \in \rho_{\mathcal{R}}$ pour tout $j \leq i$. Ces deux faits impliquent bien sûr que $n \leq \delta_R$. Finalement, nous notons que $u_{k_i+1} \cdots u_{k_{i+1}} \in E_i^*$ et $v_{k_i} \cdots v_{k_{i+1}-1} \in F_i^*$, ce qui montre le résultat.

□

Remarque 3.1.2 *Dans l' expression donnée pour $\mathcal{R}^*(E^*F^*)$, il suffit de considérer seulement les expressions étoiles $(E_i + F_i)^*$ telles que $E_i$et F_i sont maximaux, dans le sens suivant :*

- *il n'existe pas de lettre $b \in F \backslash F_i$ telle que $(E_i, b) \in \rho_{\mathcal{R}}$,*
- *il n'existe pas de lettre $a \in E_{i-1} \backslash E_i$ telle que $(a, F_i) \in \rho_{\mathcal{R}}$, où $E_0 = E$.*

Remarque 3.1.3 *Notons que la longueur des produits dans l'expression pour $\mathcal{R}^*(E^*F^*)$ est au plus $\delta_{\mathcal{R}} + 2$.*

Exemple 3.1.2 *Considérons le produit $p = (a+b+c)^*(e+f+g)^*$, et la relation de semi-commutation $\mathcal{R}_2$ définie par la relation binaire $\rho_{\mathcal{R}_2} = \{(a,e),(c,g),(b,e),(b,f)\}$. La proposition ci-dessus implique que $\mathcal{R}_2^*(p) = (a+b+c)^* \sqcup\!\sqcup_{\mathcal{R}_2} (e+f+g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* + (a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^*$.*

Nous montrons maintenant comment calculer effectivement une APC qui représente $\mathcal{R}^*(p) = \mathcal{R}^*(e_1 e_2 \cdots e_n)$. Dans le lemme 3.1.3 et la proposition 3.1.2, nous avons montré le résultat pour $n = 2$. Supposons maintenant que $\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 f_2 \cdots f_k$, où les f_i sont des expressions atomiques. Montrons que $\mathcal{R}^*(e_1 e_2 \cdots e_n)$, qui est égal à $\sum e_1 \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_k)$ appartient aussi à APC. Pour ce faire, nous avons juste besoin de calculer $e_1 \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_k)$ et de montrer qu'elle est sous la bonne forme. Nous distinguons deux cas selon que e_1 est une lettre ou une expression étoile. Le premier cas est direct :

Lemme 3.1.4 *Soit $a \in \Sigma$ et $f_1, \ldots, f_n$ des expressions atomiques, alors*

$$a \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_n) = \sum_j g_1 \cdots g_j a h_j f_{j+1} \cdots f_n$$

tels que, pour tout $i \leq j$ nous avons :

- *si $f_i = E^*$, alors $g_i = \{x \in E \mid (a,x) \in \rho_{\mathcal{R}}\}^*$,*
- *si $f_i = b \in \Sigma$ et $(a,b) \in \rho_{\mathcal{R}}$, alors $g_i = b$.*

En plus, $h_j = \varepsilon$ si $f_j \in \Sigma$, et $h_j = f_j$ sinon.

Exemple 3.1.3 *Soit $\mathcal{R}_3$ une relation de semi-commutation définie par*

$$\rho_{\mathcal{R}_3} = \{(h,a),(h,e)\}.$$

Le lemme précédent implique que $h \sqcup\!\sqcup_{\mathcal{R}_3} (a+b+c)^*(a+b+e)^*(b+e+f)^* = a^*h(a+b+c)^*(a+b+e)^*(b+e+f)^* + (a+e)^*h(a+b+e)^*(b+e+f)^*$.

La proposition suivante généralise le lemme 3.1.3 et la proposition 3.1.2.

Proposition 3.1.3 *Soient E et F deux sous ensembles de Σ, $a \in \Sigma$, et $L \subseteq \Sigma^*$, nous avons :*

1. $E^* \sqcup\!\sqcup_{\mathcal{R}} (aL) = (E^* \sqcup\!\sqcup_{\mathcal{R}} a)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L)$, *où* $E' = \{b \in E \mid (b,a) \in \rho_{\mathcal{R}}\}$.
2. $E^* \sqcup\!\sqcup_{\mathcal{R}} (F^* L) = \sum_{\substack{(E',F') \in \rho_{\mathcal{R}} \\ E' \subseteq E\ ,F' \subseteq F}} (E^* \sqcup\!\sqcup_{\mathcal{R}} F'^*)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L)$.

Preuve : La première égalité est directe puisque E' est l'ensemble des lettres de E capables de traverser a et après commuter avec L. Nous montrons maintenant la seconde identité. L'inclusion $\supseteq$ est claire. Réciproquement, soit $w \in E^* \sqcup\!\sqcup_{\mathcal{R}} F^* L$, avec $w \in x \sqcup\!\sqcup_{\mathcal{R}} yz$ tels que $x \in E^*, y \in F^*$ et $z \in L$. Supposons que $w = uv$ et $x = x_1 x_2$ avec $u \in x_1 \sqcup\!\sqcup_{\mathcal{R}} y, v \in x_2 \sqcup\!\sqcup_{\mathcal{R}} z$. Soit $F' = \alpha(y)$ et $E' = \alpha(x_2)$. Il est clair que $(E', F') \in \rho_{\mathcal{R}}$, $u \in E^* \sqcup\!\sqcup_{\mathcal{R}} F^*$, et $v \in F^* \sqcup\!\sqcup_{\mathcal{R}} L$.

□

Corollaire 3.1.1 *Soient E et F deux sous ensemble de Σ, et soit $L \subseteq \Sigma^*$, alors*

$$E^* \sqcup\!\sqcup_{\mathcal{R}} (F^* L) = \sum E^*(E_1 + F_1)^*(E_2 + F_2)^* \cdots (E_k + F_k)^*(E_k^* \sqcup\!\sqcup_{\mathcal{R}} L)\,,$$

où l'union est prise sur tous les sous ensembles E_i et F_i de Σ qui satisfont :
- $E_k \subsetneq \cdots \subsetneq E_1 \subseteq E$,
- $\emptyset \neq F_1 \subsetneq \cdots \subsetneq F_k \subseteq F$,
- $(E_i, F_j) \in \rho_{\mathcal{R}}$ *pour tout* $1 \leq j \leq i \leq k$.

Preuve : L'inclusion droite-gauche est directe. A partir de la proposition 3.1.3, il reste à montrer que

$$(E^* \sqcup\!\sqcup_{\mathcal{R}} F'^*)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L) \subseteq \sum E^*(E_1 + F_1)^* \cdots (E_k + F_k)^*(E_k^* \sqcup\!\sqcup_{\mathcal{R}} L)\,,$$

où $E' \subseteq E$ et $F' \subseteq F$ sont des sous ensembles satisfaisant $(E', F') \in \mathcal{R}$. La proposition 3.1.2 implique que :

$$(E^* \sqcup\!\sqcup_{\mathcal{R}} F'^*)(E'^* \sqcup\!\sqcup_{\mathcal{R}} L) = \sum E^*(E_1 + F_1')^* \cdots (E_n + F_n')^* F'^*(E'^* \sqcup\!\sqcup_{\mathcal{R}} L)$$

où la somme est prise sur tous les sous ensembles E_i et F_i' de Σ satisfaisant :
- $\emptyset \neq E_n \subsetneq \cdots \subsetneq E_1 \subseteq E$,
- $\emptyset \neq F_1' \subsetneq \cdots \subsetneq F_n' \subseteq F'$,
- $(E_i, F_j') \in \rho_{\mathcal{R}}$ pour tout $1 \leq j \leq i \leq n$.

Il reste alors à montrer que pour de tels sous ensembles E_i, F_j', ils existent deux séquences $(\widetilde{E_i})_i$ et $(\widetilde{F_i})_i$ satisfaisant les conditions ci-dessus tels que :

$$E^*(E_1 + F_1')^* \cdots (E_n + F_n')^* F'^*(E'^* \sqcup\!\sqcup_{\mathcal{R}} L) \subseteq E^*(\widetilde{E_1} + \widetilde{F_1})^* \cdots (\widetilde{E_k} + \widetilde{F_k})^*(\widetilde{E_k}^* \sqcup\!\sqcup_{\mathcal{R}} L)$$

Ces séquences peuvent être définies inductivement comme suit :
- $\widetilde{E_1} = E_1 + E'$, et $\widetilde{F_1} = F_j'$, où $j = \max\{i \mid (E_i + E') = (E_1 + E')\}$,

– Si $\widetilde{F_i} = F'_l$ alors :
  – Si $l < n$, alors $\widetilde{E_{i+1}} = E_{l+1} + E'$ et $\widetilde{F_{i+1}} = F'_j$, où $j = \max_i\{l+1 \leq i \leq n \mid (E_i + E') = (E_{l+1} + E')\}$;
  – Si $l = n$, $\widetilde{E_{i+1}} = E'$ et $\widetilde{F_{i+1}} = F'$.

Alors, nous avons clairement, $\widetilde{E_k} \subsetneq \cdots \subsetneq \widetilde{E_1} \subseteq E$, $\emptyset \neq \widetilde{F_1} \subsetneq \cdots \subsetneq \widetilde{F_k} \subseteq F'$, et $(\widetilde{E_i}, \widetilde{F_j}) \in \rho_{\mathcal{R}}$ pour tout $1 \leq j \leq i \leq k$ puisque $(E', F') \in \rho_{\mathcal{R}}$ et $(E_i, F'_j) \in \rho_{\mathcal{R}}$ pour tout $1 \leq j \leq i \leq n$.

□

Exemple 3.1.4 *Soit $\mathcal{R}_4$ la relation de semi-commutations définie par*

$$\rho_{\mathcal{R}_4} = \{(a,e),(c,g),(b,e),(b,f),(a,d)\}$$

Le corollaire ci-dessus et l'exemple 3.1.2 impliquent que
$(a+b+c)^* \sqcup\!\sqcup_{\mathcal{R}_4} (e+f+g)^* d (f+g)^* = (a+b+c)^*(c+g)^*(e+f+g)^* d (f+g)^* + (a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^* d (f+g)^* + (a+b+c)^*(a+b+e)^* da^*(f+g)^*$.

Résumant les calculs précédents, la proposition 3.1.3 et le corollaire 3.1.1 fournissent l'étape de récursion pour calculer $E^* \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_n)$:

Proposition 3.1.4 *Soit $E \subseteq \Sigma$ et soient $f_1, \ldots, f_n$ des expressions atomiques. Alors $E^* \sqcup\!\sqcup_{\mathcal{R}} (f_1 f_2 \cdots f_n)$ est égal à l'une des expressions suivantes :*

1. *Pour une expression étoile $f_1 = F^*$:*

$$\sum E^*(E_1 + F_1)^* \cdots (E_k + F_k)^* (E_k^* \sqcup\!\sqcup_{\mathcal{R}} f_2 \cdots f_n),$$

   où l'union est prise sur tous les sous ensembles E_i, F_i satisfaisant $E_{i+1} \subsetneq E_i \subseteq E$, $\emptyset \neq F_i \subsetneq F_{i+1} \subseteq F$ et $(E_i, F_j) \in \rho_{\mathcal{R}}$ pour tout $j \leq i$.

2. *Pour une lettre $f_1 = a$:*

$$E^* a (E'^* \sqcup\!\sqcup_{\mathcal{R}} f_2 \cdots f_n),$$

   où $E' = \{x \in E \mid (x,a) \in \rho_{\mathcal{R}}\}$.

L'algorithme de calcul de la clôture d'une expression APC $\sum e_1 \cdots e_n$ par une relation de semi-commutation $\mathcal{R}$ est le suivant : Nous calculons inductivement

$$\mathcal{R}^*(e_2 \cdots e_n) = \sum f_1 \cdots f_k.$$

L'étape d'induction est donnée par le lemme 3.1.4, si e_1 est une lettre. Sinon, pour $e_1 = E^*$ nous appliquons la proposition 3.1.4, qui est en soit une étape d'induction. Il est facile de voir que chaque étape préserve l'inclusion dans APC. Ceci montre le théorème 3.1.2.

Corollaire 3.1.2 *Soit p un produit de longueur n, alors $l(\mathcal{R}^*(p)) \in \mathcal{O}(n\delta_{\mathcal{R}})$, et $taille(\mathcal{R}^*(p)) \in 2^{O(|\Sigma| n \delta_{\mathcal{R}})}$.*

Preuve : Il est facile de voir que chaque étape de récursion rajoute au plus $\delta_{\mathcal{R}}$ expressions atomiques dans chaque produit. Puisqu'il existe n étapes de récursion, il s'en suit que la longueur des produits de l'expression calculée pour $\mathcal{R}^*(p)$ est au plus $\mathcal{O}(n\delta_{\mathcal{R}})$. En plus, puisqu'il existe $2^{|\Sigma|} + |\Sigma|$ expressions atomiques différentes, il s'en suit que la taille de $\mathcal{R}^*(p)$ est au plus $2^{O(|\Sigma| n\delta_{\mathcal{R}})}$. □

Nous illustrons le calcul de la clôture d'un produit par une relation de semi-commutation par l'exemple suivant :

Exemple 3.1.5 *Considérons le produit $p = h(a+b+c)^*(e+f+g)^*d$, et la relation de semi-commutation $\mathcal{R}$ définie par*

$$\rho_{\mathcal{R}} = \{(a,e),(h,a),(h,e),(c,g),(b,e),(b,f),(f,d),(g,d),(a,d)\}$$

Alors $\mathcal{R}^*(p) = h \sqcup\!\sqcup_{\mathcal{R}} \Big((a+b+c)^* \sqcup\!\sqcup_{\mathcal{R}} \big((e+f+g)^* \sqcup\!\sqcup_{\mathcal{R}} d \big) \Big)$. En utilisant les exemples précédents nous obtenons

$\mathcal{R}^*(p) = a^*h(a+b+c)^*(c+g)^*(e+f+g)^*d(f+g)^* +$
$a^*h(a+b+c)^*(a+b+e)^*(b+e+f)^*(e+f+g)^*d(f+g)^* +$
$a^*h(a+b+c)^*(a+b+e)^*da^*(f+g)^* +$
$(a+e)^*h(e+f+g)^*d(f+g)^* +$
$(a+e)^*h(a+b+e)^*(b+e+f)^*(e+f+g)^*d(f+g)^* +$
$(a+e)^*h(a+b+e)^*da^*(f+g)^*.$

Remarque 3.1.4 *Observons que APC est le plus haut niveau dans les hiérarchies de Straubing-Thérien et la hiérarchie Σ_n qui est fermée par semi-commutations. En effet, $(ab)^*$ est le complément d'un langage dans APC. Ainsi, il appartient au niveau 2 de la hiérarchie de Straubing-Thérien, et peut être décrit par une Σ_3-formule. La clôture de $(ab)^*$ par la relation de semi-commutations donnée par $\{(a,b),(b,a)\}$ est bien sûr non régulière (elle est égale à l'ensemble $\{w \in (a+b)^* \mid |w|_a = |w|_b\}$).*

Nous pouvons montrer que décider si une APC est fermée par une relation de semi-commutations est PSPACE-complet :

Théorème 3.1.3 *[BMT01] Le problème suivant est PSPACE-complet :*
Entrée : *Une expression APC $\mathcal{L}$ sur Σ et une relation de semi-commutation $\mathcal{R} \subseteq \Sigma \times \Sigma$.*
Question : *A-t-on $\mathcal{R}^*(\mathcal{L}) = \mathcal{L}$?*

3.1.2.3 Application : Analyse du contrôleur d'ascenceur

Reprenons l'exemple du contrôleur d'ascenceur vu à la section 3.1.1. Le tableau 3.1 montre les calculs des configurations accessibles du système. La première colonne représente l'ensemble de configurations auquels nous appliquons l'action donnée dans la deuxième colonne. Les ensembles obtenus sont représentés dans les colonnes 3 et 4. Puisque l'ensemble des accessibles est une APC, l'image par $\mathsf{request}^*$ est facilement calculée (les APC sont effectivement clos par substitutions), et les images par $\mathsf{move\text{-}up}^*$ et $\mathsf{move\text{-}down}^*$ peuvent être calculées par notre algorithme. Notons que le fait que la

L_0	request*	$\#A\uparrow\ (\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$\mathcal{L}_1$
$\mathcal{L}_1$	move-up*	$\#(\bot + p\downarrow)^*A\uparrow\ (\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$\mathcal{L}_2$
$\mathcal{L}_2$	request*	$\#(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*A\uparrow\ (\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$\mathcal{L}_3$
$\mathcal{L}_3$	take-up	$\#(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*(\bot + p\downarrow)A\uparrow\ (\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$\subseteq \mathcal{L}_3$
$\mathcal{L}_3$	up2down	$\#(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*A\downarrow\ \#$	$\mathcal{L}_4$
$\mathcal{L}_4$	move-down*	$\#(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*A\downarrow\ (\bot + p\uparrow)^*\#$	$\mathcal{L}_5$
$\mathcal{L}_5$	request*	$\#(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*A\downarrow\ (\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$\mathcal{L}_6$
$\mathcal{L}_6$	take-down	$\#(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*A\downarrow\ (\bot + p\uparrow)(\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$\subseteq \mathcal{L}_6$
$\mathcal{L}_6$	down2up	$\#A\uparrow\ (\bot + p\uparrow + p\downarrow + p\uparrow\downarrow)^*\#$	$= \mathcal{L}_1$

TAB. 3.1 – Analyse d'accessibilité du contrôleur d'ascenceur

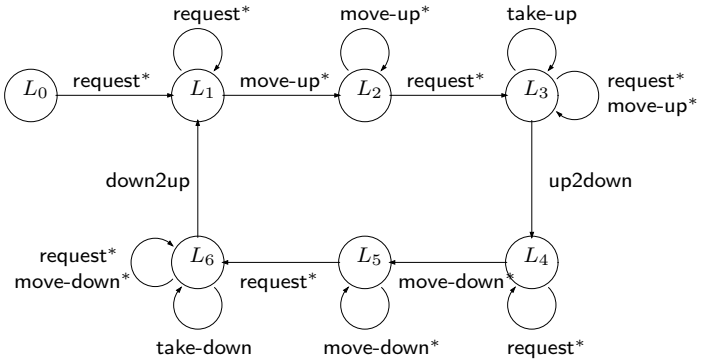


FIG. 3.1 – Graphe d'accessiblité symbolique du contrôleur d'ascenceur

fermeture par move-up* reste une APC est crucial pour pouvoir calculer la clôture par move-down* en appliquant notre algorithme.

Comme montré dans le tableau 3.1, notre algorithme pour les APC a permis de calculer l'ensemble des accessibles du contrôleur d'ascenceur. L'analyse précédente permet également de calculer une abstraction finie de ce système infini. En effet, le tableau 3.1 définit un graphe d'accessiblité symbolique du contrôleur d'ascenceur (figure 3.1).

3.1.3 Systèmes paramétrés à topologie circulaire

Dans cette section[2], nous considérons des systèmes paramétrés à topologie circulaire. Dans ce cas, une configuration est représentée par un mot circulaire, c-à-d., un mot $x_1 \cdots x_n$ avec la signification que x_1 suit x_n. Ceci signifie que $x_1 \cdots x_n$ et les mots $x_k x_{k+1} \cdots x_n x_1 \cdots x_{k-1}$ représentent la même configuration. Ainsi, l'ensemble de configurations d'un système à topologie circulaire est un ensemble de mots $\mathcal{L}$ qui

[2]Les résultats présentés dans cette section sont dûs à Anca Muscholl, et ont été présentés dans [BMT01].

est fermé par conjugaison, c-à-d., tel que $\mathcal{L} = Conj(\mathcal{L})$, où cette notion de conjugaison est définie comme suit :

Définition 3.1.2 (Conjugaison) *Deux mots x et $y \in \Sigma^*$ sont dits* conjugués *si $x = uv$ et $y = vu$ pour certains $u, v \in \Sigma^*$. Pour un langage $\mathcal{L}$, nous dénotons par $Conj(\mathcal{L})$ l'ensemble $\{uv \in \Sigma^* \mid vu \in \mathcal{L}\}$ des conjugués des mots de $\mathcal{L}$. Une classe de langages $\mathcal{C}$ est* fermée par conjugaison *si $\mathcal{L} \in \mathcal{C}$ implique que $Conj(\mathcal{L}) \in \mathcal{C}$.*

Considérons par exemple le *token ring protocol* où un nombre arbitraire de processus sont disposés en anneau et font passer un jeton de la gauche vers la droite. Chaque processus est soit dans l'état 0 s'il n'a pas le jeton, soit dans l'état 1 s'il l'a. L'ensemble de configurations du système peut donc être représenté par le langage $\mathcal{L} = 0^*10^*$ signifiant qu'à chaque étape, un seul processus détient le jeton. $\mathcal{L}$ est bien fermé par conjugaison puisque $0^*10^* = Conj(0^*10^*)$. L'action du système qui fait passer le jeton d'un processus à son voisin de droite peut être représentée (comme dans le cas du *token passing protocol*, qui est la version linéaire de ce protocole décrite dans l'introduction de la thèse) par la relation de semi-commutations $10 \rightarrow 01$.

Pour pouvoir analyser ces systèmes à topologie circulaire, il est donc important de savoir calculer $\mathcal{R}^*(\mathcal{L})$, pour une relation de semi-commutations $\mathcal{R}$, et un ensemble de mots circulaires $\mathcal{L}$. C-à-d., pour un ensemble $\mathcal{L}$ fermé par conjugaison. Ceci revient à calculer $\mathcal{R}_c^*(\mathcal{L})$ pour un langage $\mathcal{L}$ quelconque, où $\mathcal{R}_c$ est la *relation de réécriture circulaire* $\mathcal{R}_c = \mathcal{R}^* \circ Conj \circ \mathcal{R}^*$.

Nous pouvons montrer que la $\mathcal{R}$-clôture circulaire $\mathcal{R}_c^*(\mathcal{L})$ de *n'importe quel* langage $\mathcal{L}$ (pas nécessairement régulier) peut être obtenue en appliquant alternativement conjugaison et semi-commutations un nombre fini de fois :

Théorème 3.1.4 *[BMT01] Soit $\mathcal{L} \subseteq \Sigma^*$, alors $\mathcal{R}_c^*(\mathcal{L}) = \mathcal{R}_c^{2|\Sigma|}(\mathcal{L})$.*

Nous obtenons alors que la classe APC est fermée par réécriture circulaire :

Corollaire 3.1.3 *Soit $\mathcal{L}$ une expression APC, alors $\mathcal{R}_c^*(\mathcal{L})$ est dans APC et peut être calculé effectivement.*

Preuve : Ceci est dû au fait que $\mathcal{R}_c^*(\mathcal{L}) = \mathcal{R}_c^{2|\Sigma|}(\mathcal{L}) = (\mathcal{R}^* \circ Conj \circ \mathcal{R}^*)^{2|\Sigma|}(\mathcal{L})$, que APC est effectivement fermée par semi-commutations (théorème 3.1.2), et qu'elle est aussi effectivement fermée par conjugaison. Pour montrer ce dernier point, il suffit de montrer que pour chaque produit p, l'ensemble des conjugués $Conj(p)$ est une APC. Soit donc un produit $p = e_1 \cdots e_n$, il est facile de voir que :

$$Conj(p) = \sum_i e_i \cdots e_n e_1 \cdots e_{i-1} e'_i$$

tels que :
- $e'_i = e_i$ si e_i est une expression étoile,
- $e'_i = \epsilon$ si e_i est une lettre.

Puisque l'expression précédente est une APC, le résultat suit.

□

3.2 Systèmes paramétrés à topologies arborescentes

Dans le cas où les processus sont disposés dans une architecture arborescente, une configuration est représentée par un arbre étiqueté dont chaque nœud correspond à un processus, et l'étiquette de ce nœud correspond à l'état de contrôle de ce processus. Par conséquent, un ensemble de configurations peut être représenté par un langage régulier d'arbres [KMM+97]. Typiquement, les actions dans de tels systèmes sont des communications entre les processus et leurs pères et fils. Ces actions peuvent être vues comme des transformations qui modifient les étiquettes des arbres, c-à-d., elles peuvent être représentées par des relations de réétiquetage.

3.2.1 Exemples de réseaux paramétrés à architecture arborescente

Nous montrons dans cette section deux exemples de modélisation de protocoles définis sur des sytèmes paramétrés ayant une topologie arborescente. Un autre exemple est donné au chapitre 9.

3.2.1.1 Un algorithme pour calculer le OU

Prenons l'exemple d'un programme booléen, appelé PERCOLATE [KMM+97], qui calcule le OU d'un ensemble de valeurs booléennes : Nous considérons un nombre arbitraire de processus disposés en une topologie arborescente. Chaque processus a une variable *val* appartenant à $\{0, 1, \perp\}$. Initialement, toutes les feuilles ont une valeur $val \in \{0, 1\}$, et tous les autres nœuds ont $val = \perp$. Le but du programme est de faire passer à la racine la valeur 1 si au moins une des feuilles a $val = 1$. Une transition du système consiste à associer 1 à un nœud si l'un de ses fils a $val = 1$, et 0 sinon. Ceci peut être modélisé par la relation de réétiquetage $\mathcal{R}_{percolate}$ qui contient les paires de termes $(t, t') \in T_\Sigma$ où $\Sigma = \{0, 1, \perp\}$ tels qu'il existe un contexte C, et quatre termes t_1, t_2, t_3, et t_4 tels que :

- $t = C\Big[\perp\big(1(t_1, t_2), 1(t_3, t_4)\big)\Big]$ et $t' = C\Big[1\big(1(t_1, t_2), 1(t_3, t_4)\big)\Big]$, ou
- $t = C\Big[\perp\big(1(t_1, t_2), 0(t_3, t_4)\big)\Big]$ et $t' = C\Big[1\big(1(t_1, t_2), 0(t_3, t_4)\big)\Big]$, ou
- $t = C\Big[\perp\big(0(t_1, t_2), 1(t_3, t_4)\big)\Big]$ et $t' = C\Big[1\big(0(t_1, t_2), 1(t_3, t_4)\big)\Big]$, ou
- $t = C\Big[\perp\big(0(t_1, t_2), 0(t_3, t_4)\big)\Big]$ et $t' = C\Big[0\big(0(t_1, t_2), 0(t_3, t_4)\big)\Big]$.

Pour vérifier que ce programme calcule bien le OU des valeurs booléennes disposées aux feuilles, il faut calculer l'ensemble des accessibles par $\mathcal{R}_{percolate}$ de l'ensemble initial, qui est l'ensemble régulier des termes ayant 1 ou 0 aux feuilles et $\perp$ dans les autres nœuds, et vérifier que cet ensemble n'intersecte pas l'ensemble des mauvaises configurations, c-à-d, l'ensemble régulier des termes qui n' ont que des 0 aux feuilles et 1 à la racine ; ou qui ont au moins une feuille étiquetée par 1 et 0 à la racine.

3.2.1.2 L'arbre de parité [CGJ95]

Ici aussi, nous considérons des arbres binaires où les feuilles ont les valeurs 0 et 1. Le programme consiste à calculer la parité de la somme des valeurs aux feuilles et la transmettre à la racine. Une transition du système consiste à attribuer 1 à un nœud si exactement un de ses fils a la valeur 1, et 0 si les deux ont 0 ou si les deux ont 1 comme valeurs. Cette transition peut être représentée par la relation $\mathcal{R}_{parité}$ qui contient les paires de termes $(t, t') \in T_\Sigma$ où $\Sigma = \{0, 1, \perp\}$ tels qu'il existe un contexte C, et quatre termes t_1, t_2, t_3, et t_4 tels que :

- $t = C\Big[\perp\big(1(t_1, t_2), 1(t_3, t_4)\big)\Big]$ et $t' = C\Big[0\big(1(t_1, t_2), 1(t_3, t_4)\big)\Big]$, ou
- $t = C\Big[\perp\big(1(t_1, t_2), 0(t_3, t_4)\big)\Big]$ et $t' = C\Big[1\big(1(t_1, t_2), 0(t_3, t_4)\big)\Big]$, ou
- $t = C\Big[\perp\big(0(t_1, t_2), 1(t_3, t_4)\big)\Big]$ et $t' = C\Big[1\big(0(t_1, t_2), 1(t_3, t_4)\big)\Big]$, ou
- $t = C\Big[\perp\big(0(t_1, t_2), 0(t_3, t_4)\big)\Big]$ et $t' = C\Big[0\big(0(t_1, t_2), 0(t_3, t_4)\big)\Big]$.

Il s'agit alors de vérifier que la racine est étiquetée par 1 ssi le nombre de feuilles étiquetées par 1 est impair. Cette propriété est représentable par un automate d'arbres fini. Donc, comme précédemment, vérifier la correction de cet algorithme peut se faire en calculant l'ensemble des accessibles par $\mathcal{R}_{parité}$.

3.2.2 Well-Oriented Systems

Nous avons essayé d'étendre le travail précédent aux langages réguliers d'arbres pour pouvoir calculer l'ensemble des accessibles des systèmes à topologies arborescentes par des "semi-commutations d'arbres". En effet, ce genre de relations permet de modéliser la communication entre père et fils dans les réseaux à architectures arborescentes. Nous avons alors essayé de trouver une classe significative de langages réguliers d'arbres (binaires par exemple) analogue aux APCs qui soit effectivement fermée par "semi-commutations d'arbres", c-à-d., par des relations de la forme

$$\mathcal{R}_{a,b} = \{(t, t') \mid t = C\Big[a\big(b(t_1, t_2), t_3\big)\Big], t' = C\Big[b\big(a(t_1, t_2), t_3\big)\Big]\} \cup$$

$$\{(t, t') \mid t = C\Big[a\big(t_1, b(t_2, t_3)\big)\Big], t' = C\Big[b\big(t_1, a(t_2, t_3)\big)\Big]\},$$

où C est un contexte et t_1, t_2, et t_3 sont des termes. Seulement, même si une telle classe existe, elle ne serait pas intéressante pour la modélisation des systèmes paramétrés arborescents. En effet, pour ces systèmes, on distingue souvent les feuilles des autres nœuds, c-à-d., on s'intéresse souvent aux langages contenant des termes où toutes les feuilles sont dans un certain état b et tous les autres nœuds sont dans un autre état a. Par exemple, on aimerait vérifier si les protocoles décrits ci-dessus sont corrects dans le cas où initialement toutes les feuilles ont $val = 1$ et tous les autres nœuds sont à l'état $\perp$. Seulement, si nous considérons le langage régulier d'arbres dont les termes sont tels que des feuilles sont étiquetées par b et les nœuds par a (ce langage est reconnu par l'automate d'arbres $\mathcal{A} = (Q, \Sigma, F, \delta)$ où $Q = F = \{q\}$, $\Sigma = \Sigma_0 = \Sigma_2 = \{a, b\}$, et δ contient les règles $b \to q$ et $a(q, q) \to q$). La fermeture de ce langage par les semi-commutations $\mathcal{R}_{a,b} \cup \mathcal{R}_{b,a}$ n'est pas régulière et est égale à l'ensemble des termes

sur l'alphabet $\{a, b\}$ dont le nombre de nœuds étiquetés par b est égal au nombre de feuilles.

Pour contourner ce problème, nous proposons d'affaiblir la classe de relations considérée, tout en permettant aux pères/fils de communiquer. Nous introduisons alors une classe de relations de réétiquetage, appelée la classe des Well-Oriented Systems, qui est assez significative pour pouvoir modéliser les communications dans ces systèmes, et nous montrons que la clôture réflexive-transitive d'une relation de cette classe est une relation de réétiquetage qui peut être effectivement calculée.

3.2.2.1 Définition des Well-Oriented Systems

Plusieurs protocoles et algorithmes parallèles définis sur des réseaux à structure arborescente satisfont les caractéristiques suivantes : (1) les informations montent des feuilles vers la racine et vice versa, ce qui veut dire que chaque nœud communique directement soit avec ses fils soit avec son père, (2) il existe un nombre fini d'alternances entre les phases de propagation ascendantes et descendantes des informations (ex., les demandes sont envoyées par les feuilles, et ensuite les réponses sont envoyées par la racine, et ainsi de suite), (3) l'état de chaque processus est modifié après chaque transmission de l'information, c-à-d., à chaque phase, quand un nœud du réseau est traversé, il est marqué par une nouvelle étiquette. Ceci correspond par exemple au marquage des chemins de passage, mémorisation des messages envoyés, etc.

Nous introduisons un modèle pour décrire la dynamique de tels réseaux paramétrés arborescents, qui consiste en des systèmes de réécriture de termes appelés *well-oriented systems*. Pour simplifier la présentation, nous nous restreignons aux arbres binaires, le cas général est similaire.

Définition 3.2.1 *Soit $\mathcal{S} = \mathcal{S}_0 \cup \mathcal{S}_1 \cup \cdots \cup \mathcal{S}_n$, où les $\mathcal{S}_i$ sont des ensembles finis de symboles qui sont disjoints.*

Un n-phase well-oriented system *(n-phase WOS) sur $\mathcal{S}$ est un ensemble de règles de réécriture de la forme :*

$$
\begin{aligned}
b(a(x_1,x_2),c_1(x_3,x_4)) &\rightarrow a(b'(x_1,x_2),c_1(x_3,x_4)) && (3.17)\\
a(b(x_1,x_2),c_1(x_3,x_4)) &\rightarrow b'(a(x_1,x_2),c_1(x_3,x_4)) && (3.18)\\
a(b(x_1,x_2),c_2(x_3,x_4)) &\rightarrow b'(a(x_1,x_2),a(x_3,x_4)) && (3.19)\\
b(x_1,x_2) &\rightarrow d(x_1,x_2) && (3.20)\\
b(a(x_1,x_2),c_1(x_3,x_4)) &\rightarrow d(a(x_1,x_2),c_1(x_3,x_4)) && (3.21)\\
a(b(x_1,x_2),c_1(x_3,x_4)) &\rightarrow a(d(x_1,x_2),c_1(x_3,x_4)) && (3.22)\\
a(b(x_1,x_2),c_2(x_3,x_4)) &\rightarrow a(d(x_1,x_2),d(x_3,x_4)) && (3.23)
\end{aligned}
$$

Ainsi que les formes symétriques de ces règles obtenues en permutant les fils gauches et droits, où $a, b', c_1 \in \mathcal{S}_{i+1}$, $b, c_2 \in \mathcal{S}_i$, et $d \in \mathcal{S}_{i+2}$, tels que $0 \leq i \leq n-1$ pour les règles (3.17), (3.18), et (3.19), et $0 \leq i \leq n-2$ pour les dernières règles.

Le système ci-dessus induit une relation de réétiquetage $\mathcal{R}_\mathcal{S}$ sur les arbres définie par : $(t, t') \in \mathcal{R}_\mathcal{S}$ s'il existe un contexte C, et :

– soit une règle $b(x_1, x_2) \to d(x_1, x_2)$ de $\mathcal{S}$, et deux termes u_1 et u_2 tels que

$$t = C[b(u_1, u_2)] \text{ et } t' = C[d(u_1, u_2)],$$

– soit une règle $a(b(x_1, x_2), c(x_3, x_4)) \to d(e(x_1, x_2), f(x_3, x_4))$ de $\mathcal{S}$, et quatre termes u_1, u_2, u_3 et u_4 tels que

$$t = C[a(b(u_1, u_2), c(u_3, u_4))] \text{ et } t' = C[d(e(u_1, u_2), f(u_3, u_4))].$$

Dans la suite, nous ne faisons pas de distinction entre une règle de réécriture d'un système WOS $\mathcal{S}$, et la relation de réétiquetage $\mathcal{R}_\mathcal{S}$ qui lui correspond.

Intuitivement, une règle (3.17) correspond à la montée de a. Quand a traverse b, elle prend sa place et étiquette son ancienne place par b' pour marquer son chemin. L'ensemble de ces règles sera dénoté par $\mathcal{R}^{\uparrow}_{i+1}$ (elles correspondent à la montée des lettres a de $\mathcal{S}_{i+1}$). De manière similaire, une règle (3.18) correspond à la descente de a, et une règle (3.19) correspond au broadcasting de a. L'ensemble de ces règles sera dénoté par $\mathcal{R}^{\downarrow}_{i+1}$ (elles correspondent à la descente des lettres a de $\mathcal{S}_{i+1}$). Finalement, les quatre dernières règles permettent de passer d'une phase à une autre. Plus précisément, les règles (3.20) correspondent à un passage inconditionnel, et les règles (3.21), (3.22) et (3.23) à un passage conditionnel. Cet ensemble de règles sera dénoté par $\mathcal{R}_{i+1\to i+2}$ (elles correspondent au passage de la phase $i+1$ à la phase $i+2$, c-à-d., de la propagation des symboles de $\mathcal{S}_{i+1}$ à la propagation des symboles de $\mathcal{S}_{i+2}$).

3.2.2.2 Exemples de WOSs

Le système $\mathcal{R}_{percolate}$ donné à la section 3.2.1.1 est un 1-phase WOS où $\mathcal{S}_0 = \{\bot\}$, et $\mathcal{S}_1 = \{0, 1\}$, et l'ensemble des règles de réécriture correspondant à $\mathcal{R}_{percolate}$ est donné par :

$$\begin{aligned}
\bot(1(x_1, x_2), 1(x_3, x_4)) &\to 1(1(x_1, x_2), 1(x_3, x_4)) \\
\bot(1(x_1, x_2), 0(x_3, x_4)) &\to 1(1(x_1, x_2), 0(x_3, x_4)) \\
\bot(0(x_1, x_2), 1(x_3, x_4)) &\to 1(0(x_1, x_2), 1(x_3, x_4)) \\
\bot(0(x_1, x_2), 0(x_3, x_4)) &\to 0(0(x_1, x_2), 0(x_3, x_4))
\end{aligned}$$

De même, le système $\mathcal{R}_{parité}$ donné à la section 3.2.1.2 est un 1-phase WOS où $\mathcal{S}_0 = \{\bot\}$, et $\mathcal{S}_1 = \{0, 1\}$, et l'ensemble des règles de réécriture correspondant à $\mathcal{R}_{parité}$ est donné par :

$$\begin{aligned}
\bot(1(x_1, x_2), 1(x_3, x_4)) &\to 0(1(x_1, x_2), 1(x_3, x_4)) \\
\bot(1(x_1, x_2), 0(x_3, x_4)) &\to 1(1(x_1, x_2), 0(x_3, x_4)) \\
\bot(0(x_1, x_2), 1(x_3, x_4)) &\to 1(0(x_1, x_2), 1(x_3, x_4)) \\
\bot(0(x_1, x_2), 0(x_3, x_4)) &\to 0(0(x_1, x_2), 0(x_3, x_4))
\end{aligned}$$

3.2.3 Analyser les Well-Oriented Systems

Nous montrons dans cette section que pour tout well-oriented system $\mathcal{R}$, $\mathcal{R}^*$ est une relation de réétiquetage régulière qui peut être effectivement construite.

Lemme 3.2.1 *Soit $\mathcal{R}$ un n-phase well-oriented system. Alors*

$$\mathcal{R}^* = \mathcal{R}_n^* \circ \mathcal{R}_{n-1\to n}^* \circ \mathcal{R}_{n-1}^* \circ \cdots \circ \mathcal{R}_{1\to 2}^* \circ \mathcal{R}_1^*.$$

Preuve : L'application d'une des règles (3.17)–(3.23) augmente toujours l'indice des étiquettes des nœuds de l'arbre auxquels la règle a été appliquée. Cette propriété, avec le fait que les contextes "c_1" sont dans $\mathcal{S}_{i+1}$ assure que la phase $i+1$ ($a \in \mathcal{S}_{i+1}$) dépend juste des phases précédentes $j \leq i+1$. Par conséquent, il est facile de voir que $\mathcal{R}^* = \mathcal{R}_n^* \circ \mathcal{R}_{n-1\to n}^* \circ \mathcal{R}_{n-1}^* \circ \cdots \circ \mathcal{R}_{1\to 2}^* \circ \mathcal{R}_1^*$. □

Donc, pour calculer $\mathcal{R}^*$, il suffit de calculer les clôtures transitives $\mathcal{R}_{i\to i+1}^*$, pour $1 \leq i < n$ et $\mathcal{R}_i^*$ pour $1 \leq i \leq n$. Nous construisons ci-dessous des réétiquetages correspondant à ces relations.

3.2.3.1 Construction de $\mathcal{R}_i^*$

Nous donnons dans ce qui suit un réétiquetage qui reconnaît $\mathcal{R}_i^*$. Soit $\mathcal{A}_i$ le réétiquetage $\mathcal{A}_i = (Q, \mathcal{S}, \mathcal{S}, F, \delta)$ où $Q = \{q\} \cup \{q_c \mid c \in \mathcal{S}_i\} \cup \{q_{b'}^a, q_a^b \mid a, b' \in \mathcal{S}_i, b \in \mathcal{S}_{i-1}\}$, $F = \{q\}$, et δ est l'ensemble des règles de transition suivantes :

(α_1) $a/a \to q$ et $a/a(q,q) \to q$, pour tout $a \in \mathcal{S}$,
(α_2) $c/c \to q_c$ et $c/c(q,q) \to q_c$, pour tout $c \in \mathcal{S}_i$,
(α_3) $q_a \to q$, pour tout $a \in \mathcal{S}_i$,
et tel que :

– si $\mathcal{R}_i^{\uparrow}$ contient la règle

$$b(a(x_1,x_2), c(x_3,x_4)) \to a(b'(x_1,x_2), c(x_3,x_4)),$$

où $a, b', c \in \mathcal{S}_i$, et $b \in \mathcal{S}_{i-1}$, alors la relation de transition δ contient :

(α_4) $a/b'(q,q) \to q_{b'}^a$,
(α_5) $a/b' \to q_{b'}^a$,
(α_6) $b/a(q_{b'}^a, q_c) \to q_a$,
(α_7) $b/b'(q_{b'}^a, q_c) \to q_{b'}^a$,
(α_8) $b/d'(q_{b'}^a, q_c) \to q_{d'}^a$, si $\mathcal{R}_i^{\uparrow}$ contient la règle

$$d(a(x_1,x_2), e(x_3,x_4)) \to a(d'(x_1,x_2), e(x_3,x_4)),$$

(α_9) $a/f'(q_{b'}^f, q_g) \to q_{b'}^a$, si $\mathcal{R}_i^{\downarrow}$ contient la règle

$$b'(f(x_1,x_2), g(x_3,x_4)) \to f'(b'(x_1,x_2), g(x_3,x_4)),$$

(α_{10}) $a/f'(q_{b'}^f, q_{b'}^g) \to q_{b'}^a$, si $\mathcal{R}_i^{\downarrow}$ contient la règle

$$b'(f(x_1,x_2), g(x_3,x_4)) \to f'(b'(x_1,x_2), b'(x_3,x_4)).$$

– si $\mathcal{R}_i^{\downarrow}$ contient la règle $a(b(x_1,x_2), c(x_3,x_4)) \to b'(a(x_1,x_2), c(x_3,x_4))$, où $a, b', c \in \mathcal{S}_i$, et $b \in \mathcal{S}_{i-1}$, alors la relation de transition δ contient :

(α_{11}) $b/a \to q_a^b$,

(α_{12}) $b/a(q,q) \to q_a^b$,

(α_{13}) $a/b'(q_a^b, q_c) \to q_a$, $a/b'(q_a^b, q_c) \to q_{b'}$,

(α_{14}) $b/b'(q_a^b, q_c) \to q_a^b$,

(α_{15}) $b/d'(q_a^d, q_e) \to q_a^b$, si $\mathcal{R}_i^{\downarrow}$ contient la règle

$$a(d(x_1,x_2), e(x_3,x_4)) \to d'(a(x_1,x_2), e(x_3,x_4)),$$

(α_{16}) $b/f'(q_a^f, q_a^g) \to q_a^b$, si $\mathcal{R}_i^{\downarrow}$ contient la règle

$$a(f(x_1,x_2), g(x_3,x_4)) \to f'(a(x_1,x_2), a(x_3,x_4)),$$

(α_{17}) $a/h'(q_a^b, q_c) \to q_{h'}^{b'}$, si $\mathcal{R}_i^{\uparrow}$ contient la règle

$$h(b'(x_1,x_2), k(x_3,x_4)) \to b'(h'(x_1,x_2), k(x_3,x_4)),$$

– si $\mathcal{R}_i^{\downarrow}$ contient la règle $a(b(x_1,x_2), c(x_3,x_4)) \to b'(a(x_1,x_2), a(x_3,x_4))$, où $a, b' \in \mathcal{S}_i$, et $b, c \in \mathcal{S}_{i-1}$, alors la relation de transition δ contient :

(α_{18}) $b/a \to q_a^b$,

(α_{19}) $b/a(q,q) \to q_a^b$,

(α_{20}) $c/a \to q_a^c$,

(α_{21}) $c/a(q,q) \to q_a^c$,

(α_{22}) $a/b'(q_a^b, q_a^c) \to q_a$, $a/b'(q_a^b, q_a^c) \to q_{b'}$,

(α_{23}) $b/b'(q_a^b, q_a^c) \to q_a^b$,

(α_{24}) $c/b'(q_a^b, q_a^c) \to q_a^c$,

(α_{25}) $b/d'(q_a^d, q_e) \to q_a^b$, $c/d'(q_a^d, q_e) \to q_a^c$, si $\mathcal{R}_i^{\downarrow}$ contient la règle

$$a(d(x_1,x_2), e(x_3,x_4)) \to d'(a(x_1,x_2), e(x_3,x_4)),$$

(α_{26}) $b/f'(q_a^f, q_a^g) \to q_a^b$, et $c/f'(q_a^f, q_a^g) \to q_a^c$, si $\mathcal{R}_i^{\downarrow}$ contient la règle

$$a(f(x_1,x_2), g(x_3,x_4)) \to f'(a(x_1,x_2), a(x_3,x_4)),$$

(α_{27}) $a/h'(q_a^b, q_a^c) \to q_{h'}^{b'}$, si $\mathcal{R}_i^{\uparrow}$ contient la règle

$$h(b'(x_1,x_2), k(x_3,x_4)) \to b'(h'(x_1,x_2), k(x_3,x_4)).$$

Puisque l'ordre entre les fils n'est pas important, l'automate a pour chaque règle de la forme $f/g(q_1,q_2) \to q$ une règle symétrique $f/g(q_2,q_1) \to q$ qui n'est pas représentée ci-dessus.

Intuition. L'automate ci-dessus reconnaît tous les termes de la forme t/t' tels que t et t' soient deux termes vérifiant $t' \in \mathcal{R}_i^*(t)$. L'automate se comporte comme suit : C'est l'état q qui est l'état final, et qui reconnaît donc ces termes. Les règles α_1 expriment que pour tout $a \in \mathcal{S}$, $a/a \in \mathcal{R}_i^*$ puisque $a/a \in Id_{T_S}$, et que si $t_1/t'_1 \in \mathcal{R}_i^*$ et $t_2/t'_2 \in \mathcal{R}_i^*$, alors $a/a(t_1/t'_1, t_2/t'_2) \in \mathcal{R}_i^*$.
Un nœud n dans t/t' est annoté par q_a, $a \in \mathcal{S}_i$ si :

- n est étiqueté par a/a (règles α_2). Ceci veut dire qu'aucune réécriture n'a été faite à cette place c-à-d., l'étiquette de ce nœud est la même dans t et t'.
- il existe $b \in \mathcal{S}_i$ tel que n est étiqueté par b/a (règles α_6). Ceci veut dire qu'une réécriture a été faite à ce nœud en appliquant une règle $b(a(x_1, x_2), c(x_3, x_4)) \rightarrow a(b'(x_1, x_2), c(x_3, x_4))$. Le nœud n est étiqueté par b dans t et par a dans t'. L'état q_c dans la partie gauche de la règle α_6 impose que b ne peut traverser a que si son "frère" est c.
- il existe $b' \in \mathcal{S}_i$ tel que n est étiqueté par a/b'. Ceci veut dire qu'une réécriture a été faite à ce nœud en utilisant soit une règle

$$a(b(x_1, x_2), c(x_3, x_4)) \rightarrow b'(a(x_1, x_2), c(x_3, x_4))$$

(règles α_{13}), soit une règle

$$a(b(x_1, x_2), c(x_3, x_4)) \rightarrow b'(a(x_1, x_2), a(x_3, x_4))$$

(règles α_{22}). Ce nœud peut aussi être annoté par $q_{b'}$ (règles α_{13} et α_{22}).

Dans tous ces cas, un terme t/t' de racine n est annoté par q_a ($a \in \mathcal{S}_i$) si t/t' est dans $\mathcal{R}_i^*$ (ceci explique la règle α_3), et que n est étiqueté par a soit dans t soit dans t', c-à-d., si à cette position il y "avait" ou il y "a" un "a". Nous avons besoin d'annoter un tel nœud par q_a pour savoir qu'il y a (avait) un "a" à cette position. Ceci est utile si a est dans le contexte d'une règle de $\mathcal{R}_i$ (s'il est le frère d'un nœud où une réécriture peut se faire). Notons que les règles α_6 annotent les nœuds étiquetés par b/a seulement avec q_a et pas avec q_b. Ceci est dû au fait que $b \in \mathcal{S}_{i-1}$, et donc "b" ne peut pas être le contexte d'une autre règle dans $\mathcal{R}_i$, par conséquent, il est inutile de mémoriser qu'à cette position nous avions un "b".

Maintenant, nous expliquons comment l'automate utilise ces règles pour annoter un terme de $\mathcal{R}_i^*$. Soient trois termes t_1, t_2, et t_3 tels que $t_2 \in \mathcal{R}_i^*(t_1)$ et $t_3 \in \mathcal{R}_i^*(t_2)$. L'automate doit reconnaître t_1/t_3. Soit u l'arbre correspondant à ces termes. Et soient n_1, n_2 et n_3 trois nœuds de u tels que n_1 soit la racine de n_2 et n_3, n_2 étant à gauche de n_3. Il y a plusieurs cas :

- Dans t_1, n_1 est étiqueté par b, n_2 par a, n_3 par c, et une règle

$$b(a(x_1, x_2), c(x_3, x_4)) \rightarrow a(b'(x_1, x_2), c(x_3, x_4))$$

s'applique à ces trois nœuds pour donner t_2. Nous aurons alors que dans t_2, n_1 est étiqueté par a, n_2 par b', et n_3 par c. Ensuite, d'autres réécritures s'appliquent à t_2 pour obtenir t_3. Nous montrons comment l'automate annote ces nœuds pour le terme t_1/t_3 :

- Noeud n_1. Il y a deux cas :

1. n_1 est étiqueté par a dans t_3, c-à-d., a ne bouge pas de ce nœud. Dans ce cas, n_1 est étiqueté par b/a dans t_1/t_3, et est annoté en utilisant les règles α_6 (nous allons voir que n_2 est annoté par $q^a_{b'}$, et n_3 par q_c).
2. a continue à monter (elle traverse la racine de n_1,...) en appliquant la même règle au dessus, ou une règle

$$d(a(x_1, x_2), e(x_3, x_4)) \to a(d'(x_1, x_2), e(x_3, x_4)).$$

Dans ce cas, n_1 est étiqueté par b/b' ou b/d' dans t_1/t_3, et est annoté en utilisant les règles α_7 ou α_8.

– Noeud n_2. Il y a deux cas :

1. n_2 est étiqueté par b' dans t_3, c-à-d., b' ne bouge pas de ce nœud. Dans ce cas, n_2 est étiqueté par a/b' dans t_1/t_3, et est annoté par $q^a_{b'}$ en utilisant les règles α_4 ou α_5.
2. b' descend (elle traverse un des fils de n_2,...) en appliquant (par exemple) une règle

$$b'(f(x_1, x_2), g(x_3, x_4)) \to f'(b'(x_1, x_2), g(x_3, x_4)).$$

Dans ce cas, n_2 est étiqueté par a/f' dans t_1/t_3, et est annoté par $q^a_{b'}$ en utilisant les règles α_9 ou α_{10}.

– Le nœud n_3 est annoté par q_c.

– Dans t_1, n_1 est étiqueté par a, n_2 par b, n_3 par c, et une règle

$$a(b(x_1, x_2), c(x_3, x_4)) \to b'(a(x_1, x_2), c(x_3, x_4))$$

s'applique à ces trois nœuds pour donner t_2. Nous aurons alors que dans t_2, n_1 est étiqueté par b', n_2 par a, et n_3 par c. Ensuite, d'autres réécritures s'appliquent à t_2 pour obtenir t_3. Nous montrons comment l'automate annote ces nœuds pour le terme t_1/t_3 :

– Noeud n_1. Il y a deux cas :

1. n_1 est étiqueté par b' dans t_3, c-à-d., b' ne bouge pas de ce nœud. Dans ce cas, n_1 est étiqueté par a/b' dans t_1/t_3, et est annoté en utilisant les règles α_{13} (nous allons voir que n_2 est annoté par q^b_a, et n_3 par q_c).
2. b' continue à monter (elle traverse la racine de n_1,...) en appliquant une règle

$$h(b'(x_1, x_2), k(x_3, x_4)) \to b'(h'(x_1, x_2), k(x_3, x_4)).$$

Dans ce cas, n_1 est étiqueté par a/h' dans t_1/t_3, et est annoté par $q^{b'}_{h'}$ en utilisant les règles α_{17}.

– Noeud n_2. Il y a deux cas :

1. n_2 est étiqueté par a dans t_3, c-à-d., a ne bouge pas de ce nœud. Dans ce cas, n_2 est étiqueté par b/a dans t_1/t_3, et est annoté en utilisant les règles α_{11} ou α_{12}.

2. a descend (elle traverse un des fils de n_2,...) en appliquant (par exemple) une règle

$$a(d(x_1, x_2), e(x_3, x_4)) \to d'(a(x_1, x_2), e(x_3, x_4)).$$

Dans ce cas, n_2 est étiqueté par b/d' dans t_1/t_3, et est annoté par q_a^b en utilisant les règles α_{14}, α_{15}, ou α_{16}.

– Le nœud n_3 est annoté par q_c.

– Dans t_1, n_1 est étiqueté par a, n_2 par b, et n_3 par c, et une règle

$$a(b(x_1, x_2), c(x_3, x_4)) \to b'(a(x_1, x_2), a(x_3, x_4))$$

s'applique à ces trois nœuds pour donner t_2. Les nœuds de t_1/t_3 sont annotés comme précédemment en utilisant les règles (α_{18})–(α_{27}).

Nous montrons que :

Lemme 3.2.2 $\mathcal{R}_i^* = \mathcal{L}(\mathcal{A}_i)$.

Preuve : Nous montrons que pour chaque $b \in \mathcal{S}_{i-1}$, et chaque $a, b' \in \mathcal{S}_i$, nous avons :

(A) $L_q = \mathcal{R}_i^*$,

(B) $L_{q_a} = \{u \in \mathcal{R}_i^* \mid racine(u) \in \{a/a, a/b', b/a \mid b \in \mathcal{S}_{i-1} \cup \mathcal{S}_i, b' \in \mathcal{S}_i\}\}$,

(C) $L_{q^a_{b'}}$ est l'ensemble des termes $u \in T_{\mathcal{S}\times\mathcal{S}}$ t.q. il existe une règle

$$b(a(x_1, x_2), c(x_3, x_4)) \to a(b'(x_1, x_2), c(x_3, x_4)) \in \mathcal{R}_i^{\uparrow}$$

t.q. $b/a(u, c/c) \in \mathcal{R}_i^*$, ou $b/a(c/c, u) \in \mathcal{R}_i^*$.

(D) $L_{q_a^b}$ est l'ensemble des termes $u \in T_{\mathcal{S}\times\mathcal{S}}$ t.q. il existe un symbole d, et deux termes u_1 et u_2 dans $T_{\mathcal{S}\times\mathcal{S}}$ t.q. $u = b/d(u_1, u_2)$, et $a/d(u_1, u_2) \in \mathcal{R}_i^*$.

Montrons les inclusions $\subseteq$ d'abord. Soit donc u tel que $u \xrightarrow{k}_\delta q$, $u \xrightarrow{k}_\delta q_a$, $u \xrightarrow{k}_\delta q^a_{b'}$, ou $u \xrightarrow{k}_\delta q_a^b$, respectivement. Nous procédons par induction sur k.

– Pour $k = 1$:

– si $u \to_\delta q$, alors u est une feuille étiquetée par b/b pour un certain b dans $\mathcal{S}$ (règles α_1), c-à-d., $u \in \mathcal{R}_i^*$,

– si $u \to_\delta q_a$, alors u est une feuille étiquetée par a/a (règles α_2), c-à-d., $u \in \mathcal{R}_i^*$, et $racine(u) = a/a$,

– si $u \to_\delta q^a_{b'}$, alors u est une feuille étiquetée par a/b' et $b(a(x_1, x_2), c(x_3, x_4)) \to a(b'(x_1, x_2), c(x_3, x_4))$ est une règle de $\mathcal{R}_i^{\uparrow}$ (règles α_5). Alors $b/a(u, c/c) \in \mathcal{R}_i^*$, et $b/a(c/c, u) \in \mathcal{R}_i^*$ puisque $b(c(x_1, x_2), a(x_3, x_4)) \to a(c(x_1, x_2), b'(x_3, x_4))$ est aussi une règle de $\mathcal{R}_i^{\uparrow}$.

– si $u \to_\delta q_a^b$, alors u est une feuille étiquetée par b/a et , nous avons que a/a est dans $\mathcal{R}_i^*$.

– Soit $k > 1$, et soit u tels que :

– $u \xrightarrow{k}_\delta q$, alors nécessairement, il existe $a \in \mathcal{S}_i$ tel que :

1. $u \xrightarrow{k-1}_\delta q_a \to_\delta q$. Par induction, nous avons que $u \in L_{q_a} \subseteq \mathcal{R}_i^*$, ou

2. $u = a/a(u_1, u_2) \xrightarrow{k-1}_\delta a/a(q, q) \to_\delta q$ (règles α_1). Par induction nous avons que $u_1, u_2 \in \mathcal{R}_i^*$, et donc $u = a/a(u_1, u_2) \in \mathcal{R}_i^*$.

– $u \xrightarrow{k}_\delta q_a$. Il y a plusieurs cas selon la dernière règle de transition qui a été appliquée :

1. $u = a/a(u_1, u_2) \xrightarrow{k-1}_\delta a/a(q, q) \to_\delta q_a$ (règles α_2). Comme précédemment, nous pouvons déduire par induction que $u \in \mathcal{R}_i^*$. En plus, $racine(u) = a/a$.

2. Il existe une règle $r = b(a(x_1, x_2), c(x_3, x_4)) \to a(b'(x_1, x_2), c(x_3, x_4))$ dans $\mathcal{R}_i^\uparrow$, et $u = b/a(u_1, u_2) \xrightarrow{k-1}_\delta b/a(q^a_{b'}, q_c) \to_\delta q_a$ (règles α_6). Par induction, puisque $u_1 \in L_{q^a_{b'}}$ et $u_2 \in L_{q_c}$, nous avons que $b/a(u_1, c/c) \in \mathcal{R}_i^*$, $u_2 \in \mathcal{R}_i^*$, et la racine n de u_2 est étiquetée par c/c, c/d', ou d/c, pour $d \in \mathcal{S}_{i-1} \cup \mathcal{S}_i$, et $d' \in \mathcal{S}_i$. Supposons que n est étiquetée par d/c avec $d \in \mathcal{S}_{i-1}$ (les autres cas se traitent de la même manière). Soient donc $t_1, t_2, t_3, t'_1, t'_2$, et t'_3 tels que $u_1 = t_1/t'_1$ (donc $a(t'_1, c) \in \mathcal{R}_i^*\big(b(t_1, c)\big)$), et $u_2 = d/c(t_2/t'_2, t_3/t'_3)$ (nous avons $c(t'_2, t'_3) \in \mathcal{R}_i^*\big(d(t_2, t_3)\big)$). Alors, $u = b/a(u_1, u_2) = b/a\big(t_1/t'_1, d/c(t_2/t'_2, t_3/t'_3)\big) \in \mathcal{R}_i^*$ puisque $a\big(t'_1, c(t'_2, t'_3)\big) \in \mathcal{R}_i^*\Big(b\big(t_1, c(t'_2, t'_3)\big)\Big)$, et $b\big(t_1, c(t'_2, t'_3)\big) \in \mathcal{R}_i^*\Big(b\big(t_1, d(t_2, t_3)\big)\Big)$.

3. Il existe une règle $r = a(b(x_1, x_2), c(x_3, x_4)) \to b'(a(x_1, x_2), c(x_3, x_4))$ dans $\mathcal{R}_i^\downarrow$, et $u = a/b'(u_1, u_2) \xrightarrow{k-1}_\delta a/b'(q^b_a, q_c) \to_\delta q_a$ (règles α_{13}). Par induction, puisque $u_1 \in L_{q^b_a}$, il existe un symbole e, et des termes t_1, t_2, t'_1, et t'_2 tels que $u_1 = b/e(t_1/t'_1, t_2/t'_2)$ et $a/e(t_1/t'_1, t_2/t'_2) \in \mathcal{R}_i^*$. D'un autre côté, puisque $u_2 \in L_{q_c}$, nous avons que $u_2 \in \mathcal{R}_i^*$, et la racine n de u_2 est étiquetée par c/c, c/d', ou d/c, pour $d \in \mathcal{S}_{i-1} \cup \mathcal{S}_i$, et $d' \in \mathcal{S}_i$. Supposons cette fois que n est étiquetée par c/c (les autres cas se traitent de la même manière). Soient donc t_3, t_4, t'_3, et t'_4 tels que $u_2 = c/c(t_3/t'_3, t_4/t'_4)$. Il s'en suit que $u = a/b'(u_1, u_2) \in \mathcal{R}_i^*$ puisque

$$b'\big(e(t'_1, t'_2), c(t'_3, t'_4)\big) \in \mathcal{R}_i^*\Big(b'\big(a(t_1, t_2), c(t_3, t_4)\big)\Big)$$

et

$$b'\big(a(t_1, t_2), c(t_3, t_4)\big) \in r\Big(a\big(b(t_1, t_2), c(t_3, t_4)\big)\Big)$$

4. Le cas où $u = d/b'(u_1, u_2) \xrightarrow{k-1}_\delta d/b'(q^b_d, q_c) \to_\delta q_{b'}$ (règles α_{13}), où $b' = a$ se traite de manière similaire.

5. Les cas où une des règles α_{22} est appliquée en dernier se traite de la même façon.

– Les cas où $u \xrightarrow{k}_\delta q^a_{b'}$ et $u \xrightarrow{k}_\delta q^b_a$ se traitent en suivant le même schéma.

Pour l'autre direction, nous montrons par induction sur k que si $u \in \mathcal{R}_i^k$ et $racine(u) \in \{a/a, a/b', b/a \mid b \in \mathcal{S}_{i-1}, b' \in \mathcal{S}_i\}$, alors $u \in L_{q_a}$, ce qui implique, grâce aux règles α_3, que $u \in L_q$.

– si $k = 0$, alors u est un arbre étiqueté par des a/a pour des lettres $a \in \mathcal{S}$. Les règles α_1 et α_2 impliquent que $u \xrightarrow{*}_\delta q_a$.

– si $k > 0$. Soit alors $u' \in \mathcal{R}_i^{k-1}$, t_1, t_2, et t_3 tels que $u' = t_1/t_2$, $u = t_1/t_3$, et $t_3 \in \mathcal{R}_i(t_2)$. Soit alors r une règle de $\mathcal{R}_i$ telle que $t_3 \in r(t_2)$. Il y a trois cas selon la nature de la règle r :

1. $r = b(a(x_1, x_2), c(x_3, x_4)) \to a(b'(x_1, x_2), c(x_3, x_4))$. Soient alors deux contextes C_1 et C_2, trois symboles g_1, g_2, et g_3, et huit termes $v_1, v_2, v_3, v_4, v'_1, v'_2, v'_3$, et v'_4 tels que

$$t_1 = C_1\Big(g_1\big(g_2(v_1, v_2), g_3(v_3, v_4)\big)\Big),$$

$$t_2 = C_2\Big(b\big(a(v'_1, v'_2), c(v'_3, v'_4)\big)\Big),$$

et

$$t_3 = C_2\Big(a\big(b'(v'_1, v'_2), c(v'_3, v'_4)\big)\Big).$$

Nous en déduisons que $g_1 = b$ puisque $b \in \mathcal{S}_{i-1}$, et donc à cet endroit aucune réécriture n'a été faite par les règles de $\mathcal{R}_i$. Il en découle que $a(v'_1, v'_2) \in \mathcal{R}_i^{k-1}\big(g_2(v_1, v_2)\big)$ et que $c(v'_3, v'_4) \in \mathcal{R}_i^{k-1}\big(g_3(v_3, v_4)\big)$. Donc, $g_2/a(v_1/v'_1, v_2/v'_2) \in \mathcal{R}_i^{k-1}$, et $g_3/c(v_3/v'_3, v_4/v'_4) \in \mathcal{R}_i^{k-1}$. Donc, par induction, nous avons que

$$g_2/a(v_1/v'_1, v_2/v'_2) \to_\delta^* q_a \to_\delta q$$

et que

$$g_3/c(v_3/v'_3, v_4/v'_4) \to_\delta^* q_c \to_\delta q.$$

Il s'en suit par les règles α_2 que

$$C_1/C_2\Big[b/b\big(g_2/a(v_1/v'_1, v_2/v'_2), g_3/c(v_3/v'_3, v_4/v'_4)\big)\Big] \to_\delta^* C_1/C_2[b/b(q, q)]$$

$$\to_\delta C_1/C_2[q_b] \to_\delta^* q_h.$$

Nous notons cette dérivation par $(\star)$. Par exemple, si la racine de t_1 ou celle de t_2 est égale à h. Nous voulons montrer que

$$u = C_1/C_2\Big[b/a\big(g_2/b'(v_1/v'_1, v_2/v'_2), g_3/c(v_3/v'_3, v_4/v'_4)\big)\Big] \to_\delta^* q_h.$$

pour ce faire, il suffit de montrer que

$$b/a\big(g_2/b'(v_1/v'_1, v_2/v'_2), g_3/c(v_3/v'_3, v_4/v'_4)\big) \to_\delta^*$$

$$b/a\big(g_2/b'(v_1/v'_1, v_2/v'_2), q_c\big) \to_\delta^* q_b.$$

Donc, en appliquant les règles α_6, il suffit de montrer que

$$g_2/b'(v_1/v'_1, v_2/v'_2) \to_\delta^* q^a_{b'}.$$

Ceci dépend de la nature de la lettre g_2 :

– $g_2 = a$, c'est à dire il n'y a pas eu de réécriture à cette place entre t_1 et t_2. Donc v_1/v'_1 et v_2/v'_2 sont dans $\mathcal{R}_i^{k-1}$. Il s'en suit par induction qu'ils peuvent être annotés par q, et donc, en appliquant les règles α_4 nous obtenons que :

$$(a/b'(v_1/v'_1, v_2/v'_2) \to_\delta^* a/b'(q,q) \to_\delta q^a_{b'}$$

– $g_2 = d$. Il y a trois cas selon la dernière règle qui a été utilisée pour annoter $d/a(v_1/v'_1, v_2/v'_2)$ par q_a lors de la dérivation $(\star)$:

(a) Soit c'est une règle α_6 qui a été appliquée. Dans ce cas, il existe forcément une règle

$$r' = d(a(x_1,x_2), e(x_3,x_4)) \to a(d'(x_1,x_2), e(x_3,x_4))$$

dans $\mathcal{R}_i^{\uparrow}$ telle que $v_1/v'_1 \to_\delta^* q^a_{d'}$ et $v_2/v'_2 \to_\delta^* q_e$. Les règles α_8 permettent que l'on ait : $d/b'(v_1/v'_1, v_2/v'_2) \to_\delta^* d/b'(q^a_{d'}, q_e) \to_\delta q^a_{b'}$

(b) Soit c'est une règle α_{13} qui a été appliquée. Dans ce cas, il existe forcément une règle

$$r'' = d(f(x_1,x_2), e(x_3,x_4)) \to f'(d(x_1,x_2), e(x_3,x_4))$$

dans $\mathcal{R}_i^{\downarrow}$ telle que $f' = a$ et $v_1/v'_1 \to_\delta^* q^f_d$ et $v_2/v'_2 \to_\delta^* q_e$. Les règles α_{17} permettent que l'on ait : $d/b'(v_1/v'_1, v_2/v'_2) \to_\delta^* d/b'(q^f_d, q_e) \to_\delta q^a_{b'}$.

(c) Le cas où c'est une des règles α_{22} qui a été appliquée est similaire au cas précédent.

2. $r = a(b(x_1,x_2), c(x_3,x_4)) \to b'(a(x_1,x_2), c(x_3,x_4))$. Soient alors deux contextes C_1 et C_2, trois symboles g_1, g_2, et g_3, et huit termes $v_1, v_2, v_3, v_4, v'_1, v'_2, v'_3$, et v'_4 tels que

$$t_1 = C_1\Big(g_1\big(g_2(v_1,v_2), g_3(v_3,v_4)\big)\Big),$$

$$t_2 = C_2\Big(a\big(b(v'_1,v'_2), c(v'_3,v'_4)\big)\Big),$$

et

$$t_3 = C_2\Big(b'\big(a(v'_1,v'_2), c(v'_3,v'_4)\big)\Big).$$

Nous en déduisons que $g_2 = b$ puisque $b \in \mathcal{S}_{i-1}$, et donc à cet endroit aucune réécriture n'a été faite par les règles de $\mathcal{R}_i$. Il en découle que $b/b(v_1/v'_1, v_2/v'_2) \to_\delta^* q$ au cours de l'annotation de u', puisque $b \in \mathcal{S}_{i-1}$, et donc la seule règle possible pour annoter un nœud étiqueté par b/b est la règle $\alpha_1 : b/b(q,q) \to q$. Et donc, pour la même raison,

$$g_1/a\big(b/b(v_1/v'_1, v_2/v'_2), g_3/c(v_3/v'_3, v_4/v'_4)\big) \to_\delta^* g_1/a(q,q') \to_\delta q'',$$

où $q' = q$ vu la forme des règles, et q'' est un état de l'automate. Nous en déduisons que $g_3/c(v_3/v'_3, v_4/v'_4) \in \mathcal{R}_i^*$, et plus précisément, nous avons

que $g_3/c(v_3/v'_3, v_4/v'_4) \in \mathcal{R}_i^{k-1}$, et donc par induction, nous déduisons que $g_3/c(v_3/v'_3, v_4/v'_4) \to_\delta^* q_c$.
Reprenons l'annotation de u'. Nous avons

$$u' \to_\delta^* C_1/C_2[q''] \to_\delta^* q_h$$

pour un certain symbole h. Comme l'étiquette de la racine de u est la même que celle de u', il faut que nous montrons que $u \to_\delta^* q_h$.
En appliquant les règles α_{12}, nous obtenons :

$$g_1/b'\big(b/a(v_1/v'_1, v_2/v'_2), g_3/c(v_3/v'_3, v_4/v'_4)\big) \to_\delta^* g_1/b'(q_a^b, q_c).$$

Nous allons montrer que $g_1/b'(q_a^b, q_c) \to_\delta q''$, auquel cas, nous aurons

$$u \to_\delta^* C_1/C_2[q''] \to_\delta^* q_h.$$

Il y a trois cas selon la nature de g_1 :

(a) $g_1 = a$. Dans ce cas $q'' = q_a$ ($a/a(q, q) \to_\delta q_a$), et par α_{13}, nous obtenons $a/b'(q_a^b, q_c) \to_\delta q_a$.

(b) $g_1 = b$. Alors $q'' = q_a^b$ ($b/a(q, q) \to_\delta q_a^b$), et par α_{14}, nous obtenons $b/b'(q_a^b, q_c) \to_\delta q_a^b$.

(c) $g_1 = d$. Alors $q'' = q_a^d$ ($d/a(q, q) \to_\delta q_a^d$), et par α_{15}, nous obtenons $d/b'(q_a^b, q_c) \to_\delta q_a^d$.

3. Le cas où $r = a(b(x_1, x_2), c(x_3, x_4)) \to b'(a(x_1, x_2), a(x_3, x_4))$ est similaire au cas précédent.

□

3.2.3.2 Construction de $\mathcal{R}^*_{i\to i+1}$

Nous construisons un réétiquetage qui reconnaît $\mathcal{R}^*_{i\to i+1}$. Soit le réétiquetage

$$\mathcal{A}_{i\to i+1} = (Q, \mathcal{S}, \mathcal{S}, F, \delta)$$

où $Q = \{q\} \cup \{q_a \mid a \in \mathcal{S}_i\} \cup \{q_{b/d} \mid b \in \mathcal{S}_{i-1}, d \in \mathcal{S}_{i+1}\}$, $F = \{q\}$, et δ est l'ensemble de règles suivantes :

(γ_1) $a/a \to q$ et $a/a(q, q) \to q$, pour chaque $a \in \mathcal{S}$,

(γ_2) $a/a \to q_a$ et $a/a(q, q) \to q_a$, pour chaque $a \in \mathcal{S}_i$,

(γ_3) $q_a \to q$, pour chaque $a \in \mathcal{S}_i$,

(γ_4) $b/d \to q$ et $b/d(q, q) \to q$, pour chaque règle $b(x_1, x_2) \to d(x_1, x_2)$ de $\mathcal{R}_{i\to i+1}$,

(γ_5) $b/d(q_a, q_c) \to q$, pour chaque règle de $\mathcal{R}_{i\to i+1}$,

$$b(a(x_1, x_2), c(x_3, x_4)) \to d(a(x_1, x_2), c(x_3, x_4))$$

(γ_6) $b/d(q, q) \to q_{b/d}$, $b/d \to q_{b/d}$,

(γ_7) $a/a(q_{b/d}, q_c) \to q_a$ pour chaque règle de $\mathcal{R}_{i\to i+1}$,

$$a(b(x_1, x_2), c(x_3, x_4)) \to a(d(x_1, x_2), c(x_3, x_4))$$

(γ_8) $a/a(q_{b/d}, q_{c/d}) \to q_a$ pour chaque règle de $\mathcal{R}_{i\to i+1}$

$$a(b(x_1, x_2), c(x_3, x_4)) \to a(d(x_1, x_2), d(x_3, x_4)).$$

Ici aussi, l'automate ci-dessus a pour chaque règle de la forme $f/g(q_1, q_2) \to q$ une règle symétrique $f/g(q_2, q_1) \to q$ qui n'est pas représentée.

Intuitivement, un nœud est annoté par q_a s'il est étiqueté par a/a. Comme précédemment, ces états q_a sont utiles pour savoir si la contrainte d'une règle est validée à un certain point. Puisque ces contraintes appartiennent juste à $\mathcal{S}_i$, il suffit de mémoriser les places des symboles $a \in \mathcal{S}_i$. Un nœud est annoté par $q_{b/d}$ s'il est étiqueté par b/d après l'application d'une règle de la forme (3.22) ou (3.23). Un nœud annoté par cet état doit avoir un père étiqueté avec a/a (les règles γ_6 et γ_7).

Nous montrons que :

Lemme 3.2.3 $\mathcal{R}^*_{i\to i+1} = \mathcal{L}(\mathcal{A}_{i\to i+1})$.

Preuve :

Nous montrons (comme précédemment) que :
- $L_q = \mathcal{R}^*_{i\to i+1}$,
- $L_{q_a} = \{u \in \mathcal{R}^*_{i\to i+1} \mid racine(u) = a/a\}$,
- $L_{q_{b/d}} = \{u = b/d(u_1, u_2) \mid u_1 \in \mathcal{R}_{i\to i+1}, u_2 \in \mathcal{R}_{i\to i+1}\}$.

□

Nous obtenons alors le résultat principal de cette section :

Théorème 3.2.1 *Soit $\mathcal{R}$ un well-oriented system, alors $\mathcal{R}^*$ est une relation de réétiquetage régulière et peut être effectivement représentée par un réétiquetage.*

Preuve : Se déduit directement des lemmes 3.2.1, 3.2.2, et 3.2.3. □

3.3 Conclusion

Dans ce chapitre, nous avons contribué à étendre l'applicabilité du *regular model checking* à la vérification des systèmes paramétrés linéaires et arborescents dans deux sens. D'abord, nous nous sommes intéressés aux systèmes paramétrés linéaires. Nous avons considéré la classe des relations de semi-commutations, qui est une classe significative qui apparaît dans la modélisation des systèmes où les communications se font entre voisins (comme c'est le cas par exemple pour le *token passing protocol* donné à l'introdution de la thèse). Comme ces relations ne préservent pas la régularité, nous avons défini une sous classe d'expressions régulières : la classe APC qui est intéressante pour la modélisation des systèmes paramétrés (nous référons à tous les exemples de

protocoles paramétrés définis sur des structures linéaires donnés dans ce document). Nous avons montré que cette classe est effectivement fermée par semi-commutations, ce qui permet l'analyse des systèmes paramétrés linéaires modélisés par ces relations. Nous avons également montré comment ce résultat peut être utilisé pour la vérification des systèmes paramétrés ayant une topologie en anneau où l'échange d'informations se fait entre les voisins.

Selon notre connaissance, c'est la première fois qu'il est montré qu'une sous classe non-triviale de langages réguliers satisfait cette propriété. Comme mentionné précédemment, APC correspond au niveau 3/2 dans la hiérarchie de Straubing-Thérien, et au niveau Σ_2 dans la hiérarchie des alternances des quantificateurs de la logique de premier ordre. Nous avons montré que APC correspond au plus haut niveau dans les deux hiérarchies qui est fermé par semi-commutations.

Dans la deuxième partie de ce chapitre, nous nous sommes intéressés aux systèmes paramétrés arborescents. Nous avons mis en évidence une classe de relations de réétiquetages (la classe des Well-Oriented Systems ou WOS) qui est intéressante pour la modélisation des systèmes où la communication a lieu entre les processus père et les processus fils. Nous avons montré que la clôture réflexive-transitive d'une relation de cette classe est une relation de réétiquetage qui peut être effectivement construite. Ceci permet l'analyse des systèmes paramétrés qui peuvent être modélisés par ce genre de relations. C'est le cas par exemple des protocoles décrits dans la section 3.2.1.

Deuxième partie

Vérification des programmes récursifs parallèles par analyse d'accessibilité des PRS

Introduction

Nous nous intéressons dans cette partie et la partie suivante à l'analyse des programmes avec récursion, création dynamique de processus parallèles, et synchronisation. Dans cette partie, nous développons notre approche basée sur le modèle PRS. Ce formalisme a été introduit dans [May98]. Il est plus général que les systèmes à pile, les systèmes PA, et les réseaux de Petri. Nous proposons donc de le considérer pour modéliser les programmes récursifs parallèles. En effet, les automates à pile ont été proposés comme un modèle naturel pour les programmes récursifs [EK99, ES01]. Les systèmes PA sont utilisés pour abstraire les programmes vers un modèle qui ne tient pas compte de la synchronisation et du retour de résultats aux procédures appelantes [EK99, EP00]. De même, les réseaux de Petri ont été utilisés pour modéliser les programmes concurrents avec création dynamique de processus mais sans récursivité (dans ce cas, chaque processus est un système fini, mais le nombre des processus à un instant donné peut être arbitrairement grand) [BCR01, DBR02]. Les techniques d'analyse d'accessibilité symbolique de ces modèles sont donc utilisées pour vérifier ces programmes. Comme le modèle PRS est plus général que ces trois formalismes, nous définissons des techniques d'analyse d'accessibilité symboliques pour ce modèle qui permettent d'étendre et d'unifier les approches existantes pour les systèmes à pile, les réseaux de Petri, et les systèmes PA. Nous montrons comment utiliser ces techniques dans l'analyse d'accessibilté des programmes concurrents parallèles avec appels récursifs de procédures, création dynamique de processus, et synchronisation.

Dans le chapitre suivant, nous définissons ce formalisme et nous donnons une traduction directe automatisable des programmes vers des PRS. Le PRS obtenu a en général plus de comportements que le programme concret, mais nous identifions une classe intéressante de programmes appelés *programmes bien reliés* pour lesquels notre traduction est exacte. Ces programmes sont plus généraux que la classe des programmes récursifs parallèles qui ne contiennent pas de synchronisation. Nous donnons dans les chapitres 5 et 6 des algorithmes qui permettent l'analyse d'accessibilité symbolique de ce modèle.

Chapitre 4

Modélisation des programmes par des PRS

En général, les programmes récursifs parallèles sont modélisés par des *parallel flow graphs* (PFG). Nous décrivons ce modèle dans la première section de ce chapitre. Ensuite, nous définissons le modèle des Process Rewrite Systems (PRS) introduit par Mayr [May98]. Nous donnons après une transformation automatisable qui permet de passer d'un PFG à un PRS. Le modèle obtenu a en général plus de comportements que le programme originel. Ceci est dû au fait que PRS ne permet pas de modéliser la synchronisation de manière exacte dans tous les contextes. Cette modélisation permet d'avoir une analyse "safe", dans le sens que si le PRS satisfait une propriété de sûreté, alors nous pouvons être sûrs que le vrai programme aussi la satisfait.

Pour avoir des résultats précis, nous définissons une classe de programmes que nous appelons *programmes bien reliés* pour lesquels notre transformation est exacte. Cette classe est assez générale puisqu'elle comprend les programmes récursifs ainsi que les programmes récursifs parallèles sans synchronisation.

Egalement, nous mettons en évidence une autre classe importante de programmes qui peut être analysée sans considérer toutes les équivalences structurelles de PRS. Comme nous allons le voir au chapitre suivant, ceci nous permet d'analyser de manière exacte cette classe de programmes.

Nous proposons enfin une traduction qui permet d'abstraire un programme quelconque vers un PAD (une sous-classe de PRS qui est plus générale que les systèmes à pile et les systèmes PA). Cette abstraction est utile car comme nous allons le voir aux chapitres suivants, il est toujours possible d'analyser exactement les PAD en tenant compte de toutes les équivalences, alors que ce n'est pas toujours le cas pour les systèmes PRS quelconques.

4.1 Parallel flow graphs

Dans le cadre que nous considérons, un programme est composé d'un nombre fini de procédures. Ces procédures peuvent interagir en s'appelant récursivement. Plusieurs

procédures peuvent être appelées en parallèle. L'exécution de chaque procédure dépend des valeurs des arguments passées par la procédure appelante. Quand les procédures appelées terminent, elles retournent leurs résultats à la procédure appelante qui peut alors poursuivre son exécution jusqu'ici suspendue. L'exécution commence par une procédure spéciale `main` qui ne peut pas être appelée. Les processus parallèles peuvent se synchroniser par rendez-vous à travers des émissions/réceptions de signaux. Un processus exécute une action $a!$ s'il émet le signal a à tous les processus qui lui sont en parallèle. $a?$ exprime que le processus attend de recevoir le signal a pour exécuter une certaine action.

4.1.1 Définition des PFG

Comme dans [Mo02, SS00], nous supposons que de tels programmes sont donnés sous forme de *parallel flow graphs (PFG)*. Nous supposons que les données de type infini sont abstraites en des données de type fini en utilisant les techniques standard d'interprétation abstraite [CC77]. Le système abstrait ainsi obtenu peut avoir plus d'exécutions que le programme originel. Par conséquent, si notre analyse réussit à montrer que ce système abstrait est correct par rapport à une propriété de sûreté, nous pouvons déduire que le vrai programme est aussi correct.

Soit alors un ensemble fini de signaux (Sig), et un ensemble fini d'instructions $Stmt$ (pour statement). Une instruction est :

- soit une instruction de base, c-à-d. une instruction vide **skip**, ou une affectation de la forme "$v := exp$" (où v est une variable et exp est une expression) ou une conditionnelle "if-then-else" ;
- soit une émission "$a!$" d'un signal a de Sig ou une attente de réception "$a?$" d'un signal a ;
- soit un appel à une procédure "$x := call(p)$", où x est une variable. Ceci exprime que la valeur retournée par p est stockée dans x ;
- soit un appel parallèle à n procédures "$(x_1, \ldots, x_n) := call(p_1 || \cdots || p_n)$", où les x_i sont des variables. Comme précédemment, ceci exprime que les valeurs retournées par les procédures p_i sont stockées dans les variables x_i.

Un *parallel flow graph* est un ensemble fini de procédures `Proc` dont une procédure spéciale `main`. A chaque procédure p est associé un *control flow graph (CFG)* qui est un graphe dont les nœuds correspondent aux différents points de contrôle de la procédure, et dont les arêtes (qui expriment le flot de contrôle) peuvent être étiquetées par les instructions de la procédure : Une arête $n_1 \xrightarrow{inst} n_2$ exprime qu'en exécutant l'instruction $inst$, la procédure passe du point de contrôle n_1 au point de contrôle n_2. Formellement, un *control flow graph* est un quadruplet $G_p = (N_p, E_p, e_p, r_p)$ tels que :

- N_p est l'ensemble des points de contrôle de la procédure ;
- $E_p \subseteq N_p \times Stmt \times N_p$ est l'ensemble des arêtes ;
- e_p est le point d'entrée de la procédure ($e_p \in N_p$) ;
- r_p est le point de retour de la procédure ($r_p \in N_p$).

Pour simplifier la présentation, nous omettons les annotations **skip** des arêtes correspondant aux instructions vides. Nous supposons également que les ensembles des points de contrôles des différentes procédures sont disjoints, c-à-d., si p et q sont deux

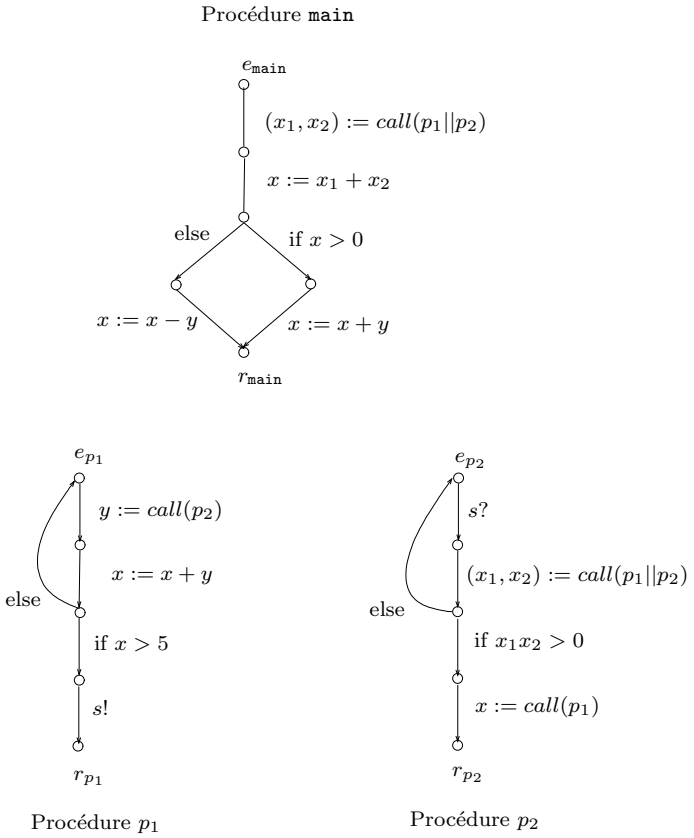


FIG. 4.1 – Un exemple de PFG

procédures différentes, alors $N_p \cap N_q = \emptyset$. Nous posons $N = \bigcup_{p \in \texttt{Proc}} N_p$ l'ensemble des points de contrôle de tout le programme et $E = \bigcup_{p \in \texttt{Proc}} E_p$ l'ensemble de toutes les instructions entre ces différents points de contrôle.

4.1.2 Exemple

La figure 4.1 montre un exemple de PFG. Il y a une procédure `main` et deux procédures p_1 et p_2. Ces procédures se synchronisent à travers le signal s. Par exemple, l'instruction $(x_1, x_2) := call(p_1||p_2)$ exprime que les procédures p_1 et p_2 sont appelées en parallèle, et que lorsqu'elles terminent, le résultat de p_1 est stocké dans x_1 et celui de p_2 dans x_2.

4.1.3 Sémantique

Nous définissons la sémantique des PFG en termes d'arbres d'exécutions. Nous nous basons sur la sémantique présentée dans [Mo02] qui décrit les comportements des PFG sans synchronisation et nous l'étendons à notre cas où la synchronisation est permise. Pour des raisons de simplicité, nous ne tenons compte que des effets des différentes instructions sur les variables locales. Nous ignorons les variables globales. Donc, les configurations décrites par la sémantique ne représentent que le contrôle et les valeurs des variables locales des procédures. Nous supposons pour simplifier que toutes les procédures ont le même ensemble de variables locales.

Dans un programme séquentiel, une configuration est représentée par une séquence de paires (n, loc) où n est un nœud de N, et loc un vecteur correspondant aux valeurs des différentes variables locales atteintes par le programme au nœud n. Cette séquence modélise une pile d'adresses de retour. Dans les PFG, les procédures peuvent aussi être appelées en parallèle. Les configurations seront alors des arbres. Chaque nœud de l'arbre est étiqueté par une paire (n, loc). Le degré d'un nœud interne de l'arbre est soit 1 soit k. Les nœuds de degré 1 correspondent aux adresses de retour pour des appels séquentiels d'une procédure. Les nœuds de degré k correspondent aux adresses de retour des appels parallèles $p_1 || \cdots || p_k$. Les points de contrôle actifs sont situés aux feuilles de l'arbre. Donc les transitions qui permettent de passer d'une configuration à une autre s'appliquent aux feuilles. La configuration initiale du programme est simplement un nœud $(e_{\texttt{main}}, loc)$ correspondant au point d'entrée de la procédure initiale `main` du programme et aux valeurs des arguments avec lesquels elle a été appelée. Pour des raisons de simplicité, nous écrivons un arbre dont la racine est étiquetée par v et a dans l'ordre $A_1, \ldots, A_k$ comme sous arbres sous la forme $v(A_1, \ldots, A_k)$. Les configurations évoluent au cours de l'exécution de la manière suivante :

Instruction de base : Nous passons d'une configuration c à une configuration c' en exécutant une instruction de base $n_1 \xrightarrow{inst} n_2$ du PFG s'il existe dans c une feuille où n_1 est actif et que c' est obtenu en substituant dans c cette occurrence active (n_1, loc_1) par (n_2, loc_2), où loc_2 est le résultat de l'application de l'instruction $inst$ aux valeurs loc_1. Plus précisément, si $inst$ est une instruction vide, alors $loc_2 = loc_1$. Si $inst$ est une affectation, loc_2 est le résultat de l'application de cette affectation à loc_1. Si $inst$ est une conditionnelle "if-then-else", alors $loc_2 = loc_1$ si loc_1 satisfait la condition, sinon la transition n'est pas prise.

Appel d'une procédure : Nous passons d'une configuration c à une configuration c' en appelant une procédure p : $n_1 \xrightarrow{x:=call(p)} n_2$ s'il existe dans c une feuille où n_1 est actif et que c' est obtenue en substituant cette feuille (n_1, loc) par la séquence $(n_2, loc)\big((e_p, loc_p)\big)$, où loc_p correspond aux valeurs des arguments passés à la procédure p. Ceci exprime que e_p devient actif avec les valeurs loc_p et que quand l'exécution de p termine, c'est n_2 qui devient actif.

Lorsque p termine, elle doit rendre son résultat à la procédure qui l'a appelée. Donc une configuration de la forme $(n_2, loc)\big((r_p, loc'_p)\big)$ (où r_p est le point de retour de la procédure p) se transforme en (n_2, loc'), où loc' représente les nouvelles valeurs des variables au point n_2. Ces nouvelles valeurs sont telles que si une variable est différente

de x, sa valeur dans loc' est la même que dans loc. Sinon, x prend la valeur retournée par p. Cette valeur est déterminée à partir de loc'_p.

Appel parallèle : Nous passons d'une configuration c à une configuration c' en appelant k procédures $p_1, \ldots, p_k$: $n_1 \xrightarrow{(x_1,\ldots,x_k):=call(p_1||\cdots||p_k)} n_2$ s'il existe dans c une feuille où n_1 est actif et que c' est obtenue en substituant cette feuille (n_1, loc) par le sous arbre $(n_2, loc)\big((e_{p_1}, loc_1), \ldots, (e_{p_k}, loc_k)\big)$ ayant (n_2, loc) comme racine et $(e_{p_1}, loc_1), \ldots, (e_{p_k}, loc_k)$ comme feuilles ; où loc_i correspond aux valeurs des arguments passés à la procédure p_i. Ceci exprime que les e_{p_i} deviennent actifs avec les valeurs loc_i des variables.

Dans ce cas, quand les p_i terminent, elles retournent leurs résultats à n_2. Donc, comme précédemment, une configuration de la forme $(n_2, loc)\big((r_{p_1}, loc'_1), \ldots, (r_{p_k}, loc'_k)\big)$ (les r_{p_i} étant les points de retour des procédures p_i) se transforme en (n_2, loc'), où loc' représente les nouvelles valeurs des variables au point n_2. Ces nouvelles valeurs sont telles que les x_i prennent les valeurs retournées par les p_i (à partir des loc'_i), et les autres variables gardent les mêmes valeurs qu'elles avaient dans loc.

Synchronisation : Nous passons d'une configuration c à une configuration c' en exécutant une action de synchronisation : $n_1 \xrightarrow{a!} n_2$ et $m_1 \xrightarrow{a?} m_2$ s'il existe dans c une feuille où n_1 est actif et une autre où m_1 est actif, et que c' est obtenue en substituant la feuille (n_1, loc) par (n_2, loc) et la feuille (m_1, loc') par (m_2, loc').

4.2 Process Rewrite Systems : Définition

Nous définissons dans cette section les Process Rewrite Systems que nous utiliserons pour modéliser les PFGs.

4.2.1 Syntaxe

Soit $Var = \{X, Y, \ldots\}$ un ensemble de variables de processus, et $\mathcal{T}$ un ensemble de termes de processus t défini par la syntaxe suivante, où X est une constante arbitraire de Var :

$$t ::= 0 \mid X \mid t \cdot t \mid t||t$$

Intuitivement, 0 est le processus nul et "." (resp. "||") dénote la composition séquentielle (resp. composition parallèle asynchrone).

Définition 4.2.1 (Classes de termes de processus) *Nous distinguons quatre classes de termes de processus :*

- **1** *Les variables de processus X,*
- **S** *Les termes qui sont soit le processus nul 0, soit une variable de processus X, soit une composition séquentielle de variables de processus. De tels termes sont appelés seq-termes. Puisque "." est associatif, un seq-terme peut être écrit sous la forme $X_1 \cdots X_n$.*
- **P** *Les termes qui sont soit le processus nul 0, soit une variable de processus X, soit une composition parallèle de variables de processus. De tels termes sont appelés*

paral-termes. Puisque "||" est associatif, un paral-terme peut être écrit sous la forme $X_1||\cdots||X_n$.

G *Les termes généraux avec des imbrications arbitraires des opérateurs "||" et "." tels que* $\big(X||(Y\cdot Z)\big)\cdot W$.

Il est facile de voir que **1** est une sous classe de **S** et **P**, qui sont des sous classes de **G** ; que **S** et **P** sont incomparables ; et que $\mathbf{S}\cap\mathbf{P}=\mathbf{1}\cup\{0\}$.

Définition 4.2.2 ([May98]) *Soient* $\alpha,\beta\in\{1,S,P,G\}$. *Un* (α,β)-Process Rewrite System *((α,β)-PRS) est un ensemble fini de règles de la forme* $l\to r$, *où* $l\in\alpha$ [1] *et* $r\in\beta$. *Un* (G,G)-*PRS est appelé PRS.*

4.2.2 Sémantique

Un PRS $\mathcal{R}$ induit une relation de transition $\to_{\mathcal{R}}$ sur $\mathcal{T}$ définie par les règles d'inférence suivantes :

$$\frac{t_1\to t_2\in\mathcal{R}}{t_1\to_{\mathcal{R}} t_2};\ \frac{t_1\to_{\mathcal{R}} t_1'}{t_1||t_2\to_{\mathcal{R}} t_1'||t_2};\ \frac{t_1\to_{\mathcal{R}} t_1'}{t_1\cdot t_2\to_{\mathcal{R}} t_1'\cdot t_2};\ \frac{t_2\to_{\mathcal{R}} t_2'}{t_1||t_2\to_{\mathcal{R}} t_1||t_2'};\ \frac{t_1\sim_0 0\ \ ,\ \ t_2\to_{\mathcal{R}} t_2'}{t_1\cdot t_2\to_{\mathcal{R}} t_1\cdot t_2'}$$

où $\sim_0$ est une équivalence entre les termes de processus qui identifie les processus nuls. Elle exprime la neutralité du processus nul "0" par rapport à "||", et "." :

$$\text{A1}: \quad t\cdot 0\sim_0 0\cdot t\sim_0 t||0\sim_0 0||t\sim_0 t$$

Nous considérons l'équivalence structurelle $\sim$ générée par les axiomes A1 et les axiomes suivants :

$$\begin{array}{lll}
\text{A2}: & (t\cdot t')\cdot t''\sim t\cdot(t'\cdot t'') & : \text{associativité de "."},\\
\text{A3}: & t||t'\sim t'||t & : \text{commutativité de "||"},\\
\text{A4}: & (t||t')||t''\sim t||(t'||t'') & : \text{associativité de '||"}.
\end{array}$$

Nous dénotons par $\sim_s$ l'équivalence induite par les axiomes A1 et A2, et par $\sim_{||}$ l'équivalence induite par les axiomes A1, A3, et A4.

Soit $\equiv$ une équivalence de l'ensemble $\{=,\sim_0,\sim_s,\sim_{||},\sim\}$, où $=$ correspond à l'identité entre les termes. Nous dénotons par $[t]_\equiv$ la classe d'équivalence modulo $\equiv$ du terme de processus t, c-à-d., $[t]_\equiv=\{t'\in\mathcal{T}\mid t\equiv t'\}$. Cette définition est étendue aux ensembles de termes de manière directe. $\equiv$ induit une relation de transition $\Rightarrow_{\equiv,\mathcal{R}}$ définie comme suit :

$$\forall t,t'\in\mathcal{T}, t\Rightarrow_{\equiv,\mathcal{R}} t' \text{ ssi } \exists u,u'\in\mathcal{T} \text{ t.q. } t\equiv u, u\to_{\mathcal{R}} u', \text{ et } u'\equiv t'$$

Soit $\overset{*}{\Rightarrow}_{\equiv,\mathcal{R}}$ la clôture réflexive-transitive de $\Rightarrow_{\equiv,\mathcal{R}}$. Soient $Post_{\mathcal{R},\equiv}(t)=\{t'\in\mathcal{T}\mid t\Rightarrow_{\equiv,\mathcal{R}} t'\}$, $Pre_{\mathcal{R},\equiv}(t)=\{t'\in\mathcal{T}\mid t'\Rightarrow_{\equiv,\mathcal{R}} t\}$, $Post^*_{\mathcal{R},\equiv}(t)=\{t'\in\mathcal{T}\mid t\overset{*}{\Rightarrow}_{\equiv,\mathcal{R}} t'\}$, et $Pre^*_{\mathcal{R},\equiv}(t)=\{t'\in\mathcal{T}\mid t'\overset{*}{\Rightarrow}_{\equiv,\mathcal{R}} t\}$. Soit $Post^*_{\mathcal{R}}(t)=Post^*_{\mathcal{R},\sim}(t)$ et $Pre^*_{\mathcal{R}}(t)=Pre^*_{\mathcal{R},\sim}(t)$. Nous écrivons également $\mathcal{R}(t)$ pour représenter $Post_{\mathcal{R}}(t)$ et $\mathcal{R}^*(t)$ pour

[1] Dans la définition introduite par Mayr, l doit être différent de 0. Ici, nous ne considérons pas cette restriction.

représenter $Post^*_{\mathcal{R}}(t)$. Ces définitions sont étendues aux ensembles de termes de manière standard . Nous omettons l'indice $\mathcal{R}$ quand aucune confusion n'est possible.

Etant donné un PRS $\mathcal{R}$, nous dénotons par $\mathcal{R}^{-1}$ le PRS obtenu en échangeant les membres gauches et droits des règles de $\mathcal{R}$. Notons que pour chaque ensemble de termes de processus $\mathcal{L}$, $Pre^*_{\mathcal{R},\equiv}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},\equiv}(\mathcal{L})$.

4.2.3 Forme normale

Nous définissons une forme normale des PRS comme suit :

Définition 4.2.3 *Un PRS $\mathcal{R}$ est en forme normale ssi toutes les règles de $\mathcal{R}$ ont l'une des deux formes suivantes :*
Par-règle $t_1 \rightarrow t_2$ *où* t_1 *et* t_2 *sont de la forme 0,* X*, ou* $X||Y$ *;*
Seq-règle $t_1 \rightarrow t_2$ *où* t_1 *et* t_2 *sont de la forme 0,* X*, ou* $X \cdot Y$ *;*
où X *et* Y *sont des variables de processus.*

Nous montrons que l'analyse d'accessibilité des PRS peut être réduite à l'analyse d'accessibilité des PRS sous forme normale.

Lemme 4.2.1 *Soit $\mathcal{R}$ un PRS défini sur les variables de Var, et $\mathcal{L}$ un ensemble de termes de processus sur Var. Alors, nous pouvons effectivement calculer un PRS $\mathcal{R}'$ sous forme normale sur un ensemble de variables $Var' \supseteq Var$, et deux relations régulières d'arbres S_1 et S_2 telles que $Post^*_{\mathcal{R}}(\mathcal{L})$ est égal aux termes de*

$$S_2\Big(Post^*_{\mathcal{R}'}\big(S_1(\mathcal{L})\big)\Big)$$

qui ne contiennent que des variables de Var.

Preuve :
Ceci découle directement du résultat suivant prouvé dans [May98] :

> Soit $\mathcal{R}$ un PRS utilisant des variables de Var. Soient t_1 et t_2 deux termes de processus sur Var. Alors, nous pouvons effectivement calculer un PRS $\mathcal{R}'$ sur un ensemble de variables $Var' \supseteq Var$, et deux termes t'_1 et t'_2, obtenus respectivement par des substitutions de t_1 et t_2, tels que :
>
> 1. toutes les règles de $\mathcal{R}'$ sont soit des seq-règles, soit des règles de la forme $X_1||\cdots||X_k \rightarrow Y_1||\cdots||Y_n$, où les X_i et les Y_i sont des variables de processus ;
> 2. $t_2 \in Post^*_{\mathcal{R}}(t_1) \Longleftrightarrow t'_2 \in Post^*_{\mathcal{R}'}(t'_1)$.

Pour obtenir notre lemme, il suffit de voir qu'une règle de la forme

$$X_1||\cdots||X_k \rightarrow Y_1||\cdots||Y_n$$

peut être simulée par des par-règles : une règle $u_1||u_2 \rightarrow u$ peut être remplacée par les règles $u_1 \rightarrow Z_1$, $u_2 \rightarrow Z_2$, et $Z_1||Z_2 \rightarrow u$. De même, une règle $u \rightarrow u_1||u_2$ est simulée par les règles $u \rightarrow Z_1||Z_2$, $Z_1 \rightarrow u_1$, et $Z_2 \rightarrow u_2$. □

Nous supposons alors dans les chapitres 5 et 6 consacrés à l'analyse d'accessibilité des PRS, que les PRS considérés sont sous forme normale.

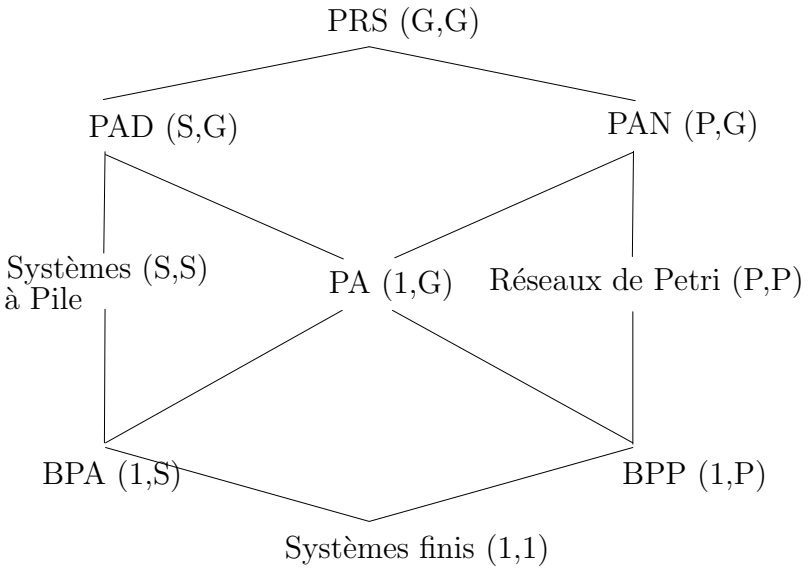


FIG. 4.2 – La hiérarchie PRS

4.3 Les sous classes de PRS

PRS comprend des classes correspondant à des modèles bien connus de systèmes infinis tels que les systèmes à piles, les réseaux de Petri, les algèbres de processus BPP, BPA, PA, etc. Ces classes sont obtenues en considérant différentes restrictions sur les termes apparaissant dans les membres gauches et droits des règles. La figure 4.2 montre une description de la hiérarchie des modèles (α, β)-PRS. Cette hiérarchie est stricte par rapport à la bisimulation des graphes de transitions générés [May98] [2]. Différentes classes de cette hiérarchie ont été étudiées de manière intensive ces dernières années. Dans [May98], Mayr donne une bonne vue d'ensemble sur les différents résultats connus pour ces classes. Nous montrons ci-après le lien entre ces différents modèles connus et les (α, β)-PRS.

4.3.1 (P, P)-PRS = Systèmes de réécriture de multiensembles

Un (P, P)-PRS est un PRS dont toutes les règles sont de la forme $X_1||\cdots||X_i \rightarrow Y_1||\cdots||Y_k$. Puisque l'opérateur "||" est associatif et commutatif, l'unique information importante codée dans un paral-terme $X_1||\cdots||X_k$ est le nombre d'occurrences de chaque variable. Par conséquent, chaque paral-terme $X_1||\cdots||X_k$ peut être vu comme un multiensemble, et un (P, P)-PRS comme un système de réécriture de multiensembles.

Ces systèmes sont aussi équivalents aux réseaux de Petri. En effet, nous pouvons

[2] Dans la définition introduite par Mayr, les règles PRS sont sous la forme $t_1 \xrightarrow{a} t_2$ où a est une action.

traduire un (P, P)-PRS vers un réseau de Petri et vice-versa comme suit : chaque variable de processus X correspond à une place p du réseau de Petri, et chaque paral-terme correspond à un marquage. Le nombre d'occurrences d'une variable X dans un paral-terme correspond au nombre de jetons présents dans la place p. Chaque règle du (P, P)-PRS correspond à une transition du réseau de Petri. Elle ne peut être appliquée que s'il y a assez de variables dans le paral-terme, c-à-d., s'il y a assez de jetons dans les places correspondantes du réseau de Petri. Dans ce document, nous ne faisons pas de distinction entre un (P, P)-PRS, un système de réécriture de multiensembles, et un réseau de Petri.

Ces systèmes sont utilisés pour modéliser les programmes parallèles concurrents avec création dynamique de processus et synchronisation, mais sans récursion [BCR01, DBR02].

4.3.2 (S, S)-PRS = Systèmes de réécriture préfixe

(S, S)-PRS correspond par définition à la classe des systèmes de réécriture préfixe. Ces systèmes sont équivalents aux automates à pile, qui sont des automates qui manipulent des piles et qui ont des règles de la forme

$$\langle p, \gamma \rangle \overset{a}{\hookrightarrow} \langle q, w \rangle$$

exprimant que si l'automate est dans l'état de contrôle p avec le symbole γ en tête de pile, il peut passer à l'état de contrôle q, dépiler γ, et empiler w (voir le chapitre 7 pour la définition formelle des automates à pile).

Les transitions d'un automate à pile peuvent être représentées par un (S, S)-PRS comme suit : La règle ci-dessus peut être représentée par la (S, S)-PRS règle

$$p \cdot \gamma \overset{a}{\rightarrow} q \cdot w.$$

Pour l'autre sens, Caucal [Cau92] a montré qu'un (S, S)-PRS est équivalent à un système à pile, dans le sens de l'isomorphisme des systèmes de transitions générés.

Ces systèmes sont utilisés pour le model-checking des programmes séquentiels récursifs ayant des variables globales et des appels récursifs de procédures [EK99, ES01].

4.3.3 $(1, S)$-PRS = Basic Process Algebra (BPA)

Un $(1, S)$-PRS est un PRS dont toutes les règles sont de la forme $X \rightarrow X_1 \cdots X_n$, c-à-d., ils sont des systèmes de réécriture préfixe où tous les membres gauches des règles sont des constantes. Ils peuvent être vus comme des systèmes à pile avec un seul état de contrôle. Ces modèles sont donc plus faibles que les systèmes à pile. Cette classe est aussi équivalente à la classe des processus hors-contexte [BE97], et à celle des Basic Process Algebra (BPA) de Bergstra et Klop [BK85].

4.3.4 $(1, P)$-PRS = BPP

Un $(1, P)$-PRS est un PRS dont les règles sont de la forme $X \rightarrow X_1 || \cdots || X_i$. Ils sont donc des systèmes de réécriture de multiensembles où les membres gauches des

règles sont des constantes. Ces systèmes sont équivalents aux réseaux de Petri sans synchronisation. Ils sont aussi équivalents aux Basic Parallel Processes (BPP), une sous classe de CCS introduite par Christensen [Chr93].

4.3.5 $(1, G)$-PRS = Systèmes PA

Ces systèmes sont des PRS dont les membres gauches des règles sont des constantes. Ils peuvent être vus comme l'union d'un BPA et d'un BPP. Le lien entre programmes parallèles et systèmes PA a été établi dans [EK99, EP00], où les auteurs proposent d'abstraire un programme parallèle par un système PA en oubliant la synchronisation et les résultats de retour des procédures appelées.

4.3.6 (S, G)-PRS = PAD

Ces systèmes correspondent à des PRS dont tous les membres gauches des termes sont des seq-termes. Le nom de la classe a été introduit par Mayr pour exprimer que ces systèmes sont l'union d'un système PA et d'un système de réécriture préfixe (ou système PushDown) : PAD=PA+PD. Ces systèmes peuvent aussi être vus comme l'union d'un BPP et d'un système de réécriture préfixe. Nous proposons dans la section 4.4.4 une transformation qui permet de calculer un PAD qui abstrait un programme parallèle récursif. L'absraction que nous proposons est plus précise que celle donnée dans [EK99, EP00] puisqu'elle tient compte des valeurs de retour des procédures appelées.

4.3.7 (P, G)-PRS = PAN

Le nom de la classe a été introduit par Mayr pour exprimer que ces systèmes comprennent à la fois PA et les réseaux de Petri (PN) : PAN=PA+PN. Ces systèmes permettent de modéliser les programmes parallèles avec création dynamique et synchronisation, mais sans variables de retour.

4.3.8 PRS$[\mathcal{C}]$

Comme nous allons nous intéresser au calcul des ensembles des accessibles, nous pouvons toujours supposer qu'un PRS $\mathcal{R}$ est une union d'un système de réécriture préfixe $\mathcal{R}_s$ et d'un système de réécriture de multiensembles $\mathcal{R}_p$ (lemme 4.2.1). Soit $\mathcal{C}$ une classe de systèmes de réécriture de multiensembles, nous définissons PRS$[\mathcal{C}]$ comme la classe des PRS $\mathcal{R}$ qui sont l'union d'un système de réécriture préfixe et d'un système de réécriture de multiensembles dans $\mathcal{C}$. Par exemple, PRS[BPP] est précisément la classe PAD.

Etant donnée une classe $\mathcal{C}$ de PRS, nous dénotons par co-$\mathcal{C}$ la classe *duale* de $\mathcal{C}$, qui est la classe de tous les systèmes $\mathcal{R}$ tels que $\mathcal{R}^{-1}$ est dans $\mathcal{C}$. Il est alors clair que PRS=co-PRS, et il en est de même pour les classes des systèmes de réécriture préfixe et de multiensembles.

4.4 Modéliser les PFG par des PRS

Nous montrons dans cette section comment passer d'un programme donné sous forme d'un PFG à un système PRS qui a en général plus de comportements que le programme initial (ce qui permet de s'assurer de la correction de ce dernier par rapport à une propriété de sûreté si le système PRS satisfait cette propriété). Nous donnons des conditions suffisantes que doit satisfaire un PFG d'un programme pour que le PRS obtenu modélise exactement ses comportements. Nous proposons également une transformation qui permet d'abstraire un programme quelconque vers un PAD. Cette abstraction est utile puisque nous donnons dans les chapitres 5 et 6 des algorithmes de calcul des ensembles des accessibles de ces systèmes, ce calcul n'étant pas toujours possible pour tous les PRS.

4.4.1 Ensemble des termes de processus

Pour modéliser un programme par un PRS, nous procédons comme suit : L'ensemble des variables de processus Var est l'ensemble de toutes les paires (n, loc) du programme, et les termes de processus correspondent aux arbres d'exécutions. Plus précisément, nous associons à chaque arbre d'exécutions un terme de processus défini de manière inductive comme suit :

- une feuille (n, loc) de l'arbre est représentée par la variable (n, loc) ;
- un arbre de la forme $(n, loc)(A)$, où A est un sous-arbre, est représenté par le terme $t \cdot (n, loc)$, où t est le terme correspondant à l'arbre A. Avec la sémantique de réécriture préfixe du "$\cdot$", ceci préserve le fait que (n, loc) ne peut pas se réécrire (n'est pas actif) tant que t n'est pas nul (c-à-d., tant que A n'a pas terminé son exécution) ;
- un arbre de la forme $(n, loc)(A_1, \ldots, A_k)$, où les A_i sont des sous-arbres, est représenté par le terme $(t_1 || \cdots || t_k) \cdot (n, loc)$, où les t_i sont les termes correspondant aux arbres A_i. Comme le "$\cdot$" a une stratégie de réécriture préfixe, ceci exprime aussi que ce sont les t_i (ou de manière équivalente les A_i) qui s'exécutent en premier en parallèle, et qu'à leur terminaison, (n, loc) devient actif.

Nous disons que n ou (n, loc) est actif dans un terme t ssi il l'est dans l'arbre d'exécutions correspondant à t.

4.4.2 Ensemble des règles

Avec cette modélisation, la sémantique des PFG, décrite précédemment (dans la section 4.1) en termes d'arbres d'exécutions, peut être représentée de manière directe par des règles PRS comme suit :

Une *instruction de base* $n_1 \xrightarrow{inst} n_2$ est représentée par une règle de la forme

$$(n_1, loc_1) \rightarrow (n_2, loc_2)$$

où loc_1 et loc_2 sont les valeurs des variables avant et après l'application de cette instruction.

Un *appel récursif* $n_1 \xrightarrow{x:=call(p)} n_2$ est représenté par la règle d'appel

$$(n_1, loc) \to (e_p, loc_p) \cdot (n_2, loc)$$

et la règle de retour de résultat

$$(r_p, loc'_p) \cdot (n_2, loc) \to (n_2, loc')$$

où les valeurs des variables locales sont telles que décrites précédemment, c-à-d., loc_p correspond aux valeurs des arguments passés à la procédure p, loc'_p mémorise les valeurs des variables de p à sa terminaison, et loc' est tel que les variables différentes de x gardent leurs valeurs de loc, et x prend la valeur retournée par p et déterminée à partir de loc'_p.

Un *appel parallèle* $n_1 \xrightarrow{(x_1,\ldots,x_k):=call(p_1||\cdots||p_k)} n_2$ est modélisé par la règle d'appel parallèle

$$(n_1, loc) \to \big((e_{p_1}, loc_1)|| \cdots ||(e_{p_k}, loc_k)\big) \cdot (n_2, loc)$$

et la règle de retour des résultats

$$\big((r_{p_1}, loc'_1)|| \cdots ||(r_{p_k}, loc'_k)\big) \cdot (n_2, loc) \to (n_2, loc')$$

Ici aussi les valeurs des variables locales sont telles que définies dans la description de la sémantique des PFG, c-à-d., loc_i correspond aux valeurs des arguments passés à la procédure p_i, loc'_i sont les valeurs des variables locales de p_i à sa terminaison, et loc' est tel que les x_i prennent les valeurs retournées par les p_i déterminées à partir de loc'_i, et les autres variables gardent leurs valeurs dans loc.

Seulement, pour modéliser la *synchronisation* représentée par deux instructions de la forme $n_1 \xrightarrow{a!} m_1$ et $n_2 \xrightarrow{a?} m_2$ il nous faut une infinité de règles PRS. En effet, ces instructions expriment que si les deux points n_1 et n_2 sont actifs, alors ils peuvent se synchroniser à travers le signal a et passent aux points de contrôle m_1 et m_2. Donc, pour représenter ces instructions, nous devons avoir une règle de la forme

$$(n_1, loc_1)||(n_2, loc_2) \to (m_1, loc_1)||(m_2, loc_2)^3$$

qui permet aux points parallèles n_1 et n_2 de se synchroniser et d'avancer simultanément. Nous devons aussi avoir la règle

$$\big((n_1, loc_1) \cdot (l_1, loc'_1)\big)||(n_2, loc_2) \to \big((m_1, loc_1) \cdot (l_1, loc'_1)\big)||(m_2, loc_2)$$

puisque (n_1, loc_1) et (n_2, loc_2) sont actifs dans le terme $\big((n_1, loc_1) \cdot (l_1, loc'_1)\big)||(n_2, loc_2)$. En fait, nous devons considérer toutes les règles dont le membre gauche est de la forme

$$\big((n_1, loc_1) \cdot (l_1, loc'_1) \cdots (l_k, loc'_k)\big)||\big((n_2, loc_2) \cdot (l'_1, loc''_1) \cdots (l'_{k'}, loc''_{k'})\big)$$

puisque dans ces termes (n_1, loc_1) et (n_2, loc_2) sont actifs. Comme il y a une infinité de tels termes, nous ne pouvons, dans le cas général, représenter la synchronisation de manière exacte par des règles PRS.

Pour contourner ce problème, nous procédons en deux étapes :

[3]Comme "||" est associatif/commutatif, cette règle peut s'appliquer à tous les termes où (n_1, loc_1) et (n_2, loc_2) sont séparés par des "||".

1. Nous détectons les points du programme pour lesquels il n'est pas possible d'atteindre ce nombre infini de situations à partir du point initial du programme. Plus précisément, nous détectons les points n_1, n_2 tels que à chaque fois que deux occurrences de n_1 et n_2 sont actives dans un terme accessible t, alors elles apparaissent dans un sous terme de t de la forme

$$\big((n_1, loc_1) \cdot (l_1, loc'_1) \cdots (l_k, loc'_k)\big) || \big((n_2, loc_2) \cdot (l'_1, loc''_1) \cdots (l'_{k'}, loc''_{k'})\big)$$

où k et k' sont bornés par une constante fixée K (les termes sont bien sûr considérés modulo associativité/commutativité du "||"). Pour ces points, la synchronisation peut être représentée de manière exacte par les règles PRS suivantes :
$\big((n_1, loc_1) \cdot (l_1, loc'_1) \cdots (l_k, loc'_k)\big) || \big((n_2, loc_2) \cdot (l'_1, loc''_1) \cdots (l'_{k'}, loc''_{k'})\big) \rightarrow \big((m_1, loc_1) \cdot (l_1, loc'_1) \cdots (l_k, loc'_k)\big) || \big((m_2, loc_2) \cdot (l'_1, loc''_1) \cdots (l'_{k'}, loc''_{k'})\big)$ où $k, k' \leq K$.

2. Pour les points qui ne satisfont pas cette condition et qui peuvent se synchroniser, comme nous n'avons aucun moyen de modéliser cette synchronisation, nous leur permettons d'avancer chacun de son côté sans communiquer avec ses partenaires. Ceci introduit dans le modèle des comportements qui n'apparaissent pas dans le vrai programme, mais évite d'introduire dans le modèle des blocages qui n'existeraient pas dans le vrai programme. Donc si n_1 et n_2 sont deux points quelconques du programme, nous modélisons les deux instructions $n_1 \xrightarrow{a!} m_1$ et $n_2 \xrightarrow{a?} m_2$ par les deux règles

$$(n_1, loc_1) \rightarrow (m_1, loc_1) \text{ et } (n_2, loc_2) \rightarrow (m_2, loc_2) \tag{4.1}$$

qui permettent à chaque point d'avancer tout seul sans se synchroniser.
Comme ces règles introduisent dans le modèle des comportements qui n'existent pas dans le programme, nous pouvons déduire que le vrai programme est correct par rapport à une propriété de sûreté si ce modèle l'est. Cette façon de faire est la seule manière que nous avons trouvé pour modéliser, quand il est possible, la synchronisation, tout en évitant d'introduire des blocages.

Dans ce qui suit, nous nous restreignons pour simplifier la présentation à la détection des points où la seule situation possible, atteignable à partir du point initial du programme, où n_1 et n_2 sont tous les deux actifs est d'avoir un sous terme de la forme (modulo associativité/commutativité du "||")

$$(n_1, loc_1) || (n_2, loc_2)$$

C'est-à-dire qu'aucun terme contenant un sous terme de la forme

$$\big((n_1, loc_1) \cdot (l_1, loc'_1) \cdots (l_k, loc'_k)\big) || \big((n_2, loc_2) \cdot (l'_1, loc''_1) \cdots (l'_{k'}, loc''_{k'})\big)$$

où k et k' sont ≥ 1 ne peut être atteint à partir du point de départ du programme. Nous disons que ces points satisfont la condition **Cond**. Pour ces points, nous représentons la synchronisation de manière exacte par la règle PRS

$$(n_1, loc_1) || (n_2, loc_2) \rightarrow (m_1, loc_1) || (m_2, loc_2)$$

Nous considérons bien sûr les règles 4.1 pour les points qui ne satisfont pas cette condition. Notre modélisation peut être étendue de manière directe aux autres cas (où $K > 1$). Nous proposons dans la section suivante une condition qui permet de détecter de tels points.

4.4.2.1 Points bien reliés

Pour détecter les points du programme qui satisfont la condition **Cond**, nous définissons un graphe qui détermine les dépendances entre les différents points du PFG, et nous donnons une condition statique sur ce graphe qui permet de déterminer si deux points satisfont **Cond**. Nous introduisons alors la notion de *graphe relationnel* comme suit :

Définition 4.4.1 (Graphe relationnel) *Soit $G = \{G_p \mid p \in \texttt{Proc}\}$ un parallel flow graph où $G_p = (N_p, E_p, e_p, r_p)$.* Le graphe relationnel $\mathcal{G}$ *de G est défini comme le graphe contenant les arêtes suivantes :*

1. *Si $(n_1, inst, n_2) \in E$, où inst est une instruction de base, alors (n_1, n_2) est un arc de $\mathcal{G}$,*
2. *Si E contient un arc $arc_i = \big(n_1, x := call(p), n_2\big)$, alors $(n_1, \boxdot_i)$, $(\boxdot_i, g, e_p)$ et $(\boxdot_i, d, n_2)$ sont des arcs de $\mathcal{G}$,*
3. *Si E contient un arc $arc_i = \big(n_1, (x_1, \ldots, x_k) := call(p_1 || \cdots || p_k), n_2\big)$, alors $(n_1, \boxdot_i)$, $(\boxdot_i, g, ||_i)$, $(\boxdot_i, d, n_2)$, $(||_i, e_{p_j})$ pour $1 \leq j \leq k$ sont des arcs de $\mathcal{G}$.*

Intuitivement, le *graphe relationnel* décrit les relations entre les différents points de contrôle du programme en explicitant les appels des procédures. Nous expliquons ci-dessous l'intuition exprimée par les différents arcs de $\mathcal{G}$:

- L'arc (n_1, n_2) défini au point (1) exprime que le programme peut passer du point de contrôle n_1 au point de contrôle n_2 en exécutant une instruction qui n'est pas un appel de procédures,
- Les arcs définis au point (2) expriment que si du point de contrôle n_1 le programme appelle la procédure p et passe à n_2 après la terminaison de celle-ci, ceci veut dire que de n_1, le programme exécute séquentiellement (dans l'arc $(n_1, \boxdot_i)$, "$\boxdot_i$" représente un appel séquentiel) d'abord, la procédure p (cet appel est représenté par l'arc $(\boxdot_i, g, e_p)$, où e_p est le point d'entrée de la procédure p), et ensuite, quand l'exécution de cette procédure aurait terminé, il passe au point n_2 (ceci est représenté par l'arc $(\boxdot_i, d, n_2)$). Dans ces deux arcs, $(\boxdot_i, g, e_p)$ et $(\boxdot_i, d, n_2)$, les indices "g" (pour *gauche*) et "d" (pour *droite*) expriment que d'abord le programme "va à gauche" et exécute p, et ensuite, après avoir fini cette exécution, il "va à droite" et passe au point n_2.
- Le point (3) exprime que si du point de contrôle n_1 le programme appelle les procédures $p_1, \ldots, p_k$ en parallèle et passe à n_2 après la terminaison de celles-ci, ceci veut dire que de n_1, le programme exécute séquentiellement (l'arc $(n_1, \boxdot_i)$) d'abord, les procédures $p_1, \ldots, p_k$ en parallèle (l'arc $(\boxdot_i, g, ||_i)$, où "$||_i$" représente un appel parallèle), et ensuite, quand l'exécution de cet appel parallèle aurait terminé, il passe au point n_2 (ceci est représenté par l'arc $(\boxdot_i, d, n_2)$). Ici aussi, les indices "g" et "d" expriment que le programme va d'abord à gauche pour

exécuter l'appel parallèle, et ensuite, quand ce dernier aurait fini, il passe à droite au point n_2. L'appel parallèle est représenté par les arcs $(||_i, e_{p_j})$, $1 \leq j \leq k$.

Nous donnons dans la section 4.4.2.2 un exemple qui illustre la construction des graphes relationnels associés aux PFGs. Nous définissons maintenant la notion de points bien reliés comme suit :

Définition 4.4.2 (Points bien reliés) *Soit G un parallel flow graph dont l'ensemble des nœuds est N, et soit $\mathcal{G}$ le graphe relationnel associé. Deux points n_1 et n_2 de N sont* bien reliés *ssi si $||_i$ est un ancêtre commun à n_1 et n_2 dans $\mathcal{G}$, alors tous les chemins qui mènent de $||_i$ à n_1 et tous ceux qui mènent de $||_i$ à n_2 sont étiquetés par des étiquettes σ telles que $\sigma \in d^*$.* [4]

Nous avons alors :

Lemme 4.4.1 *Soient n_1 et n_2 deux points bien reliés, et soit t un terme représentant une configuration atteignable à partir du point initial du programme. Si deux occurrences de n_1 et n_2 sont actives dans t, alors elles appraissent dans un sous-terme de t de la forme $(n_1, loc_1)||(n_2, loc_2)$.* [5]

Preuve : Ceci est dû au fait que si deux points n_1 et n_2 peuvent être simultanément actifs, alors ils ont forcément dans le graphe relationnel $\mathcal{G}$ un ancêtre commun étiqueté par $||_k$ (pour un certain indice k). Soient alors n et n' les deux points tels que l'on ait dans $\mathcal{G}$ les arcs $(n, \boxdot_k)$, $(\boxdot_k, g, ||_l)$, et $(\boxdot_k, d, n')$.

Le fait que les points n_1 et n_2 soient bien reliés implique que si sur le chemin menant de ce nœud "$||_k$" à n_j ($j \in \{1, 2\}$) il y a un nœud étiqueté par $\boxdot_i$, alors n_j se trouve sur la branche droite de $\boxdot_i$ (le premier arc de cette branche est étiqueté par "d"). Cette situation exprime que si n_1 et n_2 ont été appelés en parallèle par n, et sont donc en train d'être exécutés en parallèle, alors le point n' est le premier point qui est en attente de leur résultat. En effet, un nœud $\boxdot_i$ exprime que d'abord la branche correspondant à "$(\boxdot_i, g, m_1) \cdots$" est exécutée, et quand cette exécution termine, le programme exécute la deuxième branche "$(\boxdot_i, d, m_2) \cdots$" en fonction du résultat retourné par la première branche. Donc, comme il n'y a pas d'arcs de la forme "$(\boxdot_i, g, m)$" le long de cette branche qui mène de $||_k$ à n_j, il n'y a pas de points qui sont en train d'attendre le résultat de retour de n_1 ou n_2. Donc, il n'y a pas de points m_j tels que l'on ait un sous-terme de la forme $(n_j, loc_j) \cdot (m_j, loc'_j) \cdots$.

□

Nous donnons dans la section 4.4.2.2 un exemple qui illustre ce lemme.

Nous déduisons à partir de ce lemme que si deux points sont bien reliés, alors ils satisfont la condition **Cond**. De ce fait, nous modélisons la synchronisation par des règles PRS comme suit : si le PFG contient deux instructions $n_1 \xrightarrow{a!} m_1$ et $n_2 \xrightarrow{a?} m_2$ alors :

[4] Nous considérons qu'un arc sans étiquette est annoté par le mot vide ϵ.

[5] Rappelons que "||" est associatif/commutatif, donc $(n_1, loc_1)||(n_2, loc_2)$ peut être considéré comme sous-terme de $(n_1, loc_1)||t'||(n_2, loc_2)$, etc.

– si n_1 et n_2 sont deux points bien reliés, considérer la règle

$$(n_1, loc_1)||(n_2, loc_2) \rightarrow (m_1, loc_1)||(m_2, loc_2)$$

– sinon, considérer les deux règles

$$(n_1, loc_1) \rightarrow (m_1, loc_1) \text{ et } (n_2, loc_2) \rightarrow (m_2, loc_2)$$

Observons que si tous les points n_1, n_2 du programme pour lesquels il y a des règles de synchronisation de la forme $n_1 \xrightarrow{a!} m_1$ et $n_2 \xrightarrow{a?} m_2$ sont bien reliés, alors notre modélisation du programme par le PRS est exacte, et le PRS considéré n'introduit pas de comportements supplémentaires dans le modèle. Nous définissons alors la classe des *programmes bien reliés* comme suit :

Définition 4.4.3 (Programme bien relié) *Soit P un programme donné par un parallel flow graph G. Soit $\mathcal{G}$ le graphe relationnel associé à G. P est* bien relié *ssi si E comprend deux arcs de la forme $(n_1, a?, m_1)$ et $(n_2, a!, m_2)$, alors les points n_1 et n_2 sont bien reliés dans $\mathcal{G}$.*

Nous obtenons alors de manière directe à partir de notre modélisation et du lemme 4.4.1 que :

Lemme 4.4.2 *Soit P un programme bien relié, et soit R le PRS correspondant. Soient $(e_{\texttt{main}}, loc)$ le point de départ du programme, A un arbre d'exécution, et t le terme de processus correspondant à A. Alors A représente une configuration accessible du programme à partir de $(e_{\texttt{main}}, loc)$ ssi il existe une dérivation du PRS R qui mène de $(e_{\texttt{main}}, loc)$ à t.*

Donc dans le cas des programmes bien reliés, le PRS obtenu décrit *exactement* la sémantique du programme telle que présentée dans la section 4.1.

Les programmes séquentiels sans parallélisme sont un cas particulier de programmes bien reliés. Notons que dans ce cas, notre traduction coïncide avec la transformation des programmes séquentiels vers les automates à pile donnée dans [ES01]. Nous détaillons cette transformation au chapitre 7.

4.4.2.2 Exemple

Considérons le PFG de la figure 4.3 où la procédure `main` appelle deux procédures p_1 et p_2 en parallèle. Ces procédures font appel à une troisième procédure p_3. Les trois procédures se synchronisent à travers les signaux c et d. Nous représentons à la figure 4.4 le graphe relationnel associé à ce PFG.

Pour illustrer notre modélisation de la synchronisation, nous ne tenons pas compte des variables locales (pour simplifier la présentation). Donc, nos variables de processus seront juste les points de contrôle n. Par exemple, la configuration de départ est représentée par le point d'entrée $e_{\texttt{main}}$. Nous avons alors parmi les règles PRS qui modélisent ce PFG les règles suivantes (nous ne considérons que les règles pertinentes pour expliquer notre modélisation de la synchronisation) :

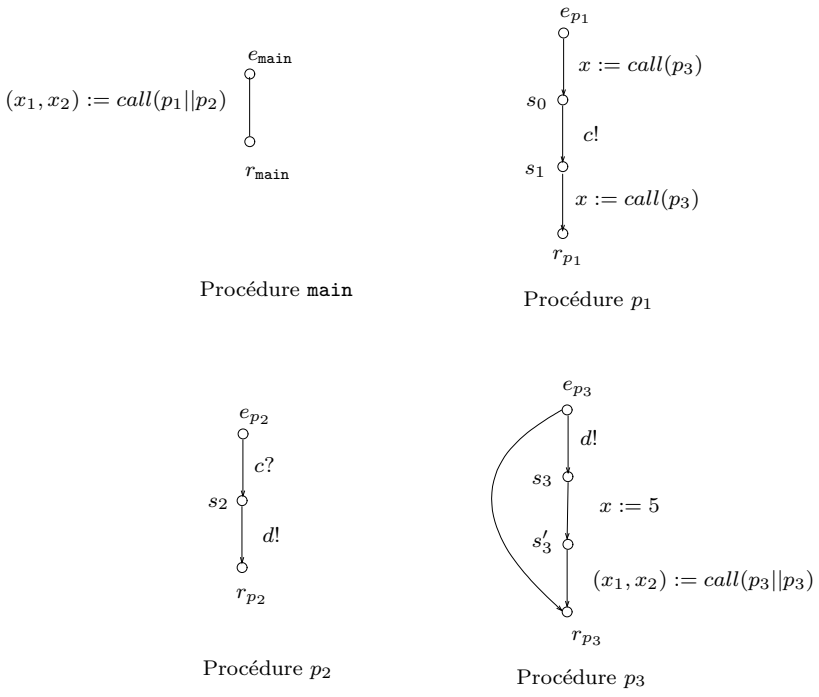

$e_{\texttt{main}}$
$(x_1, x_2) := call(p_1||p_2)$
$r_{\texttt{main}}$
Procédure `main`
e_{p_1}
$x := call(p_3)$
s_0
c!
s_1
$x := call(p_3)$
r_{p_1}
Procédure p_1
e_{p_2}
$c?$
s_2
$d!$
r_{p_2}
Procédure p_2
e_{p_3}
$d!$
s_3
$x := 5$
s'_3
$(x_1, x_2) := call(p_3||p_3)$
r_{p_3}
Procédure p_3

FIG. 4.3 – Un PFG

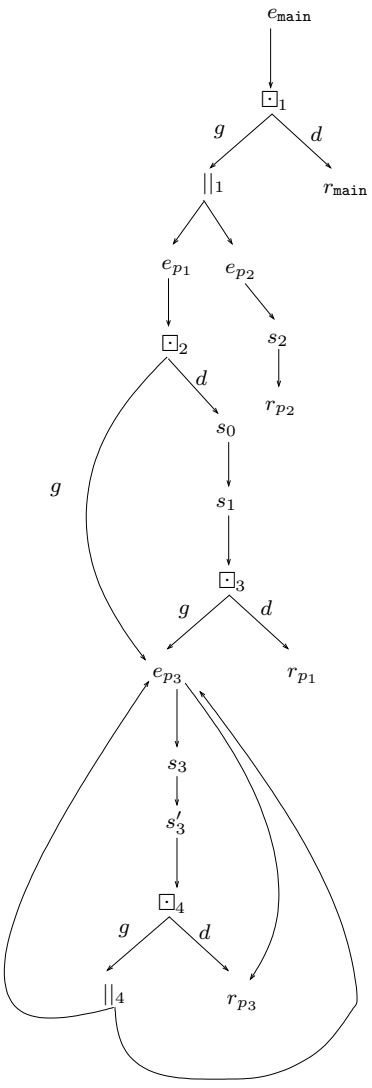


FIG. 4.4 – Le graphe relationnel $\mathcal{G}$ correspondant au PFG de la figure 4.3

1. la règle $R_1 = e_{\texttt{main}} \rightarrow (e_{p_1}||e_{p_2}) \cdot r_{\texttt{main}}$ correspondant à l'arc suivant du PFG
$$e_{\texttt{main}} \xrightarrow{(x_1,x_2):=call(p_1||p_2)} r_{\texttt{main}}$$
2. les règles $R_2 = e_{p_1} \rightarrow e_{p_3} \cdot s_0$ et $R'_2 = r_{p_3} \cdot s_0 \rightarrow s_0$ correspondant à l'arc $e_{p_1} \xrightarrow{x:=call(p_3)} s_0$.
3. la règle $R_3 = s_0||e_{p_2} \rightarrow s_1||s_2$ correspondant aux arcs $s_0 \xrightarrow{c!} s_1$ et $e_{p_2} \xrightarrow{c?} s_2$ puisque s_0 et e_{p_2} sont bien reliés dans le graphe relationnel $\mathcal{G}$. En effet, leur seul ancêtre commun étiqueté par $||_i$ est $||_1$; et les seuls chemins qui mènent de $||_1$ à s_0 et e_{p_2} sont respectivement $||_1 \rightarrow e_{p_1} \rightarrow \boxdot_2 \xrightarrow{d} s_0$ et $||_1 \rightarrow e_{p_2}$ qui sont respectivement étiquetés par d et ϵ.
4. la règle $R_4 = s_1 \rightarrow e_{p_3} \cdot r_{p_1}$ correspondant à l'arc $s_1 \xrightarrow{x:=call(p_3)} r_{p_1}$.
5. les règles $R_5 = s_2 \rightarrow r_{p_2}$ et $R_6 = e_{p_3} \rightarrow s_3$ puisque nous avons les arcs de synchronisation $e_{p_3} \xrightarrow{d?} s_3$ et $s_2 \xrightarrow{d!} r_{p_2}$, alors que e_{p_3} et s_2 ne sont pas bien reliés. En effet, dans $\mathcal{G}$, ils ont un ancêtre commun $||_1$ tel que le chemin qui mène de $||_1$ à e_{p_3} contient des étiquettes "g".
6. la règle $R_7 = e_{p_3} \rightarrow r_{p_3}$ correspondant à l'arc $e_{p_3} \rightarrow r_{p_3}$.

Donc si nous partons du point de départ du programme $e_{\texttt{main}}$, nous obtenons les dérivations suivantes :

$$\begin{aligned}
e_{\texttt{main}} &\xrightarrow{R_1} (e_{p_1}||e_{p_2}) \cdot r_{\texttt{main}} \\
&\xrightarrow{R_2} \big((e_{p_3} \cdot s_0)||e_{p_2}\big) \cdot r_{\texttt{main}} \\
&\xrightarrow{R_7} \big((r_{p_3} \cdot s_0)||e_{p_2}\big) \cdot r_{\texttt{main}} \\
&\xrightarrow{R'_2} (s_0||e_{p_2}) \cdot r_{\texttt{main}} \\
&\xrightarrow{R_3} (s_1||s_2) \cdot r_{\texttt{main}} \\
&\xrightarrow{R_4} \big((e_{p_3} \cdot r_{p_1})||s_2\big) \cdot r_{\texttt{main}} \\
&\xrightarrow{R_5} \big((e_{p_3} \cdot r_{p_1})||r_{p_2}\big) \cdot r_{\texttt{main}} \\
&\xrightarrow{R_6} \big((s_3 \cdot r_{p_1})||r_{p_2}\big) \cdot r_{\texttt{main}}
\end{aligned}$$

Nous voyons alors que comme s_0 et e_{p_2} sont bien reliés, alors s'ils sont tous les deux actifs, il y a forcément un sous-terme de la forme $s_0||e_{p_2}$. Dans ce cas, ces deux points peuvent se synchroniser de manière exacte (règle R_3). Quant aux points e_{p_3} et s_2 comme ils ne sont pas bien reliés, nous voyons que dans le terme accessible $\big((e_{p_3} \cdot r_{p_1})||s_2\big) \cdot r_{\texttt{main}}$ ils sont tous les deux actifs mais pas sous la forme $s_2||e_{p_3}$. Donc, si on n'avait pas considéré les règles R_5 et R_6, et qu'à la place, on avait considéré une règle de la forme $s_2||e_{p_3} \rightarrow r_{p_2}||s_3$, elle n'aurait pas pû s'appliquer, et on n'aurait pas eu le terme $\big((s_3 \cdot r_{p_1})||r_{p_2}\big) \cdot r_{\texttt{main}}$. Donc on aurait eu moins de comportements que le vrai programme. C'est pour ceci que nous considérons les règles R_5 et R_6 qui peuvent introduire des comportements supplémentaires (tels que le terme $\big((e_{p_3} \cdot r_{p_1})||r_{p_2}\big) \cdot r_{\texttt{main}}$), mais qui assurent que tous les comportements du vrai programme sont obtenus par le PRS.

4.4.2.3 Représentation d'un PFG par un PRS : récapitulation

Nous résumons dans ce qui suit la modélisation que nous avons décrit jusque-là. Le PRS contient les règles suivantes, pour chaque procédure Π dans `Proc` :

1. *Instruction vide :* $(n_1, loc) \rightarrow (n_2, loc)$, si $n_1 \rightarrow n_2 \in E_\Pi$;

2. *Affectation :* $(n_1, loc) \rightarrow (n_2, loc')$, si $n_1 \xrightarrow{a} n_2 \in E_\Pi$ et a est une affectation, où loc et loc' sont les valeurs des variables locales de la procédure Π avant et après l'affectation ;

3. *Conditionnelle "if-then-else" :* $(n_1, loc) \rightarrow (n_2, loc)$, si $n_1 \xrightarrow{a} n_2 \in E_\Pi$ et a est une conditionnelle "if-then-else", où loc est tel que la condition de l'instruction est satisfaite ;

4. *Appel récursif :* si $n_1 \xrightarrow{x:=callp} n_2 \in E_\Pi$, alors considérer la règle d'appel

$$(n_1, loc) \rightarrow (e_p, loc_p) \cdot (n_2, loc)$$

où loc_p représente les valeurs des arguments passées par la procédure Π à la procédure p, et (n_2, loc) mémorise les variables locales de la procédure appelante Π ;
De plus, nous considérons la règle suivante qui modélise le retour de résultat de la procédure appelée p à Π :

$$(r_p, loc'_p) \cdot (n_2, loc) \rightarrow (n_2, loc')$$

où loc'_p représente les valeurs des variables locales de p à sa terminaison, loc mémorise les valeurs des variables locales de la procédure Π au moment de son appel à p, et dans loc' toutes les variables différentes de x ont la même valeur que dans loc, alors que x prend la valeur de retour de la procédure p. Cette valeur est déterminée à partir de loc'_p.

5. *Appel parallèle :* si $n_1 \xrightarrow{(x_1,\ldots,x_k):=call(p_1||\cdots||p_k)} n_2 \in E_\Pi$, alors considérer la règle d'appel parallèle

$$(n_1, loc) \rightarrow \big((e_{p_1}, loc_1)||\cdots||(e_{p_k}, loc_k)\big) \cdot (n_2, loc)$$

où loc_i représente les valeurs des arguments passées par la procédure Π à la procédure p_i, et comme précédemment, (n_2, loc) mémorise les variables locales de la procédure appelante Π ;
Nous considérons également la règle de retour des résultats

$$\big((r_{p_1}, loc'_1)||\cdots||(r_{p_k}, loc'_k)\big) \cdot (n_2, loc) \rightarrow (n_2, loc')$$

où dans loc', les variables différentes de $x_1, \ldots, x_n$ ont les mêmes valeurs que dans loc, et les valeurs des x_i sont déterminées à partir des valeurs des loc'_i. Ces valeurs correspondent aux résultats de retour des procédures p_i.

6. *Synchronisation :* si $n_1 \xrightarrow{a!} m_1 \in E_{\Pi_1}$ et $n_2 \xrightarrow{a?} m_2 \in E_{\Pi_2}$, alors :

   (a) Si n_1 et n_2 sont bien reliés, considérer la règle

   $$(n_1, loc_1)||(n_2, loc_2) \rightarrow (m_1, loc_1)||(m_2, loc_2')$$

   (b) sinon considérer les deux règles

   $$(n_1, loc_1) \rightarrow (m_1, loc_1) \text{ et } (n_2, loc_2) \rightarrow (m_2, loc_2)$$

Rappelons qu'en général, le PRS obtenu génère plus de comportements que le vrai programme, et que dans le cas des programmes bien reliés, notre traduction est exacte (lemme 4.4.2).

4.4.3 Analyse des PFGs

Le problème d'analyse des PFGs se réduit souvent à une analyse d'accessibilité [EK99]. Par exemple, on aimerait savoir si un certain point de contôle est accessible, ou déterminer les différentes valeurs que peut prendre une certaine variable (ce problème est connu sous le nom de "propagation de constantes"), etc. Le problème consiste alors à déterminer si un ensemble de configurations est accessible à partir de la configuration initiale du programme qui est représentée par la paire $(e_{\texttt{main}}, loc)$. Comme dans cette partie nous modélisons les programmes par des PRS, nous allons nous intéresser à l'analyse d'accessibilité dans ce modèle. De par notre modélisation donnée ci-dessus, cette étude doit se faire modulo toutes les équivalences structurelles entre les termes de processus, c-à-d., en considérant l'associativité du "$\cdot$" et l'associativité/commutativité du "||". En effet, si nous considérons le terme $(n_1, loc_1)||(n_2, loc_2)||(n_3, loc_3)$, alors la règle de synchronisation $(n_1, loc_1)||(n_3, loc_3)$ ne peut s'appliquer que modulo associativité/commutativité du "||". De ce fait, notre problème sera dans cette partie de déterminer si

$$Post^*_{\mathcal{R}}(\mathcal{L}_0) \cap \mathcal{L} = \emptyset$$

ou de manière équivalente, si

$$Pre^*_{\mathcal{R}}(\mathcal{L}) \cap \mathcal{L}_0 = \emptyset$$

où $\mathcal{L}$ et $\mathcal{L}_0$ sont des ensembles de termes de processus représentant des ensembles de configurations du programme. Le plus souvent $\mathcal{L}_0$ est la configuration initiale $(e_{\texttt{main}}, loc)$.

Il s'avère que dans certains cas, nous pouvons ne pas considérer les équivalences structurelles de l'opérateur "||". Considérons par exemple les programmes où le parallélisme est binaire, c-à-d. où les appels parallèles sont de la forme

$$n_1 \xrightarrow{(x_1,x_2):=call(p_1||p_2)} n_2$$

Alors en partant de la configuration initiale $(e_{\texttt{main}}, loc)$, nous ne pouvons atteindre que des termes où l'opérateur "||" est binaire. En effet, ce sont les règles PRS correspondant

aux appels parallèles $n_1 \xrightarrow{(x_1,x_2):=call(p_1||p_2)} n_2$ qui introduisent le parallélisme dans les termes. Dans ce cas, comme ces règles sont de la forme

$$(n_1, loc) \rightarrow \big((e_{p_1}, loc_1)||(e_{p_2}, loc_2)\big) \cdot (n_2, loc)$$

nous ne pouvons avoir qu'un parallélisme binaire dans les termes. Dans ce cas, il n'est donc pas possible d'avoir un sous-terme de la forme $(n_1, loc_1)||(n_2, loc_2)||(n_3, loc_3)$. Donc une règle de synchronisation $(n_1, loc_1)||(n_2, loc_2)$ ne peut s'appliquer qu'à des sous-termes de la forme $\big((n_1, loc_1)||(n_2, loc_2)\big) \cdot (n_3, loc_3)$. Cette règle peut alors s'appliquer sans considérer l'associativité/commutativité du "||".

Définissons alors cette classe de programmes :

Définition 4.4.4 (Programme à parallélisme binaire) *Soient P un programme donné par un parallel flow graph G. P est* à parallélisme binaire *ssi les arcs de E correspondants à des appels parallèles sont de la forme*

$$n_1 \xrightarrow{(x_1,x_2):=call(p_1||p_2)} n_2$$

Nous obtenons alors de manière directe d'après les explications précédentes :

Lemme 4.4.3 *Soit P un programme à parallélisme binaire, $\mathcal{R}$ le PRS correspondant, et $\mathcal{L}$ un ensemble de termes de processus. Alors*

$$Post^*_{\mathcal{R}}\big((e_{\texttt{main}}, loc)\big) = Post^*_{\mathcal{R},\sim_s}\big((e_{\texttt{main}}, loc)\big)$$

De plus, déterminer si $Pre^*_{\mathcal{R}}(\mathcal{L}) \cap (e_{\texttt{main}}, loc) = \emptyset$ est équivalent à déterminer si $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L}) \cap (e_{\texttt{main}}, loc) = \emptyset$.

Grâce à ce lemme, nous montrons dans le chapitre suivant qu'il est possible d'analyser de manière exacte les programmes à parallélisme binaire.

4.4.4 Abstraction des PFG vers les PAD

Nous proposons une traduction d'un PFG vers un PAD qui constitue une abstraction du vrai programme. Notre abstraction consiste à sur-approximer la synchronisation et à ignorer les résultats de retour des appels parallèles des procédures.

La traduction est définie comme précédemment à part ces deux cas :

Appel parallèle : si $n_1 \xrightarrow{(x_1,\ldots,x_k):=call(p_1||\cdots||p_k)} n_2 \in E_\Pi$, alors considérer la règle d'appel parallèle

$$(n_1, loc) \rightarrow \big((e_{p_1}, loc_1)|| \cdots ||(e_{p_k}, loc_k)\big) \cdot (n_2, loc)$$

où loc_i représente les valeurs des arguments passées par la procédure Π à la procédure p_i, et comme précédemment, (n_2, loc) mémorise les variables locales de la procédure appelante Π ;

Considérer également les règles suivantes qui modélisent la terminaison des procédures appelées pour que le contrôle revienne à la fin au point n_2 :

$$(r_{p_i}, loc'_i) \rightarrow 0$$

et ce, pour chaque i, $1 \leq i \leq k$.

Synchronisation : si $n_1 \stackrel{a!}{\rightarrow} m_1 \in E_{\Pi_1}$ et $n_2 \stackrel{a?}{\rightarrow} m_2 \in E_{\Pi_2}$, alors considérer les deux règles

$$(n_1, loc_1) \rightarrow (m_1, loc_1) \text{ et } (n_2, loc_2) \rightarrow (m_2, loc_2)$$

Cette abstraction que nous proposons est plus précise que la traduction donnée dans [EK99, EP00] qui modélise un PFG par un système PA. En effet, leur modélisation ne tient pas compte des résultats de retour des procédures séquentielles. Ceci revient à considérer la même traduction que pour PAD à part le cas de l'appel récursif qui est représenté comme suit :

si $n_1 \xrightarrow{x:=callp} n_2 \in E_\Pi$, alors considérer la règle d'appel

$$(n_1, loc) \rightarrow (e_p, loc_p) \cdot (n_2, loc)$$

où loc et loc_p sont comme précédemment. Et considérer de plus la règle de terminaison suivante :

$$(r_p, loc'_p) \rightarrow 0$$

Cette abstraction vers PAD que nous proposons est très utile puisque nous montrons dans les chapitres suivants qu'il est possible d'analyser ces systèmes de manière exacte en considérant toutes les équivalences structurelles entre les termes de porcessus.

4.4.5 Des programmes concurrents non récursifs vers les (P, P)-PRS

Ces programmes sont en général modélisés par des systèmes de réécriture de multiensembles ou de manière équivalente, des réseaux de Petri [BCR01, DBR02]. Les PFGs que nous avons considérés ne permettent pas de modéliser ces systèmes. En effet, dans le cas des programmes concurrents non récursifs, nous n'avons pas d'appels de procédures, c-à-d., nous n'avons pas les instructions $n_1 \xrightarrow{x:=call(p)} n_2$ ou

$$n_1 \xrightarrow{(x_1,\ldots,x_k):=call(p_1||\cdots||p_k)} n_2$$

qui introduisent la récursivité. En effet, ces instructions ont pour effet d'appeler les procédures $p_1, \ldots, p_k$ et d'attendre leur terminaison pour pouvoir continuer l'exécution à partir du point n_2. A la place, dans ces programmes, la création dynamique de processus est introduite par des instructions de la forme $n_1 \xrightarrow{spawn(p)} n_2$ qui expriment que n_1 lance le processus p et continue l'exécution à partir de n_2 indépendamment de p (c-à-d. sans attendre que p termine). Nous montrons dans cette section comment notre transformation peut être étendue pour modéliser cette classe de PFGs par un (P, P)-PRS (ou de manière équivalente, un système de réécriture de multiensembles ou un réseau de Petri).

Une instruction $n_1 \xrightarrow{spawn(p)} n_2$ peut être modélisée par la règle PRS suivante :

$$(n_1, loc) \rightarrow (e_p, loc_p)||(n_2, loc)$$

De même, comme dans ce cas nous n'avons pas de récursivité, l'opérateur "$\cdot$" est absent des termes de processus correspondant aux configurations du programme, et donc les instructions de synchronisation $n_1 \xrightarrow{a!} m_1$ et $n_2 \xrightarrow{a?} m_2$ peuvent toujours être modélisées de manière exacte par la règle

$$(n_1, loc_1)||(n_2, loc_2) \rightarrow (m_1, loc_1)||(m_2, loc_2)$$

Les instructions de base sont représentées de la même façon que précédemment, et la terminaison est représentée par des règles de la forme

$$(r_p, loc) \rightarrow 0$$

Avec cette traduction, nous obtenons un (P, P)-PRS modélisant le programme concurrent non récursif. Notre transformation coïncide avec la transformation bien connue qui consiste à modéliser une configuration du système par un multiensemble en comptant le nombre de processus qui sont dans chaque état (un état peut se voir comme une paire (n, loc)).

Observons que nous n'avons pas considéré ces instructions "*spawn*" dans notre modèle de PFG parce qu'en général, en présence d'appels récursifs, les PRS ne permettent pas de modéliser correctement la sémantique de ces primitives [6]. En effet, considérons l'exemple du PFG représenté à la figure 4.5, où a est une instruction de base et Π est une autre procédure qui n'est pas représentée. Alors le "vrai" programme peut appeler la procédure p, qui lance le processus Π et termine (indépendamment de la terminaison de Π), ce qui permet à l'instruction a d'être exécutée. Ce comportement ne peut pas être simulé par le PRS R de la figure 4.6 qui "aurait modélisé" ce programme. En effet, ce PRS permet les dérivations suivantes :

$$e_{\texttt{main}} \xrightarrow{R_1} e_p \cdot n \xrightarrow{R_3} (e_\Pi || r_p) \cdot n \xrightarrow{R_4} e_\Pi \cdot n$$

Dans le vrai programme, cette dérivation correspond successivement à l'appel de p, la création de Π, et la terminaison de p. Donc, à cette étape, le point n devrait être actif, et l'instruction a devrait pouvoir s'appliquer. Ceci n'est pas permis par nos règles PRS. En effet, dans le terme $e_\Pi \cdot n$, n ne devient actif qu'après la terminaison de Π, alors que dans le "vrai" programme, n est actif indépendamment de Π.

4.5 Conclusion

4.5.1 Résumé

Nous avons défini dans ce chapitre le modèle PRS introduit par Mayr [May98]. Nous avons donné une transformation automatisable qui permet de représenter un PFG par un PRS qui le sur-approxime. Nous avons défini une classe de programmes pour laquelle notre transformation est exacte. Ces programmes sont plus généraux que les programmes récursifs parallèles sans synchronisation. Nous avons réduit l'analyse statique des programmes récursifs parallèles à une analyse d'accessibilité dans ce

[6] Cette conclusion est le résultat d'une discussion que nous avons eu avec Markus Müller-Olm.

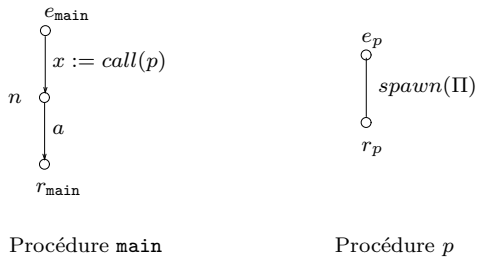


FIG. 4.5 – Un PFG avec l'instruction "*spawn*"

$$\begin{aligned} R_1 &= e_{\texttt{main}} \rightarrow e_p \cdot n \\ R_2 &= n \rightarrow r_{\texttt{main}} \\ R_3 &= e_p \rightarrow e_\Pi || r_p \\ R_4 &= r_p \rightarrow 0 \end{aligned}$$

FIG. 4.6 – Le PRS qui correspondrait au PFG de la figure 4.5

modèle. Cette analyse doit se faire dans le cas général modulo l'associativité du "$\cdot$" et l'associativité/commutativité du "$||$". Nous avons identifié une classe de programmes appelés les *programmes à parallélisme binaire* pour lesquels cette analyse d'accessibilité peut se faire sans considérer l'associativité/commutativité du "$||$". Ceci permet d'analyser ces programmes de manière exacte puisque nous donnons dans le chapitre suivant des algorithmes polynomiaux qui permettent de faire l'analyse d'accessibilité modulo $\sim_s$ de toute la classe PRS.

Nous avons également donné une traduction qui calcule un PAD qui abstrait le programme en oubliant la synchronisation et le retour de résultats des procédures parallèles appelées. Cette transformation est utile puisque, comme nous allons le voir dans les chapitres suivants, il est possible de calculer de manière exacte les ensembles des accessibles pour les PAD modulo toutes les équivalences.

4.5.2 Comparaison avec d'autres traductions

Nous avons montré que dans le cas des programmes séquentiels, les transformations que nous proposons coïncident avec celle de [ES01] qui modélise un programme séquentiel par un automate à pile ; et que dans le cas des programmes concurrents, elle coïncide avec la transformation de [BCR01, DBR02] qui utilise les réseaux de Petri pour modéliser ces programmes. De plus, l'abstraction vers un PAD que nous proposons est plus précise que celle vers les systèmes PA donnée dans [EK99, EP00] puisqu'elle permet de considérer les résultats de retour des appels récursifs non parallèles de manière exacte.

La modélisation des programmes récursifs parallèles par des PRS a indépendamment

été considérée dans [Esp02]. Dans cet article, la synchronisation est traitée de manière différente. En effet, nous partons d'un programme quelconque, et nous donnons des conditions *syntaxiques* suffisantes qui permettent de modéliser la synchronisation de manière exacte entre deux points du programme. Si ces conditions ne sont pas satisfaites, nous considérons des règles PRS qui permettent de sur-approximer l'effet de la synchronisation. Par contre, dans [Esp02], la traduction ne concerne qu'une classe de programmes définie par une restriction *sémantique* de la synchronisation qui introduit la notion de *signal local*. Cette restriction consiste à dire que deux instructions de la forme $a!$ et $a?$ ne peuvent communiquer que si a est un signal local de la même occurrence d'une procédure. Cette restriction impose que dans les programmes considérés, les synchronisations ne peuvent pas avoir lieu entre les points n_1 et n_2 dans un terme de la forme

$$\big((n_1, loc_1) \cdot (n', loc')\big) || (n_2, loc_2)$$

alors qu'avec notre modélisation, nous considérons ces programmes et nous proposons des modèles PRS qui les sur-approximent en permettant à chaque point d'avancer tout seul sans se synchroniser.

Chapitre 5

Analyse d'accessibilité des PRS par calcul de représentants

Dans ce chapitre, nous considérons le problème de calcul des ensembles des accessibles pour les systèmes PRS. Nous supposons sans perte de généralité que les PRS auxquels nous avons affaire sont sous formes normales (lemme 4.2.1). Puisque PRS est un modèle puissant qui inclut les réseaux de Petri, la construction précise de ses ensembles des accessibles est une tâche difficile, en particulier parce que les réseaux de Petri ne préservent pas la semilinéarité (les ensembles des marquages accessibles des réseaux de Petri ne sont en général pas semilinéaires).

Pour attaquer ce problème, nous avons tout d'abord besoin de pouvoir représenter de manière finie un ensemble potentiellement infini de termes de processus. Pour ce faire, nous représentons les termes de processus par des arbres binaires puisque l'ensemble des termes de processus $\mathcal{T}$ peut être vu comme $T_{\Sigma_0 \cup \Sigma_2}$ où $\Sigma_0 = \{0\} \cup Var$ et $\Sigma_2 = \{., ||\}$. Nous utilisons alors les automates finis d'arbres sur $\Sigma_0 \cup \Sigma_2$ pour représenter des ensembles réguliers de termes de processus. Notre objectif est donc de calculer $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$ pour un ensemble régulier de termes de processus $\mathcal{L}$. Seulement, comme "$\cdot$" est associatif et que "||" est associatif-commutatif, $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$ ne sont en général pas réguliers [GD89][1]. Pour contourner ce problème, nous adoptons une approche basée sur "l'oubli" de certaines équivalences entre les termes de processus. En effet, le lemme 4.4.3 implique qu'il suffit de considérer l'associativité du "$\cdot$" pour pouvoir analyser les programmes à parallélisme binaire. De plus, "oublier" certaines équivalences permet dans certains cas de calculer des représentants des ensembles des accessibles. C-à-d., des ensembles qui contiennent au moins un terme de

[1] Pour voir ceci, il suffit de considérer l'exemple suivant : Soit $\mathcal{L}$ le langage reconnu par l'automate fini (Q, Σ, F, δ) où $Q = \{q, q', q_X, q_Y\}$, $F = \{q\}$, $\Sigma_0 = \{X, Y\}$, $\Sigma_2 = \{\cdot\}$, et δ est l'ensemble des règles $X \to q_X$, $Y \to q_Y$, $\cdot(q_X, q_Y) \to q$, $\cdot(q, q_Y) \to q'$, et $\cdot(q_X, q') \to q$. Il est alors facile de voir que $[\mathcal{L}]_{\sim_s}$ est l'ensemble non régulier de tous les termes t ayant une frontière de la forme $X^n Y^n$: $[\mathcal{L}]_{\sim_s} = \{t \mid \exists n \in \mathbb{N}, Front(t) = X^n Y^n\}$.

chaque classe des termes accessibles (nous définissons cette notion de manière plus formelle dans la section suivante). Ceci permet l'analyse d'accessibilité pour des sous-classes de systèmes PRS. En effet, nous montrons dans la section suivante que calculer des représentants des accessibles est parfois suffisant pour résoudre les problèmes (2.10) et (2.11).

Cette idée est inspirée de [LS98, EP00], où les auteurs considèrent les systèmes PA et construisent les ensembles des accessibles de ces systèmes en oubliant l'associativité du "$\cdot$" et l'associativité/commutativité du "$||$" ; c-à-d. les ensembles $Post^*_{=}(\mathcal{L})$ et $Pre^*_{=}(\mathcal{L})$. Les ensembles obtenus sont alors des représentants de $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$. Dans le cas de toute la classe PRS, ceci n'est plus vrai, c-à-d. les ensembles $Post^*_{=}(\mathcal{L})$ et $Pre^*_{=}(\mathcal{L})$ ne sont en général pas des représentants des ensembles $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$. Ceci est dû au fait que dans PRS, il y a des règles de la forme $X||Y \rightarrow t$ (ou $X \cdot Y \rightarrow t$) qui ne sont pas présentes dans les systèmes PA, et qui font que l'on ne peut pas se restreindre à l'identité structurelle entre les termes lors des applications des règles. En effet, dans ce cas, la règle $X||Y \rightarrow t$ peut s'appliquer au terme $(X||Y)||Z$ mais pas au terme équivalent $(X||Z)||Y$.

Nous étendons cette approche aux systèmes PAD. Cette classe est plus générale que les systèmes à pile et les systèmes PA. L'intérêt principal de cette classe est qu'elle permet de modéliser les programmes parallèles où les procédures peuvent retourner leurs résultats aux procédures appelantes (ceci est possible avec les systèmes à pile, mais pas avec les systèmes PA). Nous montrons dans la première section de ce chapitre que pour PAD, pour calculer un représentant de $Post^*(\mathcal{L})$ ou de $Pre^*(\mathcal{L})$, il suffit de calculer un représentant de $Post^*_{\equiv}(\mathcal{L})$ ou de $Pre^*_{\equiv}(\mathcal{L})$ pour $\equiv \in \{=, \sim_0, \sim_s, \sim\}$. Le choix de l'équivalence $\equiv$ à considérer dépend des propriétés de fermeture du langage $\mathcal{L}$ par rapport à l'associativité de "$\cdot$" et à l'associativité/commutativité de "$||$".

Notre but dans ce chapitre sera alors de calculer des représentants de $Post^*_{\equiv}(\mathcal{L})$ ou de $Pre^*_{\equiv}(\mathcal{L})$ pour $\equiv \in \{=, \sim_0, \sim_s, \sim\}$. D'abord, nous montrons que pour chaque PRS, les langages réguliers d'arbres sont effectivement fermés par $Post^*_{=}$ et $Pre^*_{=}$. Nous donnons des constructions polynômiales des automates d'arbres correspondants. Nous montrons ensuite que ces constructions peuvent être adaptées au cas de l'équivalence $\sim_0$. Ces résultats généralisent ceux de [LS98] concernant la classe des systèmes PA. Ensuite, nous considérons le cas de l'équivalence $\sim_s$. Nous montrons que des représentants réguliers des ensembles des accessibles peuvent être construits en temps polynômial. Ceci permet l'analyse exacte des programmes à parallélisme binaire.

Dans le cas de l'équivalence $\sim$, nous nous restreignons à la classe des systèmes PAD. D'abord, nous montrons que les constructions précédentes pour les cas des équivalences $\sim_0$ et $\sim_s$ permettent de calculer en temps polynômial des représentants réguliers de l'ensemble des successeurs $Post^*(\mathcal{L})$. Pour l'ensemble des prédécesseurs, le problème est plus délicat à cause de l'opérateur $||$ qui apparaît dans les membres droits des règles. Dans le cas où l'ensemble initial est fermé par commutativité du $||$, nous montrons que nous pouvons comme précédemment utiliser les constructions pour les équivalences $\sim_0$ et $\sim_s$ pour calculer en temps polynômial un représentant régulier de l'ensemble des prédécesseurs. Si l'ensemble initial est quelconque, nous n'avons pas réussi à construire un représentant *régulier* de cet ensemble. Cependant, nous construisons un représentant *non régulier* caractérisé par un automate d'arbres à compteurs. Heureusement, nous montrons que la classe d'automates à compteurs que

nous obtenons par notre construction a de bonnes propriétés : le problème du vide de l'intersection de ces langages avec les langages réguliers est décidable, ce qui permet de résoudre le problème de vérification (2.11) auquel nous nous intéressons.

Les résultats de ce chapitre seront présentés dans [BT03b].

5.1 Réduire l'analyse d'accessibilité au calcul de représentants

Nous montrons dans cette section comment le calcul de représentants des ensembles des accessibles permet dans certains cas de résoudre les problèmes (2.10) et (2.11). Ensuite, nous montrons que pour les systèmes PAD calculer des représentants des ensembles $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$ peut se faire en "oubliant" certaines équivalences. Mais d'abord, nous définissons formellement la notion de représentants :

Définition 5.1.1 *Un ensemble de termes $\mathcal{L}$ est $\equiv$-compatible si $[\mathcal{L}]_\equiv = \mathcal{L}$. Un ensemble de termes $\mathcal{L}'$ est un $\equiv$-représentant de $\mathcal{L}$ si $[\mathcal{L}']_\equiv = \mathcal{L}$.*

Il est alors facile de voir que :

Proposition 5.1.1 *Soient $\mathcal{L}_1$ et $\mathcal{L}_2$ deux ensembles de termes de processus, et soit $\mathcal{L}'_1$ un $\sim$-représentant de l'ensemble $\mathcal{L}_1$. Si $\mathcal{L}_2$ est $\sim$-compatible, alors $\mathcal{L}_1 \cap \mathcal{L}_2 \neq \emptyset$ ssi $\mathcal{L}'_1 \cap \mathcal{L}_2 \neq \emptyset$.*

Par conséquent, calculer des $\sim$-représentants des ensembles $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$ permet de résoudre le problème d'accessibilité pour PRS. En effet, comme mentionné au chapitre 2, le problème se réduit à déterminer si $\mathcal{L}' \cap Reach \neq \emptyset$, où $Reach$ est égal à $Post^*(\mathcal{L})$ ou $Pre^*(\mathcal{L})$; $\mathcal{L}$ et $\mathcal{L}'$ étant des langages d'arbres représentant des termes de processus (problèmes 2.10 et 2.11). Et donc d'après la proposition précédente, si $\mathcal{L}'$ est $\sim$-compatible, il suffit de calculer un $\sim$-représentant de l'ensemble $Reach$.

Notre but est donc de calculer des $\sim$-représentants des ensembles $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$ pour un langage régulier de termes de processus $\mathcal{L}$. Il s'avère que pour certaines sous-classes de PRS, notamment la classe PAD, il suffit de calculer des $\equiv$-représentants des ensembles $Post^*_\equiv(\mathcal{L})$ et $Pre^*_\equiv(\mathcal{L})$, où $\equiv \in \{=, \sim_0, \sim_s, \sim\}$. Nous montrons dans la section suivante les liens qui existent entre ces différents ensembles.

5.2 Liens entre les $\equiv$-représentants

Comme dans cette section nous allons manipuler les différentes équivalences $\equiv$ en même temps, nous allons souvent utiliser les notations complètes $Post^*_{\mathcal{R},\equiv}(\mathcal{L})$ et $Pre^*_{\mathcal{R},\equiv}(\mathcal{L})$, et ce afin d'éviter tout risque de confusion.

De même, comme nous représentons les termes par des arbres binaires, nous notons dans tout le reste de ce chapitre un terme de processus $t_1 \cdot t_2$ par $\cdot(t_1, t_2)$, et $t_1||t_2$ par $||(t_1, t_2)$. Nous gardons les notations infixées pour les termes des membres gauches et droits des règles des systèmes PRS.

5.2.1 Les systèmes PA

Nous montrons que si $\mathcal{R}$ est un PA, alors $Post^*_{\mathcal{R},=}(\mathcal{L})$ est un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$, et si $\mathcal{L}$ est $\sim$-compatible alors $Pre^*_{\mathcal{R},=}(\mathcal{L})$ est égal à $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$. Ces correspondances ont été remarquées dans [LS98]. Intuitivement, le fait que $Post^*_{\mathcal{R},=}(\mathcal{L})$ soit un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$ est dû au fait que les règles d'un système PA sont de la forme $X \to t$, et donc les réécritures se font aux feuilles des termes, ce qui implique que si une règle peut s'appliquer à un terme u modulo $\sim$, alors cette règle peut aussi s'appliquer à ce même terme modulo l'identité, puisque les termes $\sim$-équivalents ont les mêmes feuilles (qui ne sont pas forcément ordonnées de la même manière). Le fait que si $\mathcal{L}$ est $\sim$-compatible alors $Pre^*_{\mathcal{R},=}(\mathcal{L})$ est égal à $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ découle du fait que $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},\sim}(\mathcal{L})$, et les règles de $\mathcal{R}^{-1}$ sont de la forme $t \to X$. L'application de ces règles entraîne la substitution d'un sous-terme par une variable, et n'introduit pas d'opérateurs supplémentaires "||" ou "$\cdot$". Donc, si le langage initial est $\sim$-compatible, il le reste après l'application de $\mathcal{R}^{-1}$.

Proposition 5.2.1 *Si $\mathcal{R}$ est un PA, alors $Post^*_{\mathcal{R},\sim}(\mathcal{L}) = [Post^*_{\mathcal{R},=}(\mathcal{L})]_\sim$, et si $\mathcal{L}$ est $\sim$-compatible alors $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = Pre^*_{\mathcal{R},=}(\mathcal{L})$.*

Preuve : Pour le premier point, il suffit de montrer que $Post^*_{\mathcal{R},\sim}(\mathcal{L}) \subseteq [Post^*_{\mathcal{R},=}(\mathcal{L})]_\sim$. Nous montrons par induction sur k que $Post^k_{\mathcal{R},\sim}(\mathcal{L}) \subseteq [Post^k_{\mathcal{R},=}(\mathcal{L})]_\sim$. Ceci est vrai pour $k = 0$. Considérons alors $k > 0$ et $t \in Post^k_\sim(\mathcal{L})$. Soient $t_0 \in Post^{k-1}_\sim(\mathcal{L})$, u, et u' tels que $u \sim t_0$, $u \to_\mathcal{R} u'$, et $u' \sim t$. Par induction, nous déduisons qu'il existe un terme $t_1 \sim t_0$ tel que $t_1 \in Post^{k-1}_=(\mathcal{L})$. Comme $\mathcal{R}$ est un PA, la règle qui s'est appliquée à u pour donner u' est de la forme $X \to s$. Donc, u est de la forme $C[X]$ pour un contexte C, et t_1 est aussi nécessairement de la forme $C'[X]$ avec $C[X] \sim C'[X]$. Donc $t_1 \to_\mathcal{R} C'[s]$, et $C'[s] \in Post^k_=(\mathcal{L})$. Puisque $t \sim C'[s]$, nous obtenons alors que $t \in [Post^*_{\mathcal{R},=}(\mathcal{L})]_\sim$.

Nous montrons maintenant que si $\mathcal{L}$ est $\sim$-compatible alors $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = Pre^*_{\mathcal{R},=}(\mathcal{L})$. Puisque $Pre^*_{\mathcal{R},\equiv}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},\equiv}(\mathcal{L})$, nous montrons par induction sur k que

$$Post^k_{\mathcal{R}^{-1},\sim}(\mathcal{L}) = Post^k_{\mathcal{R}^{-1},=}(\mathcal{L})$$

et que $Post^k_{\mathcal{R}^{-1},=}(\mathcal{L})$ est $\sim$-compatible. Ceci est vrai pour $k = 0$, soit alors $k > 0$. Puisque $Post^{k-1}_{\mathcal{R}^{-1},=}(\mathcal{L})$ est $\sim$-compatible, $Post^k_{\mathcal{R}^{-1},=}(\mathcal{L})$ l'est aussi puisque les règles de $\mathcal{R}^{-1}$ sont de la forme $s \to X$. Donc, il n'y a ni "$\cdot$" ni "||" qui sont créés, et le langage reste $\sim$-compatible.

Soit $t \in Post^k_{\mathcal{R}^{-1},\sim}(\mathcal{L})$. Soient alors $t_0 \in Post^{k-1}_{\mathcal{R}^{-1},\sim}(\mathcal{L})$, u, et u' tels que $u \sim t_0$, $u \to_{\mathcal{R}^{-1}} u'$, et $u' \sim t$. Par induction, nous en déduisons qu'il existe un terme $t_1 \sim t_0$ tel que $t_1 \in Post^{k-1}_{\mathcal{R}^{-1},=}(\mathcal{L})$, et donc puisque $Post^{k-1}_{\mathcal{R}^{-1},=}(\mathcal{L})$ est $\sim$-compatible et que $t_1 \sim u$, $u \in Post^{k-1}_{\mathcal{R}^{-1},=}(\mathcal{L})$ et donc $u' \in Post^k_{\mathcal{R}^{-1},=}(\mathcal{L})$, et donc $t \in Post^k_{\mathcal{R}^{-1},=}(\mathcal{L})$ puisque $Post^k_{\mathcal{R}^{-1},=}(\mathcal{L})$ est $\sim$-compatible. □

5.2.2 Les systèmes PAD

5.2.2.1 Liens entre les $\sim_s$-représentants et les $\sim$-représentants

Nous montrons que si $\mathcal{R}$ est un PAD, alors un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$ est un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$, et si $\mathcal{L}$ est $\sim_{||}$-compatible alors un $\sim_s$-représentant de $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})$ est un $\sim$-représentant de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$. Ceci est dû au fait que :

- Les PAD ne contiennent pas de règles de la forme $X||Y \to t$ et donc si une règle peut s'appliquer modulo $\sim$, elle peut aussi s'appliquer modulo $\sim_s$.
- $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},\sim}(\mathcal{L})$, et si $\mathcal{R}$ est un PAD, les règles de $\mathcal{R}^{-1}$ ne rajoutent pas de "||" dans les termes. Ainsi, si le langage initial est $\sim_{||}$-compatible, cette propriété est maintenue après l'application des règles de $\mathcal{R}^{-1}$.

Proposition 5.2.2 *Si $\mathcal{R}$ est un PAD, alors $Post^*_{\mathcal{R},\sim}(\mathcal{L}) = [Post^*_{\mathcal{R},\sim_s}(\mathcal{L})]_\sim$, et si $\mathcal{L}$ est $\sim_{||}$-compatible alors $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = [Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})]_\sim$.*

Preuve : Il est clair que $[Post^*_{\sim_s}(\mathcal{L})]_\sim \subseteq Post^*_\sim(\mathcal{L})$. L'autre inclusion est dûe au fait que les PAD ne contiennent pas de règles de la forme $X||Y \to t$. Nous montrons par induction sur k que $Post^k_\sim(\mathcal{L}) \subseteq [Post^k_{\sim_s}(\mathcal{L})]_\sim$. Ceci est vrai pour $k = 0$. Soit alors $k > 0$ et $t \in Post^k_\sim(\mathcal{L})$. Soient $t_0 \in Post^{k-1}_\sim(\mathcal{L})$, u, et u' tels que $u \sim t_0$, $u \to_\mathcal{R} u'$, et $u' \sim t$. Par induction, nous en déduisons qu'il existe un terme $t_1 \sim t_0$ tel que $t_1 \in Post^{k-1}_{\sim_s}(\mathcal{L})$. Comme $\mathcal{R}$ est un PAD, la règle qui s'est appliquée à u pour donner u' est de la forme $v_1 \to v_2$ où v_1 est de la forme X, 0, ou $X \cdot Y$. Nous considérons dans ce qui suit le cas (le plus compliqué) où la règle appliquée est de la forme $X \cdot Y \to s$, les autres cas se traitent de la même manière que dans la preuve précédente. Soit alors un contexte C tel que $u = C\big[\cdot(X,Y)\big]$ et $\cdot(X,Y)$ peut se réécrire dans le contexte C. Donc $t_1 \sim C\big[\cdot(X,Y)\big]$. Il existe alors un contexte $C'[x]$ tel que $C[x] \sim C'[x]$, la position x est active dans $C'[x]$ (c-à-d., elle peut se réécrire), et $t_1 \sim_s C'\big[\cdot(X,Y)\big]$. Ceci se montre facilement par induction sur le nombre de fois les axiomes A1–A4 sont utilisés pour obtenir $C\big[\cdot(X,Y)\big]$ à partir de t_1. Plus précisément, si $t_1 \sim v$ et n est le nombre de fois les axiomes A1–A4 sont utilisés pour obtenir v à partir de t_1, il s'agit de montrer par induction sur n que :

- si v est de la forme $C\big[\cdot(X,Y)\big]$, alors il existe un contexte $C'[x]$ tel que $C[x] \sim C'[x]$, la position x est active dans $C'[x]$, et $t_1 \sim_s C'\big[\cdot(X,Y)\big]$;
- si v est de la forme $C\big[\cdot\big(X,(Y,v')\big)\big]$, alors il existe un terme $v'' \sim v'$ et un contexte $C'[x]$ tels que $C[x] \sim C'[x]$, la position x est active dans $C'[x]$, et $t_1 \sim_s C'\big[\cdot\big((X,Y),v''\big)\big]$;
- si v est de la forme $C\big[\cdot\big(X,(0,Y)\big)\big]$, alors il existe un contexte $C'[x]$ tel que $C[x] \sim C'[x]$, la position x est active dans $C'[x]$, et $t_1 \sim_s C'\big[\cdot(X,Y)\big]$.

Donc, $t_1 \sim_s C'\big[\cdot(X,Y)\big]$ tel que $\cdot(X,Y)$ peut être réécrit dans le contexte C'. Nous aurons alors $C'[s] \in Post^k_{\sim_s}(\mathcal{L})$ et $C'[s] \sim u'$ puisque $u' = C[s]$ et $C[x] \sim C'[x]$. Donc $t \in [Post^k_{\sim_s}(\mathcal{L})]_\sim$.

Montrons maintenant que si $\mathcal{L}$ est $\sim_{||}$-compatible alors $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = [Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})]_\sim$. Pour ce faire, nous montrons que $Post^*_{\mathcal{R}^{-1},\sim}(\mathcal{L}) = [Post^*_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})]_\sim$. Nous avons déjà que $[Post^*_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})]_\sim \subseteq Post^*_{\mathcal{R}^{-1},\sim}(\mathcal{L})$. Pour l'autre inclusion, nous montrons par induction sur k que si $\mathcal{L}$ est $\sim_{||}$-compatible alors $Post^k_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$ est aussi $\sim_{||}$-compatible

et $Post^k_{\mathcal{R}^{-1},\sim}(\mathcal{L}) \subseteq [Post^k_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})]_\sim$. La propriété est vraie pour $k = 0$, soit alors $k > 0$. Puisque $Post^{k-1}_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$ est $\sim_{||}$-compatible, $Post^k_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$ l'est aussi puisque les règles de $\mathcal{R}^{-1}$ sont de la forme $v_1 \to v_2$ où v_2 est de la forme X, 0, ou $X \cdot Y$. Donc, il n'y a pas de "||" qui est créé, et le langage reste $\sim_{||}$-compatible.

Soient $t_0 \in Post^{k-1}_{\mathcal{R}^{-1},\sim}(\mathcal{L})$, u, et u' tels que $u \sim t_0$, $u \to_{\mathcal{R}^{-1}} u'$, et $u' \sim t$. Par induction, nous déduisons qu'il existe un terme $t_1 \sim t_0$ tel que $t_1 \in Post^{k-1}_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$. Si la règle appliquée n'est pas de la forme $X||Y \to s$, nous procédons comme précédemment. Sinon, il existe un contexte C tel que $u = C[||(X, Y)]$, $u' = C[s]$, et $t_1 \sim C[||(X, Y)]$. Comme l'application des axiomes A2 est indépendante de l'application des axiomes A3-A4 ; il existe un terme t_2 tel que $t_2 \sim_{||} t_1$ et $t_2 \sim_s C[||(X, Y)]$. Comme $Post^{k-1}_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$ est $\sim_{||}$-compatible, $t_2 \in Post^{k-1}_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$, et $C[s] \in Post^k_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})$. Donc

$$t \in [Post^k_{\mathcal{R}^{-1},\sim_s}(\mathcal{L})]_\sim$$

□

Nous en déduisons alors que :

Proposition 5.2.3 *Si $\mathcal{R}$ est un PAD et ρ un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$, alors ρ un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$. De plus, si $\mathcal{L}$ est $\sim_{||}$-compatible et ρ' un $\sim_s$-représentant de $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})$, alors ρ' est un $\sim$-représentant de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$.*

Preuve : Ceci est une conséquence immédiate de la proposition précédente : Soit ρ un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$, donc $[\rho]_\sim = \left[[\rho]_{\sim_s}\right]_\sim = [Post^*_{\mathcal{R},\sim_s}(\mathcal{L})]_\sim = Post^*_{\mathcal{R},\sim}(\mathcal{L})$. Le cas de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ est analogue. □

Par conséquent, pour résoudre le problème d'accessibilité pour un PAD $\mathcal{R}$, il suffit de calculer un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$ et, si $\mathcal{L}$ est $\sim_{||}$-compatible, un $\sim_s$-représentant de $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})$.

5.2.2.2 Liens entre les $\sim_s$-représentants et les $\sim_0$-représentants

Nous montrons dans ce qui suit qu'étant donné un PRS quelconque $\mathcal{R}$, si $\mathcal{L}$ est $\sim_s$-compatible alors $Post^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$ et $Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est un $\sim_s$-représentant de $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})$ (Corollaire 5.2.1). Plus précisément, nous définissons la classe des langages Peignés (en particulier, un langage $\sim_s$-compatible est Peigné) et nous montrons que si $\mathcal{L}$ est Peigné, alors pour tout PRS $\mathcal{R}$, $Post^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$, et $Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est un $\sim_s$-représentant de $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})$ (Théorème 5.2.1). Nous en déduisons que si $\mathcal{R}$ est un PAD et $\mathcal{L}$ est $\sim_s$-compatible, alors $Post^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$. De plus, si $\mathcal{L}$ est $\sim$-compatible alors $Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est un $\sim$-représentant de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$.

Nous définissons d'abord la notion de langages "Peignés". Soit t un terme de processus, $simp_0(t)$ est le terme obtenu en simplifiant tous les 0 de t :

Définition 5.2.1 *Soit t un terme de $\mathcal{T}$. Nous définissons le terme $simp_0(t)$ inductivement par :*

- $simp_0(X) = X$ *pour* $X \in Var$,
- $simp_0(0) = 0$,
- *soit* $f \in \{\cdot, ||\}$, $simp_0\big(f(t_1, t_2)\big)$ *est égal à :*
  - $f\big(simp_0(t_1), simp_0(t_2)\big)$ *si* $t_1 \neq 0$ *et* $t_2 \neq 0$,
  - $simp_0(t_1)$ *si* $t_2 = 0$,
  - $simp_0(t_2)$ *si* $t_1 = 0$,
  - 0 *si* $t_1 = t_2 = 0$,

Cette définition est étendue aux langages de termes de manière standard.

Etant donné un terme t sans 0, c-à-d. dont toutes les feuilles sont différentes de 0, $Peigne(t)$ est défini comme le terme $\sim_s$-équivalent à t tel que la racine du fils droit immédiat d'un nœud étiqueté par "$\cdot$" est différente de "$\cdot$" :

Définition 5.2.2 *Soit t un terme de $\mathcal{T}$ dont toutes les feuilles sont différentes de 0. Nous définissons inductivement le terme $Peigne(t)$ comme suit :*
- $Peigne(X) = X$, *pour* $X \in Var$,
- $Peigne\big(||(t_1, t_2)\big) = ||\big(Peigne(t_1), Peigne(t_2)\big)$,
- $Peigne\big(\cdot\,(t_1, t_2)\big)$ *est égal à :*
  - $\cdot\big(Peigne(t_1), Peigne(t_2)\big)$ *si* $racine(t_2) \neq$ "$\cdot$",
  - $Peigne\Big(\cdot\,\big(\cdot\,(t_1, v_1), v_2\big)\Big)$ *si* $t_2 = \cdot(v_1, v_2)$.

Nous définissons de manière standard $Peigne(\mathcal{L})$ pour un langage de termes $\mathcal{L}$. Un ensemble de termes $\mathcal{L}$ est Peigné si

$$Peigne\big(simp_0(\mathcal{L})\big) \subseteq \mathcal{L}.$$

Le terme $Peigne\big(simp_0(t)\big)$ peut se voir comme étant une forme normale pour tous les termes qui sont $\sim_s$-équivalents à t :

Proposition 5.2.4 *Soient t et t' deux termes de $\mathcal{T}$, alors*
- $Peigne\big(simp_0(t)\big) \sim_s t$,
- $t \sim_s t'$ *ssi* $Peigne\big(simp_0(t)\big) = Peigne\big(simp_0(t')\big)$.

Preuve : Pour le premier point, puisque $simp_0(t) \sim_0 t$, il suffit de montrer que si $t_0 = simp_0(t)$, alors $Peigne(t_0) \sim_s t_0$. Ceci se montre facilement par induction structurelle sur t_0. Pour le deuxième point, l'implication droite-gauche se déduit du point précédent. Pour l'autre direction, soient $t \sim_s t'$. Il est facile de voir que $simp_0(t) \sim_s simp_0(t')$ puisque $simp_0(t) \sim_0 t$ et $simp_0(t') \sim_0 t'$. Il suffit alors de montrer que si u et u' sont deux termes dont toutes les feuilles sont différentes de 0 et tels que $u \sim_s u'$, alors $Peigne(u) = Peigne(u')$. Ceci se montre facilement par induction structurelle sur u.

□

Par exemple, la figure 5.2.2.2 montre deux termes $\sim_s$-équivalents et leur Peigne.

Nous montrons la proposition suivante qui nous sera utile pour la preuve du théorème 5.2.1.

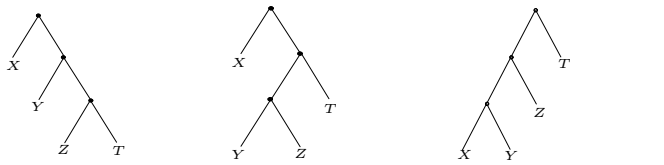


FIG. 5.1 – Exemple d'un Peigne de deux termes $\sim_s$-équivalents

Proposition 5.2.5 *Soient $u, u' \in \mathcal{T}$ et $a \in Act$ tels que $u \to_{\mathcal{R}} u'$. Alors, il existe $v, v' \in \mathcal{T}$ tels que $Peigne\big(simp_0(u)\big) \sim_0 v$, $v \to_{\mathcal{R}} v'$, et $v' \sim_0 Peigne\big(simp_0(u')\big)$.*

Preuve : Ceci se montre facilement par induction structurelle sur u, en considérant à chaque fois les différents types de règles de $\mathcal{R}$ qui peuvent s'appliquer à u pour donner u' (règle de type $X||Y \to Z||T$, ou $X \cdot Y \to Z \cdot T$, ou $X \cdot Y \to Z||T$, etc.).

□

Nous obtenons alors le résultat suivant basé sur le fait que si $\mathcal{L}$ est Peigné, il le reste après l'application d'une règle PRS modulo $\sim_0$:

Théorème 5.2.1 *Soit $\mathcal{R}$ un PRS, si $\mathcal{L}$ est Peigné, alors $Post^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [Post^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim_s}$ et $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim_s}$.*

Preuve : Nous montrons que $Post^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [Post^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim_s}$. L'autre point découle alors du fait que $Pre^*_{\mathcal{R},\equiv}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},\equiv}(\mathcal{L})$. Puisqu'il n'y a aucune confusion concernant $\mathcal{R}$, nous allons omettre l'indice $\mathcal{R}$. Il est clair que $[Post^*_{\sim_0}(\mathcal{L})]_{\sim_s} \subseteq Post^*_{\sim_s}(\mathcal{L})$. Pour l'autre direction, nous montrons par induction sur k que si $\mathcal{L}$ est Peigné, alors $Post^k_{\sim_0}(\mathcal{L})$ est Peigné et $Post^k_{\sim_s}(\mathcal{L}) \subseteq [Post^k_{\sim_0}(\mathcal{L})]_{\sim_s}$. La propriété est vraie pour $k = 0$. Soit alors $k > 0$ et $t \in Post^k_{\sim_s}(\mathcal{L})$. Soient donc $t_0 \in Post^{k-1}_{\sim_s}(\mathcal{L})$ et des termes u et u' tels que $u \sim_s t_0$, $u \to_{\mathcal{R}} u'$, et $u' \sim_s t$. Par induction, nous déduisons qu'il existe un terme $t_1 \in Post^{k-1}_{\sim_0}(\mathcal{L})$ tel que $t_1 \sim_s t_0$, et donc que $Peigne\big(simp_0(u)\big) \in Post^{k-1}_{\sim_0}(\mathcal{L})$. En effet, $t_1 \in Post^{k-1}_{\sim_0}(\mathcal{L})$ et $Post^{k-1}_{\sim_0}(\mathcal{L})$ est Peigné, et donc $Peigne\big(simp_0(t_1)\big) \in Post^{k-1}_{\sim_0}(\mathcal{L})$. En plus, $t_0 \sim_s u \sim_s t_1$, et donc $Peigne\big(simp_0(u)\big) = Peigne\big(simp_0(t_1)\big)$ (Proposition 5.2.4). La proposition 5.2.5 implique alors qu'il existe $v, v' \in \mathcal{T}$ tels que $Peigne\big(simp_0(u)\big) \sim_0 v$, $v \to_{\mathcal{R}} v'$, et $v' \sim_0 Peigne\big(simp_0(u')\big)$. Ce qui veut dire que $Peigne\big(simp_0(u)\big) \Rightarrow_{\sim_0} Peigne\big(simp_0(u')\big)$. Ceci implique que $Peigne\big(simp_0(t)\big) \in Post^k_{\sim_0}(\mathcal{L})$ puisque $Peigne\big(simp_0(u)\big) \in Post^{k-1}_{\sim_0}(\mathcal{L})$ et

$$Peigne\big(simp_0(u')\big) = Peigne\big(simp_0(t)\big)$$

car $t \sim_s u'$ (Proposition 5.2.4). Donc, $t \in [Post^k_{\sim_0}(\mathcal{L})]_{\sim_s}$ puisque $t \sim_s Peigne\big(simp_0(t)\big)$.

Nous montrons maintenant que $Post^k_{\sim_0}(\mathcal{L})$ est Peigné. Soit $t \in Post^k_{\sim_0}(\mathcal{L})$, montrons que $Peigne\big(simp_0(t)\big) \in Post^k_{\sim_0}(\mathcal{L})$. Soient alors $t_1 \in Post^{k-1}_{\sim_0}(\mathcal{L})$, u et u' tels que $u \sim_0 t_1$, $u \to_{\mathcal{R}} u'$, et $u' \sim_0 t$. Puisque $Post^{k-1}_{\sim_0}(\mathcal{L})$ est Peigné, $Peigne\big(simp_0(u)\big) \in Post^{k-1}_{\sim_0}(\mathcal{L})$ et la proposition 5.2.5 implique que $Peigne\big(simp_0(u')\big) \in Post^k_{\sim_0}(\mathcal{L})$.

Donc, $Peigne\big(simp_0(t)\big) \in Post^k_{\sim_0}(\mathcal{L})$ puisque $Peigne\big(simp_0(u')\big) = Peigne\big(simp_0(t)\big)$ car $t \sim_0 u'$. □

Comme un langage $\sim_s$-compatible est Peigné (proposition 5.2.4), nous obtenons le corollaire suivant :

Corollaire 5.2.1 *Soit $\mathcal{R}$ un PRS, si $\mathcal{L}$ est $\sim_s$-compatible, alors $Post^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [Post^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim_s}$ et $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim_s}$.*

Nous obtenons donc grâce à la proposition 5.2.2 que :

Proposition 5.2.6 *Soit $\mathcal{R}$ un PAD, si $\mathcal{L}$ est $\sim_s$-compatible, alors $Post^*_{\mathcal{R},\sim}(\mathcal{L}) = [Post^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim}$, et si $\mathcal{L}$ est $\sim$-compatible alors $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = [Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})]_{\sim}$.*

5.2.2.3 Conclusion

Il découle des propositions précédentes que pour calculer un $\sim$-représentant de $Post^*(\mathcal{L})$ pour les systèmes PAD, il suffit de calculer un $\sim_s$-représentant de $Post^*_{\sim_s}(\mathcal{L})$. Ceci se réduit au calcul de $Post^*_{\sim_0}(\mathcal{L})$ si $\mathcal{L}$ est $\sim_s$-compatible.

De la même manière, pour calculer un $\sim$-représentant de $Pre^*(\mathcal{L})$ pour PAD, il suffit de calculer un $\sim_s$-représentant de $Pre^*_{\sim_s}(\mathcal{L})$ si $\mathcal{L}$ est $\sim_{||}$-compatible. Si de plus $\mathcal{L}$ est $\sim_s$-compatible (c-à-d., $\mathcal{L}$ est $\sim$-compatible), ceci se ramène au calcul de $Pre^*_{\sim_0}(\mathcal{L})$.

Nous donnons dans le reste de ce chapitre des constructions polynômiales qui caractérisent des langages réguliers correspondant à ces ensembles. Si $\mathcal{L}$ ne vérifie pas ces conditions de compatibilité, nous sommes obligés de raisonner modulo toutes les équivalences $\sim$. Nous nous attaquons à ce problème dans la dernière partie de ce chapitre.

5.3 Accessibilité sans équivalences

Nous montrons dans cette section que pour tout langage régulier $\mathcal{L}$, $Post^*_{\mathcal{R},=}(\mathcal{L})$ et $Pre^*_{\mathcal{R},=}(\mathcal{L})$ sont effectivement réguliers.

Nous notons par $Sub_r(\mathcal{R})$ l'ensemble de tous les sous termes des membres droits des règles de $\mathcal{R}$. Soit $Q_\mathcal{R} = \{q_t \mid t \in Sub_r(\mathcal{R})\}$, et soit $\delta_\mathcal{R}$ l'ensemble suivant de règles de transition :

- $X \to q_X$ pour chaque $X \in Sub_r(\mathcal{R})$,
- $0 \to q_0$ si $0 \in Sub_r(\mathcal{R})$,
- $||(q_X, q_Y) \to q_{X||Y}$ si $X||Y \in Sub_r(\mathcal{R})$,
- $\cdot(q_X, q_Y) \to q_{X\cdot Y}$ si $X \cdot Y \in Sub_r(\mathcal{R})$.

Il est clair qu'avec $\delta_\mathcal{R}$, pour chaque $t \in Sub_r(\mathcal{R})$, l'état q_t accepte $\{t\}$ ($L_{q_t} = \{t\}$). Soit $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini. Nous définissons l'automate $\mathcal{A}^*_\mathcal{R} = (Q', \Sigma, F', \delta')$ comme suit :

- $Q' = \{q, q^{nil}, q^T \mid q \in Q \cup Q_\mathcal{R}\}$. Nous dénotons par $\tilde{q}$ tout état dans $\{q, q^T, q^{nil}\}$.
- $F' = \{q, q^{nil}, q^T \mid q \in F\}$,
- δ' est le plus petit ensemble de règles contenant $\delta \cup \delta_\mathcal{R}$ et tel que pour chaque $q_1, q_2, q \in Q \cup Q_\mathcal{R}$:

1. $q \to q^T \in \delta'$ pour chaque $q \in Q \cup Q_{\mathcal{R}}$,
2. si $0 \xrightarrow{*}_{\delta} q$, alors $0 \to q^{nil} \in \delta'$,
3. si $t_1 \to t_2 \in \mathcal{R}$, et il existe un état $q \in Q \cup Q_{\mathcal{R}}$ tel que $t_1 \xrightarrow{*}_{\delta'} q^T$, alors :
   (a) $q^T_{t_2} \to q^T \in \delta'$,
   (b) $q^{nil}_{t_2} \to q^{nil} \in \delta'$, et
   (c) $q^T_{t_2} \to q^{nil} \in \delta'$ si $t_1 = 0$,
4. si $\cdot(q_1, q_2) \to q \in \delta \cup \delta_{\mathcal{R}}$, alors :
   (a) $\cdot(q_1^{nil}, \tilde{q}_2) \to q^T \in \delta'$,
   (b) $\cdot(q_1^{nil}, q_2^{nil}) \to q^{nil} \in \delta'$,
   (c) $\cdot(q_1^T, q_2) \to q^T \in \delta'$,
5. si $||(q_1, q_2) \to q \in \delta \cup \delta_{\mathcal{R}}$, alors :
   (a) $||(q_1^{nil}, q_2^{nil}) \to q^{nil} \in \delta'$,
   (b) $||(\tilde{q}_1, \tilde{q}_2) \to q^T \in \delta'$,
6. si $q \to q' \in \delta$, alors $q^T \to q'^T \in \delta'$, et $q^{nil} \to q'^{nil} \in \delta'$.

δ' est définie inductivement et peut être calculée comme la limite d'une séquence finie d'ensembles croissants de transitions $\delta'_1 \subset \delta'_2 \subset \ldots \subset \delta'_n$, où δ'_{i+1} contient au plus trois transitions en plus que δ'_i. Ces transitions sont rajoutées par les règles d'inference (3). Cette procédure termine parcequ'il y a un nombre fini d'états dans $Q \cup Q_{\mathcal{R}}$.

Remarque 5.3.1 *Si $\mathcal{A}$ a k états et τ transitions, alors $\mathcal{A}^*_{\mathcal{R}}$ a $\mathcal{O}(k|Sub_r(\mathcal{R})|+|Sub_r(\mathcal{R})|^2+\tau)$ transitions et $3(k+|Sub_r(\mathcal{R})|)$ états.*

Théorème 5.3.1 *Soit $\mathcal{L}$ un ensemble régulier de termes de processus, et $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini qui reconnaît $\mathcal{L}$. Alors, $Post^*_{\mathcal{R},=}(\mathcal{L})$ et $Pre^*_{\mathcal{R},=}(\mathcal{L})$ sont reconnus par l'automate d'arbres fini $\mathcal{A}^*_{\mathcal{R}}$ et $\mathcal{A}^*_{\mathcal{R}^{-1}}$, respectivement.*

Nous montrons dans ce qui suit que $Post^*_{\mathcal{R},=}(\mathcal{L})$ est accepté par $\mathcal{A}^*_{\mathcal{R}}$. nous en déduisons que $\mathcal{A}^*_{\mathcal{R}^{-1}}$ accepte $Pre^*_{\mathcal{R},=}(\mathcal{L})$ puisque $Pre^*_{\mathcal{R},=}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},=}(\mathcal{L})$. Intuitivement, pour chaque $q \in Q \cup Q_{\mathcal{R}}$, l'état q^T accepte les successeurs de L_q ($L_{q^T} = Post^*_{\mathcal{R},=}(L_q)$) et l'état q^{nil} accepte les successeurs u de L_q qui ont été obtenus à partir des successeurs nuls de L_q : $L_{q^{nil}} = Post^*_{\mathcal{R},=}\big(Post^*_{\mathcal{R},=}(L_q) \cap \{u \in \mathcal{T} \mid u \sim_0 0\}\big)$. En particulier, ceci veut dire que pour chaque $t \in Sub_r(\mathcal{R})$, $L_{q_t^T} = Post^*_{\mathcal{R},=}(t)$, et $L_{q_t^{nil}} = Post^*_{\mathcal{R},=}\big(Post^*_{\mathcal{R},=}(t) \cap \{u \in \mathcal{T} \mid u \sim_0 0\}\big)$. Les règles (1) expriment que $L_q \subseteq Post^*_{\mathcal{R},=}(L_q)$. Les règles (2) marquent les feuilles nulles par l'exposant ^{nil}, indiquant qu'elles sont nulles. Les règles (3) expriment que si un terme t dans $Sub_r(\mathcal{R})$ est un successeur d'un terme dans L_q, alors tous les successeurs de t le sont aussi. Les nœuds étiquetés par "||" sont annotés par les règles (5). Par exemple, l'intuition exprimée par les règles (5b) est que si $u_1 \in Post^*_{\mathcal{R},=}(L_{q_1})$ et $u_2 \in Post^*_{\mathcal{R},=}(L_{q_2})$, alors $||(u_1, u_2) \in Post^*_{\mathcal{R},=}(L_q)$ si $||(q_1, q_2) \to q \in \delta \cup \delta_{\mathcal{R}}$. Les règles (4) annotent les nœuds étiquetés par "." en respectant la sémantique du ".". Les états q et q^{nil} jouent un rôle important pour ces règles. En effet, les règles (4a) et (4c) assurent que le fils droit d'un

nœud étiqueté par "." ne peut pas être réécrit si le fils gauche n'a pas été nul, c-à-d., si le fils gauche n'est pas annoté par un état q^{nil}. Finalement, les règles (6) expriment que si $L_q \subseteq L_{q'}$, alors $Post^*_{\mathcal{R},=}(L_q) \subseteq Post^*_{\mathcal{R},=}(L_{q'})$.

Plus formellement, pour montrer le théorème 7.6.1, nous montrons les lemmes 5.3.1 et 5.3.2.

Lemme 5.3.1 *Pour chaque $v \in \mathcal{T}$ et chaque $q \in Q \cup Q_{\mathcal{R}}$, nous avons :*
- $v \xrightarrow{*}_{\delta'} q^T \Rightarrow v \in Post^*_=(L_q)$,
- $v \xrightarrow{*}_{\delta'} q^{nil} \Rightarrow \exists u \sim_0 0 \mid v \in Post^*_=(u)$, *et* $u \in Post^*_=(L_q)$.

Preuve : Nous montrons par induction sur i que pour chaque $v \in \mathcal{T}$ et chaque $q \in Q \cup Q_{\mathcal{R}}$, nous avons :
- $v \xrightarrow{*}_{\delta'_i} q^T \Rightarrow v \in Post^*_=(L_q)$,
- $v \xrightarrow{*}_{\delta'_i} q^{nil} \Rightarrow \exists u \sim_0 0 \mid v \in Post^*_=(u)$, and $u \in Post^*_=(L_q)$.

Ces points doivent être montrés simultanément :
- $i = 0$, alors δ'_0 contient les règles δ, $\delta_{\mathcal{R}}$, en plus des règles (1), (2), (4), (5), et (6). Donc, il est facile de voir que $v \xrightarrow{*}_{\delta'_0} q^T$ implique que $v \in L_q$, et $v \xrightarrow{*}_{\delta'_0} q^{nil}$ implique que $v \sim_0 0$ et $v \in L_q$,
- $i > 0$. Soit $v \xrightarrow{*}_{\delta'_i} q^T$ une exécution de δ'_i sur le terme v (le cas où nous avons $v \xrightarrow{*}_{\delta'_i} q^{nil}$ peut être résolu de la même manière). Soit n ne nombre de transitions dans $\delta'_i \setminus \delta'_{i-1}$ appliquées pendant cette exécution. Dans ce cas, nous dénotons l'exécution précédente par $v \xrightarrow{*}{}^{n}_{\delta'_i} q^T$. Nous procédons par induction sur n :
  - $n = 0$, alors seules les règles de δ'_{i-1} ont été appliquées, et nous obtenons le résultat par induction sur i ;
  - $n > 0$. Il y a deux cas :
    1. Il existe un contexte C et un terme $v_1 \in \mathcal{T}$ tel que $v = C[v_1]$, et

$$v = C[v_1] \xrightarrow{*}{}^{n-1}_{\delta'_i} C[s'] \rightarrow_{\delta'_i} C[s] \xrightarrow{k}_{\delta'_{i-1}} q^T \tag{5.1}$$

Alors, les règles (3) impliquent qu'il existe nécessairement un terme t_1 dans $Sub_r(\mathcal{R})$, et un état $p \in Q \cup Q_{\mathcal{R}}$ tels que $s' = q^T_{t_1}$, et $s = p^T$.
Par conséquent, nous obtenons par induction sur n que $v_1 \in Post^*_=(L_{q_{t_1}})$, c-à-d. puisque $L_{q_{t_1}} = t_1$:

$$v_1 \in Post^*_=(t_1) \tag{5.2}$$

Le fait que $q^T_{t_1} \rightarrow_{\delta'_i} p^T$ implique qu'il y a un terme u tel que $u \xrightarrow{*}_{\delta'_{i-1}} p^T$, et $u \rightarrow t_1$ est une règle de $\mathcal{R}$ (règles (3)). Nous déduisons alors par induction sur i que $u \in Post^*_=(L_p)$, et donc

$$t_1 \in Post^*_=(L_p) \tag{5.3}$$

Ainsi, (5.2) et (5.3) impliquent que

$$v_1 \in Post^*_=(L_p)$$

Nous montrons maintenant par induction sur k que $v \in Post^*_=(L_q)$:

– $k = 0$. Alors la propriété est satisfaite puisque nous avons nécessairement que $v = v_1$ et $p = q$.
– $k > 0$. Il y a trois cas :

(a) $v \xrightarrow{*}_{\delta'_i} C[p^T] \xrightarrow{k-1}_{\delta'_{i-1}} {q'}^T \rightarrow_{\delta'_{i-1}} q^T$. Alors nous obtenons par induction que $v \in Post^*_=(L_{q'})$. Ensuite, il y a deux cas : (1) soit il existe un terme t_2 dans $Sub_r(\mathcal{R})$ tel que $q' = q_{t_2}$, et donc la règle ${q'}^T \rightarrow_{\delta'_{i-1}} q^T$ a été créée par les règles d'inférence (3). Dans ce cas, nous obtenons facilement que $v \in Post^*_=(L_q)$. (2) Ou $q' \in Q$, auquel cas, les règles (6) impliquent que la règle $q' \rightarrow q$ existe nécessairement dans δ, ce qui veut dire que $L_{q'} \subseteq L_q$, et donc que

$$v \in Post^*_=(L_{q'}) \subseteq Post^*_=(L_q)$$

(b) Il existe deux termes v_1, et v_2, et un contexte C' tels que

$$v = \cdot(v_1, v_2) \xrightarrow{*}_{\delta'_i} \cdot(C'[p^T], v_2) \xrightarrow{k-1}_{\delta'_{i-1}} \cdot(l_1, l_2) \rightarrow_{\delta'_{i-1}} q^T$$

(Le cas symétrique où $v = \cdot(v_1, v_2) = \cdot(v_1, C'[p^T])$ est similaire, dans ce cas le terme v_1 est nécessairement annoté par un état de la forme s^{nil}). Nous distinguons deux cas selon la nature des états l_1 et l_2 :

i. l_1 est de la forme p_1^T. Alors nécessairement l_2 est de la forme p_2 (règles (4c)). Ceci veut dire que

$$v_2 \in L_{p_2} \tag{5.4}$$

et par induction nous obtenons que

$$v_1 \in Post^*_=(L_{p_1}). \tag{5.5}$$

Soit donc $u_1 \in L_{p_1}$ tel que $v_1 \in Post^*_=(u_1)$. En plus, puisque $\cdot(p_1^T, p_2) \rightarrow q^T$ est une règle de δ', les règles (4c) impliquent que $\cdot(p_1, p_2) \rightarrow q$ est nécessairement une règle de δ. Ceci veut dire que $\cdot(u_1, v_2) \in L_q$. Comme $v = \cdot(v_1, v_2) \in Post^*_=\big(\cdot\,(u_1, v_2)\big)$, nous obtenons que :

$$v \in Post^*_=(L_q)$$

ii. l_2 est de la forme p_2^T. Alors nécessairement, l_1 est de la forme p_1^{nil} (règles (4a)). Par induction sur i nous obtenons que

$$v_2 \in Post^*_=(L_{p_2}) \tag{5.6}$$

et par induction sur k, nous déduisons qu'il existe un terme $u'_1 \sim_0 0$ tel que

$$u'_1 \in Post^*_=(L_{p_1}), \text{ et } v_1 \in Post^*_=(u'_1) \tag{5.7}$$

Soient donc $u_1 \in L_{p_1}$ tel que $u'_1 \in Post^*_=(u_1)$, et $u_2 \in L_{p_2}$ tel que $v_2 \in Post^*_=(u_2)$. En plus, puisque $\cdot(p_1^{nil}, p_2^T) \rightarrow q^T$ est une règle de

δ', les règles (4a) impliquent que $\cdot(p_1, p_2) \to q$ est nécessairement une règle de δ. Ceci veut dire que $\cdot(u_1, u_2) \in L_q$. Par conséquent, nous obtenons que

$$\cdot(u'_1, u_2) \in Post^*_=(\cdot(u_1, u_2));$$

$$\cdot(u'_1, v_2) \in Post^*_=(\cdot(u'_1, u_2))$$

puisque $u'_1 \sim_0 0$, et donc u_2 peut être réécrit, et

$$v = \cdot(v_1, v_2) \in Post^*_=(\cdot(u'_1, v_2)).$$

Donc, $v \in Post^*_=(\cdot(u_1, u_2))$, et

$$v \in Post^*_=(L_q)$$

(c) Il existe deux termes v_1 et v_2, et un contexte C' tels que

$$v = ||(v_1, v_2) \xrightarrow{*}_{\delta'_i} ||(C'[p^T], v_2) \xrightarrow{k-1}_{\delta'_{i-1}} ||(l_1, l_2) \to_{\delta'_{i-1}} q^T$$

Ce cas est similaire au cas précédent. Il est même plus simple puisqu'il n'est pas nécessaire d'imposer que si l_1 est de la forme p_2^T, alors l_1 doit être de la forme p_1^{nil}. Dans ce cas, nous utilisons les règles (5) au lieu des règles (4).

2. Il existe un contexte C tel que $v = C[v_1, v_2]$, et

$$v = C[v_1, v_2] \xrightarrow{*}{}^{n}_{\delta'_i} C[s_1, s_2] \xrightarrow{k}_{\delta'_{i-1}} q^T$$

tel qu'il existe $n' < n$ t.q.

$$v_1 \xrightarrow{*}{}^{n'}_{\delta'_i} s_1 \text{ et } v_2 \xrightarrow{*}{}^{(n-n')}_{\delta'_i} s_2$$

Supposons que $s_i = p_i^T$ (les autres cas sont similaires). Alors, nous déduisons par induction que

$$v_1 \in Post^*_=(L_{p_1}), \tag{5.8}$$

et

$$v_2 \in Post^*_=(L_{p_2}). \tag{5.9}$$

Ensuite, nous pouvons montrer par induction sur k que $v \in Post^*_=(L_q)$.

□

Lemme 5.3.2 *Pour chaque $q \in Q \cup Q_{\mathcal{R}}$, nous avons :*
- $Post^*_=(L_q) \subseteq L_{q^T}$,
- $Post^*_=\big(L_q \cap Post^*_=(\{u \mid u \sim_0 0\})\big) \subseteq L_{q^{nil}}$.

Preuve : Nous ne montrons que le premier point. Le second peut être montré de manière similaire. Puisque $L_q \subseteq L_{q^T}$ (règles (1)), il suffit de montrer que pour tout $t \in L_{q^T}$, si $t' \in Post(t)$, alors $t' \in L_{q^T}$. Soit alors $t \in L_{q^T}$, c-à-d. $t \xrightarrow{*}_{\delta'} q^T$. Considérons le contexte C tel que $t = C[t_1], t' = C[t_2]$, et $t_1 \to t_2 \in \mathcal{R}$. Soit alors $p \in Q'$ tel que

$$t \xrightarrow{*}_{\delta'} C[p] \xrightarrow{*}_{\delta'} q^T$$

(c-à-d., pendant l'annotation de t, t_1 est annoté par l'état p). Il y a deux cas :

1. $p \notin Q \cup Q_{\mathcal{R}}$. Soit alors $q' \in Q \cup Q_{\mathcal{R}}$ tel que $p = {q'}^T$ ou $p = {q'}^{nil}$. Nous considérons le cas où $p = {q'}^T$ (l'autre cas est similaire) c-à-d. $t_1 \xrightarrow{*}_{\delta'} {q'}^T$. Les règles (3) impliquent que $t_2 \xrightarrow{*}_{\delta'} q^T_{t_2} \to_{\delta'} {q'}^T = p$. Par conséquent, $t' \xrightarrow{*}_{\delta'} C[p] \xrightarrow{*}_{\delta} q^T$,
2. $p \in Q \cup Q_{\mathcal{R}}$. Nous devons montrer que $t' \xrightarrow{*}_{\delta'} C[p^T] \xrightarrow{*}_{\delta'} q^T$. Nous procédons par induction structurelle sur C (le cas où C est le contexte trivial fait partie du cas précédent) :
   - Il y a un $f \in \{\cdot, ||\}$, et un terme $u \in \mathcal{T}$ tels que $C[p] = f(p, u)$ (le cas symétrique où $C[p] = f(u, p)$ est similaire). Prenons par exemple $f = ||$, l'autre cas peut se traiter de la même manière. Soit $l \in Q'$ tel que $C[p] = ||(p, u) \xrightarrow{*}_{\delta'} ||(p, l) \xrightarrow{*}_{\delta'} q^T$. Alors, nécessairement, nous avons grâce aux règles (5b) que $C[p^T] = ||(p^T, l) \xrightarrow{*}_{\delta'} q^T$,
   - Il y a un $f \in \{\cdot, ||\}$, un contexte C', et un terme $u \in \mathcal{T}$ tels que $C[] = f(C'[], u)$ (le cas symétrique où $C[] = f(u, C'[])$ est similaire). Considérons par exemple le cas où $f = ||$, l'autre cas peut se traiter de la même manière. Soient $l_1, l_2 \in Q \cup Q_{\mathcal{R}}$ tels que $C[p] = ||(C'[p], u) \xrightarrow{*}_{\delta'} ||(l_1, l_2) \xrightarrow{*}_{\delta'} {p'}^T$. (les autres cas où $C[p] \xrightarrow{*}_{\delta'} ||(l_1^T, l_2^T)$, ...etc peuvent être traités de la même manière). Alors, les règles (5b) impliquent que nous avons $||(l_1^T, l_2) \xrightarrow{*}_{\delta'} q^T$. Puisque $C'[p] \xrightarrow{*}_{\delta'} l_1$, par induction structurelle sur C' nous obtenons que $C'[p^T] \xrightarrow{*}_{\delta'} l_1^T$. Par conséquent, $C[p^T] = ||(C'[p^T], u) \xrightarrow{*}_{\delta'} ||(l_1^T, l_2) \xrightarrow{*}_{\delta'} q^T$.

□

5.4 Accessibilité modulo $\sim_0$

Modulo $\sim_0$, chaque terme t est équivalent aux termes $t \cdot 0$, $0 \cdot t$, $t||0$, $0||t$, $0 \cdot (t \cdot 0)$,$(0 \cdot 0) \cdot t$, $(0||0) \cdot t$, ... etc. Par conséquent, pour faire l'analyse d'accessibilité modulo $\sim_0$, en partant d'un langage initial régulier $\mathcal{L}$ reconnu par un automate fini $\mathcal{A} = (Q, \Sigma, F, \delta)$, nous pouvons procéder comme suit : D'abord, rajouter des termes équivalents à 0 partout dans les termes de $\mathcal{L}$. Ceci peut se faire en considérant un état spécial q_{null} qui reconnaît tous les termes nuls (tous les termes qui sont équivalents à 0), et en rajoutant de nouvelles règles qui permettent d'avoir les termes nuls reconnus par q_{null} partout dans les arbres (par exemple, ajouter des règles de la forme $||(q, q_{null}) \to q$, pour chaque $q \in Q \cup Q_{\mathcal{R}}$). Soit donc $Q_{null} = Q \cup Q_{\mathcal{R}} \cup \{q_{null}\}$, et δ_{null} l'ensemble des règles contenant $\delta \cup \delta_{\mathcal{R}}$, et les règles suivantes qui permettent, pour chaque état $q \in Q \cup Q_{\mathcal{R}}$, de rajouter des 0 partout dans les arbres reconnus par q :

– $0 \to q_{null}$ (0 est accepté par q_{null}),
– $\cdot(q_{null}, q_{null}) \to q_{null}$ (la composition séquentielle de deux termes nuls est un terme nul),
– $||(q_{null}, q_{null}) \to q_{null}$ (la composition parallèle de deux termes nuls est un terme nul),
– $\cdot(q_{null}, q) \to q$ (cette règle exprime que $\cdot(0, t) \sim_0 t$),
– $\cdot(q, q_{null}) \to q$ (cette règle exprime que $\cdot(t, 0) \sim_0 t$),
– $||(q_{null}, q) \to q$ (cette règle exprime que $||(0, t) \sim_0 t$),
– $||(q, q_{null}) \to q$ (cette règle exprime que $||(t, 0) \sim_0 t$).

$Post^*_{\mathcal{R},\sim_0}(\mathcal{L})$ est reconnu par l'automate $\mathcal{A}^{*nil}_{\mathcal{R}} = (Q'_{null}, \Sigma, F', \delta'_{null})$ tel que δ'_{null} est le plus petit ensemble de règles qui contient δ_{null}, et tel que :

1. $q \to q^T \in \delta'_{null}$ pour chaque $q \in Q_{null}$;
2. si $t_1 \to t_2 \in \mathcal{R}$, et il existe un état $q \in Q_{null}$ tel que $t_1 \stackrel{*}{\to}_{\delta'_{null}} q^T$, alors $q^T_{t_2} \to q^T \in \delta'_{null}$;
3. si $\cdot(q_1, q_2) \to q \in \delta_{null}$, alors $\cdot(q^T_1, q_2) \to q^T \in \delta'_{null}$;
4. si $||(q_1, q_2) \to q \in \delta_{null}$, alors $||(q^T_1, q^T_2) \to q^T \in \delta'_{null}$;
5. si $q \to q' \in \delta_{null}$, alors $q^T \to {q'}^T \in \delta'_{null}$;
6. si $0 \stackrel{*}{\to}_{\delta'_{null}} q^T_1$, et $\cdot(q_1, q_2) \to q \in \delta_{null}$, ou $||(q_1, q_2) \to q \in \delta_{null}$, ou $||(q_2, q_1) \to q \in \delta_{null}$, alors δ'_{null} contient la règle $q^T_2 \to q^T$;
7. $\cdot(q^T_{null}, q^T) \to q^T \in \delta'_{null}$.

Comme précédemment, les états q^T reconnaissent les successeurs de L_q. Les cinq premières règles d'inférence sont exactement celles de la construction de $Post^*_{\mathcal{R},=}(\mathcal{L})$. La règle (6) est une règle de simplification, elle exprime que si par exemple $||(q_1, q_2) \to q$, ou $\cdot(q_1, q_2) \to q$ sont des règles de δ_{null}, 0 est un successeur de L_{q_1}, et u est un successeur de L_{q_2}, alors u doit aussi être un successeur de L_q. Les règles (7) expriment que si u_1 est un successeur de 0 et u_2 est un successeur de q, alors $\cdot(u_1, u_2)$ est un successeur de q. Notons que dans ce cas les états de la forme "q^{nil}" ne sont plus nécessaires. En effet, ces états étaient utilisés pour respecter la sémantique du point : si $\cdot(q_1, q_2) \to q$ est une règle de l'automate initial, alors nous considérons la règle $\cdot(q^{nil}_1, q^T_2) \to q^T$ qui exprime que si $u = \cdot(u_1, u_2) \in L_q$ est tel que $u_1 \in L_{q_1}$ et $u_2 \in L_{q_2}$, et v_1, v_2 des successeurs de u_1 et u_2 respectivement tels que $v_2 \neq u_2$; alors $v = \cdot(v_1, v_2)$ est un successeur de u ssi il existe un terme nul u'_1 tel que u'_1 est un successeur de u_1 et v_1 est un successeur de u'_1. Dans le cas précédent, v_1 et v_2 étaient respectivement reconnus par les états q^{nil}_1 et q^T_2 ; et la règle ci-dessus permettait d'annoter v par q^T. Maintenant, c'est l'état q^T_{null} qui joue le rôle des états indicés par "^{nil}". En effet, comme u'_1 est nul, il est reconnu par q_{null}, et donc v_1 est reconnu par q^T_{null}. Les règles (7) permettent d'annoter v par q^T_2. Comme u'_1 est nul et est successeur de L_{q_1}, les règles de simplification (6) permettent d'avoir la règle de transition $q^T_2 \to q^T$, ce qui permet d'annoter v par q^T. Nous obtenons alors le résultat suivant :

Théorème 5.4.1 *Soit $\mathcal{R}$ un PRS et $\mathcal{L}$ un langage régulier d'arbres reconnu par l'automate $\mathcal{A} = (Q, \Sigma, F, \delta)$, alors l'automate $\mathcal{A}^{*nil}_{\mathcal{R}}$ (resp. $\mathcal{A}^{*nil}_{\mathcal{R}^{-1}}$) reconnaît $Post^*_{\mathcal{R},\sim_0}(\mathcal{L})$ (resp. $Pre^*_{\mathcal{R},\sim_0}(\mathcal{L})$). De plus, ces automates peuvent être calculés en temps polynômial.*

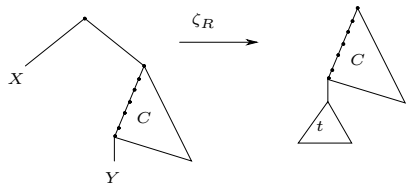


FIG. 5.2 – Application de la règle $X \cdot Y \to t$ modulo $\sim_s$

Nous obtenons directement à partir de ce théorème et de la proposition 5.2.6 que :

Corollaire 5.4.1 *Soit $\mathcal{R}$ un PAD, alors nous pouvons effectivement calculer en temps polynômial :*

- *un $\sim$-représentant régulier de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$ si $\mathcal{L}$ est $\sim_s$-compatible ;*
- *un $\sim$-représentant régulier de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ si $\mathcal{L}$ est $\sim$-compatible.*

5.5 Accessibilité modulo $\sim_s$

Nous montrons dans cette section que pour chaque langage régulier quelconque $\mathcal{L}$ et chaque PRS $\mathcal{R}$, nous pouvons calculer effectivement des automates d'arbres finis qui reconnaissent des $\sim_s$-représentants de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$ et $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L})$. Lorsque nous considérons l'associativité du "$\cdot$", la difficulté principale vient du fait que les réécritures ne se font plus de manière locale comme dans les cas précédents. En effet, dans les sections précédentes, une règle $X \cdot Y \to t$ est appliquée à un terme u seulement si $\cdot(X, Y)$ est un sous terme explicite de u. Ceci n'est plus le cas quand les termes sont considérés modulo $\sim_s$. En effet, cette règle doit être appliquée par exemple aux termes $\cdot\Big(X, \cdot\big(Y, \cdot(Z, T)\big)\Big)$ et $\cdot\Big(X, \cdot\big(\cdot(Y, Z), T\big)\Big)$ puisqu'ils sont $\sim_s$-équivalent à $\cdot\big(\cdot(X, Y), \cdot(Z, T)\big)$.

De manière générale, si nous définissons un *seq-contexte* comme suit :

Définition 5.5.1 *Soit $x \in \mathcal{X}$, un* seq-contexte *est un contexte à une seule variable $C[x]$ tel que : (1) x est la feuille la plus à gauche de C, et (2) tous les ancêtres de la variable x sont étiquetés par "$\cdot$".*

Alors, modulo $\sim_s$, une règle $X \cdot Y \to t$ peut être appliquée à tous les termes de la forme $\cdot(X, C[Y])$ pour un seq-contexte C, pour produire un terme qui est $\sim$-équivalent à $C[t]$. Nous définissons maintenant la relation $\zeta_{\mathcal{R}}$ qui réalise cette transformation pour un PRS arbitraire. Cette relation est définie comme la plus petite relation sur $\mathcal{T}$ qui contient $\Rightarrow_{\sim_0,\mathcal{R}}$ (puisque $\sim_s$ est plus fine que $\sim_0$) et telle que pour chaque règle $X \cdot Y \to t$ dans $\mathcal{R}$, chaque contexte C', et chaque seq-contexte C, $\Big(C'\big[\cdot(X, C[Y])\big], C'\big[C[t]\big]\Big) \in \zeta_{\mathcal{R}}$. Cette transformation est représentée sur la figure 5.2.

Il est facile de voir que $\zeta^*_{\mathcal{R}}(\mathcal{L})$ est un $\sim_s$-représentant de $Post^*_{\mathcal{R},\sim_s}(\mathcal{L})$. Ainsi, pour chaque langage d'arbres $\mathcal{L}$, nous avons :

Proposition 5.5.1 *Pour chaque PRS $\mathcal{R}$ et langage régulier $\mathcal{L}$, $Post^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [\zeta^*_{\mathcal{R}}(\mathcal{L})]_{\sim_s}$ et $Pre^*_{\mathcal{R},\sim_s}(\mathcal{L}) = [\zeta^*_{\mathcal{R}^{-1}}(\mathcal{L})]_{\sim_s}$.*

Notre objectif est donc de calculer $\zeta^*_{\mathcal{R}}(\mathcal{L})$ pour un langage régulier $\mathcal{L}$. Dans le reste de cette section, nous montrons le théorème suivant :

Théorème 5.5.1 *Soit $\mathcal{L}$ un langage régulier, alors $\zeta^*_{\mathcal{R}}(\mathcal{L})$ est effectivement régulier. En plus, à partir d'un automate ayant k états et τ transitions qui reconnaît $\mathcal{L}$, il est possible de construire, en un temps polynômial, un automate acceptant $\zeta^*_{\mathcal{R}}(\mathcal{L})$, ayant $\mathcal{O}\big((k + |Sub_r(\mathcal{R})|)^2|Var|^2 + \tau\big)$ transitions et $\mathcal{O}(k|Var| + |Sub_r(\mathcal{R})||Var|)$ états.*

Nous déduisons alors à partir de ce théorème et du lemme 4.4.3 que :

Corollaire 5.5.1 *Soit P un programme à parallélisme binaire, $\mathcal{R}$ le PRS correspondant, $\mathcal{L}$ un ensemble de termes de processus, et $(e_{\texttt{main}}, loc)$ la configuration initiale du programme. Alors il est possible calculer en temps polynômial un $\sim$-représentant de $Post^*_{\mathcal{R}}\big((e_{\texttt{main}}, loc)\big)$. De même, il est possible de déterminer en temps polynômial si $Pre^*_{\mathcal{R}}(\mathcal{L}) \cap (e_{\texttt{main}}, loc) = \emptyset$.*

Nous obtenons également de manière directe à partir du théorème 5.5.1 et de la proposition 5.2.3 que :

Corollaire 5.5.2 *Soit $\mathcal{R}$ un PAD, alors nous pouvons effectivement calculer en temps polynômial :*
- *un $\sim$-représentant régulier de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$;*
- *un $\sim$-représentant régulier de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ si $\mathcal{L}$ est $\sim_{||}$-compatible.*

5.5.1 Construction

Soient $\mathcal{L}$ un langage d'arbres régulier, et $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres qui reconnaît $\mathcal{L}$. Puisque nous considérons aussi la neutralité du 0, soient δ_{null}, q_{null}, et $\mathcal{A}_{null} = (Q_{null}, \Sigma, F, \delta_{null})$ la relation de transitions, l'état, et l'automate définis dans la section précédente, respectivement.

Nous définissons l'automate $\mathcal{A}^*_{\zeta_{\mathcal{R}}} = (\tilde{Q}, \Sigma, \tilde{F}, \tilde{\delta})$ tel que :
- $\tilde{Q} = \{q, q^T \mid q \in Q_{null}\} \cup \{(q, X) \mid q \in Q_{null}, X \in Var\}$,
- $\tilde{F} = \{q, q^T \mid q \in F\}$,
- $\tilde{\delta}$ est le plus petit ensemble de règles contenant δ_{null} et tel que pour chaque $q_1, q_2, q \in Q_{null}$, et chaque $X, Y, X' \in Var$:

  (α_1) $q \to q^T \in \tilde{\delta}$,

  (α_2) si $Y \xrightarrow{*}_{\tilde{\delta}} q$ et $X \cdot Y \to t \in \mathcal{R}$, alors $q^T_t \to (q, X) \in \tilde{\delta}$,

  (α_3) si $t_1 \to t_2 \in \mathcal{R}$ et il existe un état $q \in Q_{null}$ tel que :

    (a) $t_1 \xrightarrow{*}_{\tilde{\delta}} q^T$, alors : $q^T_{t_2} \to q^T \in \tilde{\delta}$, ou

    (b) $t_1 \xrightarrow{*}_{\tilde{\delta}} (q, X)$, alors : $q^T_{t_2} \to (q, X) \in \tilde{\delta}$,

  (α_4) si $\cdot(q_1, q_2) \to q \in \delta_{null}$, alors :

    (a) $\cdot(q^T_1, q_2) \to q^T \in \tilde{\delta}$,

(b) $\cdot\big((q_1, X), q_2\big) \to (q, X) \in \tilde{\delta}$,

(c) si $0 \xrightarrow{*}_{\tilde{\delta}} (q_1, X)$ alors $q_2^T \to (q, X) \in \tilde{\delta}$,

(d) si $0 \xrightarrow{*}_{\tilde{\delta}} q_1^T$ alors $q_2^T \to q^T \in \tilde{\delta}$,

(e) si $X \xrightarrow{*}_{\tilde{\delta}} q_1^T$ alors $(q_2, X) \to q^T \in \tilde{\delta}$,

(f) si $X \xrightarrow{*}_{\tilde{\delta}} (q_1, X')$ alors $(q_2, X) \to (q, X') \in \tilde{\delta}$,

(α_5) si $||(q_1, q_2) \to q \in \delta_{null}$, alors :

(a) $||(q_1^T, q_2^T) \to q^T \in \tilde{\delta}$,

(b) si $0 \xrightarrow{*}_{\tilde{\delta}} q_1^T$ alors $q_2^T \to q^T \in \tilde{\delta}$,

(c) si $0 \xrightarrow{*}_{\tilde{\delta}} q_2^T$ alors $q_1^T \to q^T \in \tilde{\delta}$,

(α_6) si $q \to q' \in \delta_{null}$, alors $q^T \to q'^T \in \tilde{\delta}$, et $(q, X) \to (q', X) \in \tilde{\delta}$,

(α_7) $\cdot(q_{null}^T, q^T) \to q^T \in \tilde{\delta}$.

Notons que les régles d'inférence (α_2), (α_3), (α_4), et (α_5) construisent une séquence finie d'ensembles croissants de transitions $\tilde{\delta}_1 \subset \tilde{\delta}_2 \subset \ldots \subset \tilde{\delta}_n$. Cette procédure termine parcequ'il y a un nombre fini d'états dans $\tilde{Q}$.

Nous avons alors le résultat suivant :

Théorème 5.5.2 *Soit $\mathcal{L}$ un ensemble régulier de termes de processus, et soit $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini reconnaissant $\mathcal{L}$. Alors, $\zeta_{\mathcal{R}}^*(\mathcal{L})$ est reconnu par l'automate d'arbres $\mathcal{A}_{\zeta_{\mathcal{R}}}^*$.*

5.5.2 Intuition

Pour annoter un sous terme ayant la forme décrite dans la partie droite de la figure 5.2, l'automate devine quand il arrive à la racine du sous terme t qu'à cette position il y avait un Y qui a été réécrit par la règle $X \cdot Y \to t$. Cette supposition doit être validée plus tard, à la racine du seq-contexte C.

L'idée est la même que précédemment : q^T reconnaît les successeurs de L_q par $\zeta_{\mathcal{R}}$. L'automate utilise les nouveaux états (q, X) pour pouvoir deviner. Un terme u est annoté par (q, X) s'il existe un seq-contexte C et une règle $r = X \cdot Y \to t$ dans $\mathcal{R}$ tels que $C[Y] \in L_q$ et l'automate devine que la règle r a été appliquée à $\cdot(X, C[Y])$ pour obtenir $C[t]$ d'abord, et ensuite u après plusieurs réécritures ($u \in \zeta_{\mathcal{R}}^*(C[t])$). Ceci veut dire que s'il existe une règle de l'automate de la forme $\cdot(q', q) \to q''$ telle que $X \xrightarrow{*}_{\tilde{\delta}} {q'}^T$ (c-à-d. $X \in \zeta_{\mathcal{R}}^*(L_{q'})$), alors l'automate peut valider sa prédiction et déduire que $u \in \zeta_{\mathcal{R}}^*(L_{q''})$. Ceci est exprimé par les règles ($\alpha_4 e$). En plus, si $X \xrightarrow{*}_{\tilde{\delta}} (q', X')$, ce qui signifie que l'automate a deviné que X peut être obtenu à partir de $L_{q'}$ en utilisant un X' du voisinage gauche de q', alors il peut valider la prédiction concernant X et mémoriser la prédiction concernant X', c-à-d., il peut annoter u avec (q'', X'). Ceci est exprimé par les règles ($\alpha_4 f$). Ces prédictions se font à la racine des successeurs du sous-terme t par les règles (α_2), et sont remontées à la racine du contexte C par les règles ($\alpha_4 b$) qui s'assurent que C est bien un seq-contexte.

Les règles (α_3) expriment que si $t_1 \to t_2$ est une règle de $\mathcal{R}$, et si t_1 est un successeur de L_q, alors tous les successeurs de t_2 le sont aussi. De même, si l'automate a deviné

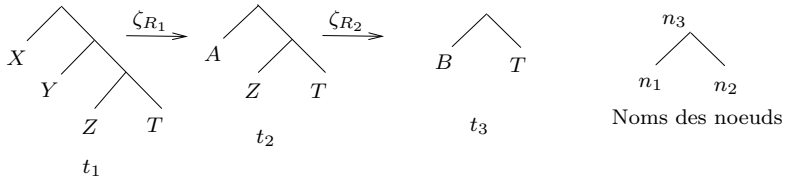


FIG. 5.3 – Exemple 1

que t_1 est un successeur de L_q (si t_1 est annoté par (q, X)), alors il a aussi deviné que les successeurs de t_2 sont aussi des successeurs de L_q. Les règles $(\alpha_4 c)$, $(\alpha_4 d)$, $(\alpha_5 b)$, $(\alpha_5 c)$ et (α_7) prennent en compte la neutralité de 0 comme c'est le cas dans la section 5.4.

Notons que le point crucial de cette construction est qu'à chaque nœud l'automate doit mémoriser au plus une prédiction. Ceci est dû à la sémantique de l'opérateur "$\cdot$" qui assure qu'à chaque nœud, au plus deux réécritures non locales peuvent avoir lieu. En effet, considérons le nœud étiqueté par Y sur la figure 5.2. Il est d'abord réécrit en t, et l'automate mémorise X. Ensuite, puisque le voisinage gauche de ce nœud est nul, une règle ζ ne peut s'appliquer à cette position sans la faire disparaître que s'il y a deux variables Z et W telles que $\mathcal{R}$ contient une règle r' de la forme $W \cdot Z \to u$ où Z est un successeur de t, et W est un successeur du processus nul 0, c-à-d., $W \in \zeta^*_{\mathcal{R}}(0)$. Dans ce cas, le terme $C[t]$ de la figure 5.2 peut se transformer d'abord en $\cdot(W, C[Z])$ (puisque $C[t]$ est équivalent à $\cdot(0, C[t])$, Z est un successeur de t, et W est un successeur de 0), et ensuite en $C[u]$ en appliquant la règle r'. Des explications précédentes, on pourrait croire que pendant l'annotation du sous terme u, l'automate doit faire deux prédictions puisqu'il doit deviner tout d'abord qu'à la place de u il y avait un Z qui a consommé un W du voisinage gauche pour appliquer la règle r', et ensuite qu'à la place de ce Z il y avait un Y qui a consommé un X pour appliquer la règle r. Cependant, l'automate construit ne se comporte pas de cette manière, puisqu'il ne peut faire qu'une seule prédiction à la fois (les règles α_2). Pour pouvoir déterminer que le terme $C[u]$ est un successeur de $\cdot(X, C[Y])$ en ne faisant qu'une seule prédiction, l'automate observe que $C[u]$ est aussi un successeur de $C[\cdot(W, Z)]$, qui est successeur de $C[\cdot(0, Z)]$, et par conséquent de $C[t]$ (puisqu'il est $\sim_0$-équivalent à $C[\cdot(0, t)]$ et $\zeta_{\mathcal{R}}$ contient $\Rightarrow_{\sim_0, \mathcal{R}}$). Donc, quand il annote le sous terme u, l'automate détermine que c'est un successeur de $\cdot(W, Z)$, qui est lui-même successeur de $\cdot(0, Z)$, c-à-d., de Z, et donc l'automate devine juste que pour obtenir ce Z, un X a été consommé en appliquant la règle r. Ainsi, une seule prédiction est faite.

5.5.3 Exemple

Soit $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini qui reconnaît le terme t_1 de la figure 5.3, où $Q = \{q_1, \ldots, q_7\}$, $F = \{q_7\}$, et δ contient les règles suivantes :

- $X \to q_1, Y \to q_2, Z \to q_3, T \to q_4$,
- $\cdot(q_3, q_4) \to q_5, \cdot(q_2, q_5) \to q_6, \cdot(q_1, q_6) \to q_7$.

Soit $\mathcal{R}$ le PRS qui contient les règles $\mathcal{R}_1 : X \cdot Y \to A$ et $\mathcal{R}_2 : A \cdot Z \to B$. $\mathcal{R}_1$ peut s'appliquer à t_1 pour le transformer en t_2, qui peut se transformer en t_3 en appliquant $\mathcal{R}_2$. Pour reconnaître le terme t_3 comme successeur du terme t_1, l'automate annote d'abord les feuilles de t_3 : Il annote T par q_4 en utilisant les règles de l'automate $\mathcal{A}$. Pour annoter n_1, il devine qu'à la place de B il y avait un Z qui a été transformé en utilisant la règle $\zeta_{\mathcal{R}_2}$. Puisque l'automate $\mathcal{A}$ reconnaît Z avec l'état q_3, l'automate $\mathcal{A}^*_{\zeta_\mathcal{R}}$ annote le nœud n_2 par (q_3, A) (règles α_2), indiquant que ce nœud a "consommé" un A de son voisinage gauche (c-à-d., la règle $A \cdot Z \to B$ a été appliquée). Puisque $\cdot(q_3, q_4) \to q_5$ est une règle de l'automate $\mathcal{A}$, le nœud n_3 est annoté par (q_5, A) (règles α_4b), pour exprimer qu'initialement, il y avait un terme de la forme $C[Z]$, pour un certain seq-contexte C (ici $C[Z]$ est $\cdot(Z, T)$), qui est reconnu par q_5 et qui a utilisé un A, comme décrit sur la figure 5.2, pour devenir le terme t_3. Maintenant, l'automate doit valider cette prédiction concernant le A, c-à-d., il doit vérifier qu'il y a bien un A à gauche de ce nœud. l'automate devine que ce A qui a été consommé du voisinage gauche provient de l' application de $\zeta_{\mathcal{R}_1}$, c-à-d., que cette variable a pris la place d'un Y après avoir consommé un X du voisinage gauche (en utilisant la règle $X \cdot Y \to A$). Ceci veut dire que l'automate devine que A peut être annoté par (q_2, X) puisque Y est reconnu par q_2 (règles α_2). Puisque $\cdot(q_2, q_5) \to q_6$ est une règle de $\mathcal{A}$, nous déduisons que l'automate a réussi à montrer qu'il y avait bien un A à gauche de n_3, pourvu qu'il y a un X qui a été consommé du voisinage gauche. Le nœud n_3 est alors annoté par (q_6, X) (règles α_4f). Maintenant, il est facile de montrer qu'il y avait un X à gauche puisque nous avons $\cdot(q_1, q_6) \to q_7$, et $X \to q_1$ comme règles de $\mathcal{A}$. Donc, n_3 est annoté par q_7^T, exprimant qe t_3 est un successeur du terme t_1 (règles α_4e).

Nous allons construire maintenant l'automate $\mathcal{A}^*_{\zeta_\mathcal{R}}$. Pour ce faire, nous allons calculer les règles $\tilde{\delta}$ (nous ne considérons que les règles dont nous avons besoin pour annoter les termes de la figure 5.3) :

(r_1) De (α_1) nous obtenons que $q_7 \to q_7^T$ est dans $\tilde{\delta}$.

(r_2) Puisque $Y \to_\delta q_2$ et $X \cdot Y \to A$ est une règle de $\mathcal{R}$, les règles (α_2) impliquent que $q_A^T \to (q_2, X)$ est dans $\tilde{\delta}$.

(r_3) Puisque $\cdot(q_2, q_5) \to q_6 \in \delta$, les règles ($\alpha_4 b$) impliquent que $\tilde{\delta}$ contient la règle $\cdot((q_2, X), q_5) \to (q_6, X)$.

(r_4) Puisque $\cdot(q_1, q_6) \to q_7 \in \delta$ et $X \to_\delta q_1$, les règles ($\alpha_4 e$) impliquent que $(q_6, X) \to q_7^T$ est dans $\tilde{\delta}$.

(r_5) Puisque $Z \to_\delta q_3$ et $A \cdot Z \to B$ est une règle de $\mathcal{R}$, les règles (α_2) impliquent que $q_B^T \to (q_3, A)$ est dans $\tilde{\delta}$.

(r_6) Puisque $\cdot(q_3, q_4) \to q_5 \in \delta$, les règles ($\alpha_4 b$) impliquent que $\tilde{\delta}$ contient la règle $\cdot((q_3, A), q_4) \to (q_5, A)$.

(r_7) Puisque $\cdot(q_2, q_5) \to q_6 \in \delta$ et $A \to_\delta q_A \to_{\tilde{\delta}} q_A^T \to_{\tilde{\delta}} (q_2, X)$, les règles ($\alpha_4 f$) impliquent que $(q_5, A) \to (q_6, X)$ est dans $\tilde{\delta}$.

Avec ces règles, tous les termes de la figure (5.3) peuvent être reconnus comme successeurs du terme t_1. Plus précisément, tous ces termes peuvent être annotés par l'état final q_7^T comme suit :

- t_1 est annoté par q_7^T, puisqu'il est annoté par q_7 (r_1).
- Concernant t_2, le sous terme $\cdot(Z, T)$ est annoté par q_5 en utilisant les règles de

δ. Ensuite, A est annoté par (q_2, X) (en utilisant r_2). Grâce à (r_3), la racine de t_2 est annotée par (q_6, X), et ensuite par q_7^T grâce à (r_4).
- Pour t_3, T est annoté par q_4 et B par (q_3, A) (grâce à (r_5)). Par conséquent, nous pouvons annoter t_3 par (q_5, A) en appliquant (r_6), ensuite par (q_6, X) en appliquant (r_7), et finalement par q_7^T grâce à (r_4).

5.5.4 Preuve du théorème 5.5.2

Nous montrons que :

Lemme 5.5.1 *Pour tout $q \in Q \cup Q_{\mathcal{R}}$:*
- $L_{q^T} = \zeta_{\mathcal{R}}^*(L_q)$,
- $L_{(q,X)} = \{u \in \mathcal{T} \mid \exists$ *un seq-contexte* C, *une règle* $X \cdot Y \to t$, *t.q.* $u \in \zeta_{\mathcal{R}}^*(C[t])$, *et* $C[Y] \in L_q\}$.

Preuve :

$\Rightarrow$: Nous montrons d'abord les inclusions $\subseteq$. La preuve est fastidieuse et il y a plusieurs cas à considérer. Nous n'allons considérer que les cas les plus intéressants. Nous montrons par induction sur i que pour chaque $v \in \mathcal{T}$ et chaque $q \in Q \cup Q_{\mathcal{R}}$, nous avons :
- $v \xrightarrow{*}_{\tilde{\delta}_i} q^T \Rightarrow v \in \zeta_{\mathcal{R}}^*(L_q)$,
- $v \xrightarrow{*}_{\tilde{\delta}_i} (q, X) \Rightarrow \exists$ un seq-contexte C, une règle $X \cdot Y \to t$, t.q. $v \in \zeta_{\mathcal{R}}^*(C[t])$, et $C[Y] \in L_q$.

• $i = 0$, alors $\tilde{\delta}_0$ contient les règles α_1, α_2, α_4a, α_4b, α_5a, α_6, et α_7. Il y a deux cas :

1. Supposons que $v \xrightarrow{*}_{\tilde{\delta}_1} q^T$. A ce stade de la construction de $\tilde{\delta}$, une telle exécution ne considère pas les états (q, X) . Par conséquent, de par la nature des règles de $\tilde{\delta}_0$, il s'en suit que $v \xrightarrow{*}_{\tilde{\delta}_0} q^T$ ssi $v \xrightarrow{*}_{\delta_{null}} q$. Ceci implique que $v \in [L_q]_{\sim_0} \subseteq \zeta_{\mathcal{R}}^*(L_q)$,
2. Supposons que $v \xrightarrow{*}_{\tilde{\delta}_1} (q, X)$. Alors, les règles de $\tilde{\delta}_1 \setminus \tilde{\delta}_0$ sont nécessairement rajoutées par les règles d'inférence (α_2). Nous montrons par induction sur k que $v \xrightarrow{k}_{\tilde{\delta}_1} (q, X)$ implique l'existence d'un seq-contexte C, un terme $t \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \to t$ dans $\mathcal{R}$ tels que $v \in \zeta_{\mathcal{R}}^*(C[t])$, et $C[Y] \in L_q$:
   - La propriété est satisfaite pour $k = 3$ (le cas de base). En effet, dans ce cas la seule possibilité est d'avoir une dérivation de la forme suivante :

     $$v = Z \to_{\tilde{\delta}_0} q_Z \to_{\tilde{\delta}_0} q_Z^T \to_{\tilde{\delta}_1} (q, X)$$

     Alors, nécessairement, grâce aux règles (α_2), il existe une variable Y, et une règle $X \cdot Y \to Z$ dans $\mathcal{R}$ tels que $Y \xrightarrow{*}_{\tilde{\delta}_0} q$, c-à-d. $Y \in L_q$, et la propriété est satisfaite avec un C égal au contexte trivial. Si une telle règle n'existe pas, c-à-d. si toutes les règles de $\mathcal{R}$ de la forme $X \cdot Y \to t$ sont telles que $t = f(V, W)$ $(f \in \{., ||\})$, alors nous devons commencer l'induction avec $k = 4$.
   - $k > 3$. Il y a trois cas :
     (a) Il existe $t \in Sub_r(\mathcal{R})$ tel que

     $$v \xrightarrow{k-1}_{\tilde{\delta}_1} q_t^T \to_{\tilde{\delta}_1} (q, X)$$

Par induction, nous déduisons que $v \in \zeta^*_{\mathcal{R}}(t)$, et donc, puisque $q_t^T \rightarrow_{\tilde{\delta}_1} (q, X)$ est une (α_2)-règle, la propriété est satisfaite avec un C égal au contexte trivial.

(b) Il existe un état p tel que

$$v \overset{k-1}{\rightarrow}_{\tilde{\delta}_1} (p, X) \rightarrow_{\tilde{\delta}_1} (q, X)$$

Par induction, il s'en suit qu'il existe un seq-contexte C, un terme $t \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \rightarrow t$ dans $\mathcal{R}$ tels que $v \in \zeta^*_{\mathcal{R}}(C[t])$, et $C[Y] \in L_p$. En plus, $(p, X) \rightarrow_{\tilde{\delta}_1} (q, X)$ implique que $p \overset{*}{\rightarrow}_{\delta} q$ (les règles (α_6)), ce qui veut dire que $L_p \subseteq L_q$ et $C[Y] \in L_q$.

(c) Il existe deux termes v_1 et v_2, et deux états p_1 et p_2 tels que

$$v = \cdot(v_1, v_2) \overset{k-1}{\rightarrow}_{\tilde{\delta}_1} \cdot(p_1, p_2) \rightarrow_{\tilde{\delta}_1} (q, X)$$

Les règles α_4b impliquent qu'il existe une règle $\cdot(q_1, q_2) \rightarrow q$ dans δ_{null} telle que $p_1 = (q_1, X)$ et $p_2 = q_2$. Il s'en suit que $v_2 \in L_{q_2}$, et par induction, qu'il existe un seq-contexte C, un terme $t \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \rightarrow t$ dans $\mathcal{R}$ tels que $v_1 \in \zeta^*_{\mathcal{R}}(C[t])$ et $C[Y] \in L_{q_1}$. Soit donc C' le seq-contexte $C'[x] = \cdot(C[x], v_2)$. Nous avons $v \in \zeta^*_{\mathcal{R}}(C'[t])$, et $C'[Y] \in L_q$ (puisque $v_2 \in L_{q_2}$, $C[Y] \in L_{q_1}$, et $\cdot(q_1, q_2) \rightarrow q$ est dans δ_{null}).

Notons que les règles α_4c et α_4f ne peuvent pas être appliquées à ce stade puisque $\tilde{\delta}_1$ ne contient pas encore ces règles.

• $i > 0$. Considérons le cas où $v \overset{*}{\rightarrow}_{\tilde{\delta}_i} q^T$. Nous n'allons pas considérer le cas où $v \overset{*}{\rightarrow}_{\tilde{\delta}_i} (q, X)$, qui peut être traité de la même manière.

Soit donc $v \overset{*}{\rightarrow}_{\tilde{\delta}_i} q^T$ une dérivation de $\tilde{\delta}_i$ sur le terme v. Soit n le nombre de transitions dans $\tilde{\delta}_i \setminus \tilde{\delta}_{i-1}$ appliquées pendant cette dérivation. Dans ce cas, nous dénotons cette dérivation par $v \overset{*}{\rightarrow}^n_{\tilde{\delta}_i} q^T$. Nous procédons par induction sur n :

- $n = 0$, seules les règles de $\tilde{\delta}_{i-1}$ sont appliquées, et nous obtenons le résultat par induction sur i,
- $n > 0$. Il y a deux cas :
  1. Il existe un contexte C et un terme $v_1 \in \mathcal{T}$ tels que $v = C[v_1]$, et

$$v = C[v_1] \overset{*}{\rightarrow}^{n-1}_{\tilde{\delta}_i} C[s'] \rightarrow_{\tilde{\delta}_i} C[s] \overset{k}{\rightarrow}_{\tilde{\delta}_{i-1}} q^T \tag{5.10}$$

Il y a différents cas selon la nature de s et s'. Par exemple, nous pouvons traiter le cas où $s' = {p'}^T$ et $s = p^T$ de la même manière que dans la preuve du lemme 5.3.1. Dans la suite, nous ne considérons que les deux cas suivants, les autres cas peuvent se traiter de la même manière : D'abord, nous considérons le cas où $s' = {p'}^T$ et $s = (p, X)$. Ensuite, nous traitons le cas où $s' = (p', X')$ et $s = (p, X)$.

- $s' = {p'}^T$, et $s = (p, X)$. Alors, selon la règle d'inférence par laquelle la règle de transition ${p'}^T \rightarrow_{\tilde{\delta}_i} (p, X)$ a été créée, nous distingons deux cas :

(a) La règle a été créée par la règle d'inférence α_3b, alors il existe une règle $t_1 \to t$ dans $\mathcal{R}$ telle que $p' = q_t$ et $t_1 \overset{*}{\to}_{\tilde{\delta}_{i-1}} (p, X)$. Nous obtenons par induction sur n que $v_1 \in \zeta^*_{\mathcal{R}}(t)$, et par induction sur i nous déduisons qu'il existe un seq-contexte C', un terme $t' \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \to t'$ dans $\mathcal{R}$ tels que $t_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$ (ce qui veut dire que $v_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$ puisque $v_1 \in \zeta^*_{\mathcal{R}}(t)$ et $t_1 \to t \in \mathcal{R}$), et $C'[Y] \in L_p$.

(b) La règle a été créée par la règle d'inférence α_4c. Soit alors p' un état de Q_{null} tel que $\cdot(p_1, p') \to p$ soit une règle de δ_{null}, et t.q. $0 \overset{*}{\to}_{\tilde{\delta}_{i-1}} (p_1, X)$. Nous obtenons par induction sur i qu'il existe un seq-contexte C_0, un terme $t' \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \to t'$ dans $\mathcal{R}$ tels que $0 \in \zeta^*_{\mathcal{R}}(C_0[t'])$ et $C_0[Y] \in L_{p_1}$. Puisque $v_1 \overset{*}{\to}^{n-1}_{\tilde{\delta}_i} {p'}^T$, nous déduisons par induction sur n que $v_1 \in \zeta^*_{\mathcal{R}}(L_{p'})$. Soit donc $u_1 \in L_{p'}$ tel que $v_1 \in \zeta^*_{\mathcal{R}}(u_1)$. Soit C' le seq-contexte $C'[x] = \cdot(C_0[x], u_1)$. Nous avons que $C'[Y] \in L_p$ puisque $u_1 \in L_{p'}$ et $v_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$ (à cause de la neutralité de 0, puisque $\cdot(0, v_1) \in \zeta^*_{\mathcal{R}}(C'[t'])$).

Dans tous ces cas, il existe un seq-contexte C', un terme $t' \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \to t'$ dans $\mathcal{R}$ tels que $v_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$ et $C'[Y] \in L_p$. Soit donc $u'' \in L_p$ tel que $C'[Y] = u''$. Nous montrons par induction sur k que $v \in \zeta^*_{\mathcal{R}}(L_q)$:

– $k = 1$. Alors C est le contexte trivial, et selon les règles α_4e, il existe un état s tel que $\cdot(s, p) \to q$ est une règle de δ_{null}, et $X \overset{*}{\to}_{\tilde{\delta}_{i-1}} s^T$. Ceci veut dire que $X \in \zeta^*_{\mathcal{R}}(L_s)$. Soit alors $u' \in L_s$ tel que $X \in \zeta^*_{\mathcal{R}}(u')$. Puisque $\cdot(s, p) \to q$ est une règle de δ_{null}, $\cdot(u', u'') \in L_q$. Par conséquent, nous obtenons

$$\cdot(X, C'[Y]) \in \zeta^*_{\mathcal{R}}\big(\cdot(u', u'')\big) \subseteq \zeta^*_{\mathcal{R}}(L_q).$$

Puisque C' est un seq-contexte, en appliquant la règle $X \cdot Y \to t'$ à $\cdot(X, C'[Y])$, nous obtenons $C'[t']$, et puisque $v_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$, nous avons que

$$v = v_1 \in \zeta^*_{\mathcal{R}}\big(\cdot(X, C'[Y])\big) \subseteq \zeta^*_{\mathcal{R}}(L_q)$$

– $k > 1$. Il y a quatre cas :

(a) $v = C[v_1] \overset{*}{\to}_{\tilde{\delta}_i} C[(p, X)] \overset{k-1}{\to}_{\tilde{\delta}_{i-1}} {q'}^T \to_{\tilde{\delta}_{i-1}} q^T$. Par induction sur k, nous déduisons que $v \in \zeta^*_{\mathcal{R}}(L_{q'})$, et les règles α_3a, α_4d, α_5b, α_5c, et α_6 impliquent que $v \in \zeta^*_{\mathcal{R}}(L_q)$.

(b) $v = C[v_1] \overset{*}{\to}_{\tilde{\delta}_i} C[(p, X)] \overset{k-1}{\to}_{\tilde{\delta}_{i-1}} (q', Z) \to_{\tilde{\delta}_{i-1}} q^T$. Ce cas peut être traité comme le cas où $k = 1$.

(c) $v = C[v_1] \overset{*}{\to}_{\tilde{\delta}_i} C[(p, X)] \overset{k-1}{\to}_{\tilde{\delta}_{i-1}} ||({q_1}^T, q_2^T) \to_{\tilde{\delta}_{i-1}} q^T$. Alors il existe un contexte C_1, et un terme v_2 tels que

$$v = C[v_1] = ||(C_1[v_1], v_2) \overset{*}{\to}_{\tilde{\delta}_i} ||(C_1[(p, X)], v_2) \overset{k-1}{\to}_{\tilde{\delta}_{i-1}}$$

$$||({q_1}^T, q_2^T) \to_{\tilde{\delta}_{i-1}} q^T$$

(le cas où $v = C[v_1] = ||(v_2, C_1[v_1])$ est symétrique). Nous déduisons alors par induction sur k que $C_1[v_1] \in \zeta^*_{\mathcal{R}}(L_{q_1})$, et par induction sur i que $v_2 \in \zeta^*_{\mathcal{R}}(L_{q_2})$. Par conséquent, en utilisant les règles α_5a, nous obtenons que $v \in \zeta^*_{\mathcal{R}}(L_q)$.

(d) $v = C[v_1] \xrightarrow{*}_{\tilde{\delta}_i} C[(p, X)] \xrightarrow{k-1}_{\tilde{\delta}_{i-1}} \cdot({q_1}^T, q_2) \rightarrow_{\tilde{\delta}_{i-1}} q^T$. Ce cas peut être résolu comme le cas précédent.

– $s' = (p', X')$, et $s = (p, X)$. Nous procédons comme précédemment. Nous montrons d'abord qu'il existe un seq-contexte C', un terme $t' \in Sub_r(\mathcal{R})$, et une règle $X \cdot Y \rightarrow t'$ dans $\mathcal{R}$ tels que $v_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$, et $C'[Y] \in L_p$. Ensuite, nous procédons par induction sur k pour montrer que $v \in \zeta^*_{\mathcal{R}}(L_q)$.
Par induction sur n, nous déduisons qu'il existe un seq-contexte C'' et une règle $X' \cdot Y' \rightarrow t''$ dans $\mathcal{R}$ tels que $v_1 \in \zeta^*_{\mathcal{R}}(C''[t''])$ et $C''[Y'] \in L_{p'}$. Puisque $(p', X') \rightarrow (p, X)$ est une règle de $\tilde{\delta}_i \setminus \tilde{\delta}_{i-1}$, c'est une règle α_4f. Ceci implique qu'il existe un état $p_1 \in Q_{null}$ tel que $\cdot(p_1, p') \rightarrow p$ est une règle de δ_{null}, et $X' \xrightarrow{*}_{\tilde{\delta}_{i-1}} (p_1, X)$. Nous obtenons par induction sur i qu'il existe un seq-contexte C_0 et une règle $X \cdot Y \rightarrow t'$ dans $\mathcal{R}$ tels que $X' \in \zeta^*_{\mathcal{R}}(C_0[t'])$ et $C_0[Y] \in L_{p_1}$. Soit C' le seq-contexte $C'[x] = \cdot(C_0[x], C''[Y'])$. Nous avons alors que

$$C'[Y] \xrightarrow{*}_{\tilde{\delta}} \cdot(p_1, p') \rightarrow_{\tilde{\delta}} p,$$

et $v_1 \in \zeta^*_{\mathcal{R}}(C'[t'])$. En effet, $v_1 \in \zeta^*_{\mathcal{R}}(C''[t''])$, $C''[t''] \in \zeta_{\mathcal{R}}\big(\cdot(X', C''[Y'])\big)$, et $\cdot(X', C''[Y']) \in \zeta^*_{\mathcal{R}}(C_0[t], C''[Y'])$. Par conséquent, la propriété est satisfaite.

2. Il existe un contexte C tel que $v = C[v_1, v_2]$,

$$v = C[v_1, v_2] \xrightarrow{*}{}^{n}_{\tilde{\delta}_i} C[p_1, p_2] \xrightarrow{k}_{\tilde{\delta}_{i-1}} q^T$$

et il existe $n' < n$ tel que

$$v_1 \xrightarrow{*}{}^{n'}_{\tilde{\delta}_i} p_1 \text{ et } v_2 \xrightarrow{*}{}^{(n-n')}_{\tilde{\delta}_i} p_2$$

Selon la nature des états p_1 et p_2, nous pouvons montrer en utilisant l'hypothèse d'induction (puisque $n' < n$, et $n - n' < n$) que $v \in \zeta^*_{\mathcal{R}}(L_q)$. Pour ce faire, nous procédons comme précédemment par induction sur k.

$\Leftarrow$: Nous montrons maintenant les inclusions inverses. Nous ne montrons que le premier point. Le second peut être montré de manière similaire. Puisque $L_q \subseteq L_{q^T}$ (les règles α_1), il suffit de montrer que pour chaque $t \in L_{q^T}$, si $t' \in \zeta_{\mathcal{R}}(t)$, alors $t' \in L_{q^T}$. Soit alors $t \in L_{q^T}$, c-à-d. $t \xrightarrow{*}_{\tilde{\delta}} q^T$. Le seul cas intéressant est quand nous appliquons une règle de $\zeta_{\mathcal{R}}$ de manière non locale. Si la règle est appliquée localement, nous procédons comme dans la preuve du lemme 5.3.2. Soit alors un contexte C, et un seq-contexte C' tels que $t = C\big[\cdot(X, C'[Y])\big], t' = C\big[C'[u]\big]$, et $X \cdot Y \rightarrow u \in \mathcal{R}$. Soient alors $p_1, p_2 \in \tilde{Q}$ tels que

$$t \xrightarrow{*}_{\tilde{\delta}} C[\cdot(p_1, p_2)] \rightarrow_{\tilde{\delta}} C[p] \xrightarrow{*}_{\tilde{\delta}} q^T$$

Nous montrons dans ce qui suit que $t' \xrightarrow{*}_{\tilde{\delta}} q^T$. Il y a quatre cas selon la nature des états p_1 et p_2.

1. $p_1 = {q'_1}^T, p_2 = q'_2, p = {q'}^T$. Dans ce cas, nous avons que $\cdot(q'_1, q'_2) \to q'$ est une règle δ_{null}. Alors, puisque $X \xrightarrow{*}_{\tilde{\delta}} {q'_1}^T$, il s'en suit que $(q'_2, X) \to_{\tilde{\delta}} {q'}^T$ (les règles α_4e), et nous pouvons montrer par induction structurelle sur C' (en utilisant les règles α_2 pour le cas de base où C' est le contexte trivial, et les règles α_4b pour l'étape d'induction) que $C'[u] \xrightarrow{*}_{\tilde{\delta}} (q'_2, X)$. Ceci implique que :

$$t' = C\big[C'[u]\big] \xrightarrow{*}_{\tilde{\delta}} C\big[(q'_2, X)\big] \to_{\tilde{\delta}} C\big[{q'}^T\big] \xrightarrow{*}_{\tilde{\delta}} q^T$$

2. $p_1 = (q'_1, Z)$, $p_2 = q'_2, p = (q', Z)$. Ce cas est similaire au cas précédent : nous avons nécessairement que $\cdot(q'_1, q'_2) \to q$ est une règle de δ_{null} (les règles α_4b). Puisque $X \xrightarrow{*}_{\tilde{\delta}} (q'_1, Z)$ et $\cdot(q'_1, q'_2) \to q$ est une règle de δ_{null}, il s'en suit que $(q'_2, X) \to_{\tilde{\delta}} (q', Z)$ (les règles α_4f). Nous montrons comme précédemment par induction structurelle sur C' que $C'[u] \xrightarrow{*}_{\tilde{\delta}} (q'_2, X)$. Ceci implique que $C'[u] \xrightarrow{*}_{\tilde{\delta}} (q', Z)$, et donc que :

$$t' = C\big[C'[u]\big] \xrightarrow{*}_{\tilde{\delta}} C\big[(q', Z)\big] \xrightarrow{*}_{\tilde{\delta}} q^T$$

3. $p_1 = {q_{null}}^T, p_2 = p = {q'}^T$ (les règles α_7). Soit s un état tel que

$$C'[Y] \xrightarrow{*}_{\tilde{\delta}} C'[s] \xrightarrow{*}_{\tilde{\delta}} {q'}^T.$$

   Il y a deux cas en fonction de la nature de s :

   (a) Il existe un état q'' tel que $s = {q''}^T$. Pour montrer que $t' \xrightarrow{*}_{\tilde{\delta}} q^T$, il suffit de montrer que $u \xrightarrow{*}_{\tilde{\delta}} {q''}^T$. Ceci découle directement des règles α_3a puisque $X \cdot Y \to u$ est une règle de $\mathcal{R}$, et

$$\cdot(X, Y) \xrightarrow{*}_{\tilde{\delta}} \cdot({q_{null}}^T, {q''}^T) \to_{\tilde{\delta}} {q''}^T.$$

   Par conséquent, nous obtenons que

$$u \xrightarrow{*}_{\tilde{\delta}} q_u^T \to_{\tilde{\delta}} {q''}^T.$$

   (b) Il existe un état q'' et une variable Z tels que $s = (q'', Z)$. Dans ce cas, il suffit de montrer que $u \xrightarrow{*}_{\tilde{\delta}} (q'', Z)$. Soit la variable Z', et les état $\bar{q}$ et $\bar{q}'$ tels que l'on ait

$$Y \xrightarrow{*}_{\tilde{\delta}} \bar{q} \to_{\tilde{\delta}} (\bar{q}', Z') \xrightarrow{*}_{\tilde{\delta}} (q'', Z).$$

   Puisque $Y \xrightarrow{*}_{\tilde{\delta}} \bar{q}$, nous pouvons montrer comme précédemment que $u \xrightarrow{*}_{\tilde{\delta}} \bar{q}$, et donc, il s'en suit que $u \xrightarrow{*}_{\tilde{\delta}} (q'', Z)$.

   (c) $s \in Q_{null}$. Dans ce cas, $\cdot(X, Y) \xrightarrow{*}_{\tilde{\delta}} \cdot({q_{null}}^T, s^T) \to_{\tilde{\delta}} s^T$, et donc les règles α_3a impliquent que $u \xrightarrow{*}_{\tilde{\delta}} s^T$. Nous devons alors montrer par induction structurelle sur C' (comme dans la preuve du lemme 5.3.2) que $C'[u] \xrightarrow{*}_{\tilde{\delta}} {q'}^T$.

4. $p_1, p_2, p \in Q_{null}$. Alors $C'[u] \xrightarrow{*}_{\tilde{\delta}} p^T$, et nous pouvons montrer comme précédemment que $C[p^T] \xrightarrow{*}_{\tilde{\delta}} q^T$ par induction structurelle sur C'.

□

5.6 Accessiblité modulo $\sim$

Nous déduisons à partir des corollaires 5.4.1 et 5.5.2 que pour la classe PAD, nous sommes capables de calculer en temps polynômial des $\sim$-représentants de $Post^*(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$; et des $\sim$-représentants de $Pre^*(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$ qui est $\sim_{||}$-compatible. Notre but dans cette section est de calculer un $\sim$-représentant de l'ensemble $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ pour un langage régulier $\mathcal{L}$ et un PAD $\mathcal{R}$ dans le cas général où l'on n'a pas cette condition de compatibilité. Nous ne pouvons malheureusement pas construire un $\sim$-représentant régulier de cet ensemble. Cependant, nous pouvons construire un $\sim$-représentant non régulier en utilisant des automates d'arbres à compteurs. Heureusement, nous pouvons montrer que le problème du vide des automates à compteurs obtenus par notre construction est décidable. Ceci permet d'utiliser notre construction pour résoudre le problème 2.11. Dans ce qui suit, nous introduisons d'abord cette classe d'automates d'arbres à compteurs, et nous donnons ensuite notre construction.

5.6.1 Automates d'arbres à compteurs

Nous définissons dans cette section une nouvelle classe d'automates à compteurs qui étend les automates d'arbres finis avec des compteurs. Nous utilisons $\vec{c}$ pour représenter le vecteur $(c_1, \ldots, c_m)$, où $c_1, \ldots, c_m$ sont m compteurs d'entiers. Nous dénotons par $dim(\vec{c})$ la dimension, m, de $\vec{c}$.

Définition 5.6.1 *Un* **automate d'arbres à compteurs** *(ou CTA pour Counter Tree Automaton) est un quintuplet $\mathcal{A} = (Q, \Sigma, F, \vec{c}, \delta)$ où Q est un ensemble fini d'états, Σ est un alphabet muni d'une fonction d'arité, $F \subseteq Q$ est un ensemble d'états finaux, et δ est un ensemble de règles de la forme*

$$f\big(q_1, \ldots, q_n\big) \xrightarrow{\mu/\lambda} q \tag{5.11}$$

$$a \xrightarrow{vrai/\lambda'} q \tag{5.12}$$

$$q \xrightarrow{\mu/\lambda} q' \tag{5.13}$$

où $a \in \Sigma_0$, $n \geq 0$, $f \in \Sigma_n$, $q_1, \ldots, q_n, q, q' \in Q$, μ est une formule de Presburger telle que $FV(\mu) = \{c_1, \ldots, c_m\}$, et λ (resp. λ') est une affectation de la forme $\vec{c}' := \vec{c} + \vec{k}$ (resp. de la forme $\vec{c}' := \vec{k}$), où $\vec{c}'$ dénote la nouvelle valeur du vecteur $\vec{c}$, et $\vec{k}$ est un vecteur constant de $\mathbb{Z}^m$.

Dans la suite, la règle $f(q_1, \ldots, q_n) \xrightarrow{\mu/\lambda} q$, où $\mu = vrai$ et λ est l'affectation nulle, sera représentée par $f(q_1, \ldots, q_n) \to q$. Les règles de la forme $f\big(q_1, \ldots, q_n\big) \xrightarrow{vrai/\lambda} q$,

$a \xrightarrow{vrai/\lambda'} q$, ou $q \xrightarrow{vrai/\lambda} q'$ sont appelées règles *non contraintes*, alors que les règles de la forme $f(q_1, \ldots, q_n) \xrightarrow{\mu/\lambda} q$ ou $q \xrightarrow{\mu/\lambda} q'$ telles que $\mu \neq vrai$ sont appelées règles *contraintes*.

Un CTA est un automate d'arbres auquel nous associons un ensemble de compteurs. Les règles (5.11), (5.12), et (5.13) se comportent respectivement comme les règles (2.1), (2.2), et (2.3) de la définition 2.1.20, sauf que dans ce cas, l'automate annote chaque noeud par un état et une valeur des compteurs : Comme dans le cas des automates d'arbres standards, l'automate commence par les feuilles : si une feuille est étiquetée par la lettre a, l'automate utilise les règles 5.12 pour l'annoter par $(q, \vec{k})$. Ensuite, les noeuds internes sont étiquetés en utilisants les règles 5.11 et 5.13 : si un noeud est annoté par $(q, \vec{v})$, et si $q \xrightarrow{\mu/\lambda} q'$ est une règle de l'automate, alors ce noeud peut être annoté par $(q, \vec{v} + \vec{k})$ si $\vec{v} \models \mu$. De même, si les termes $t_1, \ldots, t_n$ sont respectivement annotés par $(q_1, \vec{v_1}), \ldots, (q_n, \vec{v_n})$, et si l'automate comprend la règle $f\big(q_1, \ldots, q_n\big) \xrightarrow{\mu/\lambda} q$, alors si $\vec{v_1} + \cdots + \vec{v_n} \models \mu$, le terme $f\big(t_1, \ldots, t_n\big)$ peut être annoté par $(q, \vec{v_1} + \cdots + \vec{v_n} + \vec{k})$. L'annotation de l'arbre continue ainsi jusqu'à arriver à la racine. Si cette dernière est annotée par un état final, alors l'arbre est accepté par l'automate.

Formellement, soit $\mathcal{A} = (Q, \Sigma, F, \vec{c}, \delta)$ un automate d'arbres à compteurs, nous introduisons un automate d'arbres "standard" $\mathcal{A}_c$ tel que le langage, $\mathcal{L}(\mathcal{A})$, de $\mathcal{A}$ est défini comme étant l'ensemble des termes clos acceptés par $\mathcal{A}_c$. Cet automate est donné par : $\mathcal{A}_c = (Q_c, \Sigma, F_c, \delta_c)$ tel que $Q_c = Q \times \mathbb{Z}^m$, $F_c = F \times \mathbb{Z}^m$, et δ_c contient les règles de transition suivantes :

- $f\big((q_1, \vec{v_1}), \ldots, (q_n, \vec{v_n})\big) \rightarrow (q, \vec{v}) \in \delta_c$ si $f(q_1, \ldots, q_n) \xrightarrow{\mu/\lambda} q$ est dans δ, $\vec{v} = \vec{v_1} + \cdots + \vec{v_n} + \vec{k}$, $\vec{v_1} + \cdots + \vec{v_n} \models \mu$, et λ est de la forme $\vec{c}' := \vec{c} + \vec{k}$,
- $a \rightarrow (q, \vec{v}) \in \delta_c$ si $a \xrightarrow{vrai/\lambda'} q$ est dans δ, $\vec{v} = \vec{k}$, et λ' est une affectation de la forme $\vec{c}' := \vec{k}$,
- $(q, \vec{v}) \rightarrow (q', \vec{v}') \in \delta_c$ si $q \xrightarrow{\mu/\lambda} q'$ est dans δ, $\vec{v}' = \vec{v} + \vec{k}$, $\vec{v} \models \mu$, et λ est de la forme $\vec{c}' := \vec{c} + \vec{k}$.

Nous dénotons par $\rightarrow_\delta$ la relation $\rightarrow_{\delta_c}$.

Théorème 5.6.1 *L'intersection d'un langage régulier d'arbres et d'un langage reconnu par un CTA est un langage reconnu par un CTA.*

Preuve : Soit $\mathcal{A}_1 = (Q_1, \Sigma, F_1, \delta_1)$ un automate d'arbres fini et $\mathcal{A}_2 = (Q_2, \Sigma, F_2, \vec{c}_2, \delta_2)$ un CTA. Alors $\mathcal{L}(\mathcal{A}_1) \cap \mathcal{L}(\mathcal{A}_2)$ est reconnu par le CTA $\mathcal{A}_1 \times \mathcal{A}_2 = (Q_1 \times Q_2, \Sigma, F_1 \times F_2, \vec{c}_2, \delta)$, où δ est l'ensemble des règles :

- $f\big((q_1, q_1'), \ldots, (q_k, q_k')\big) \xrightarrow{\mu/\lambda} (q, q')$ si $f(q_1, \ldots, q_k) \longrightarrow q \in \delta_1$ et $f(q_1', \ldots, q_k') \xrightarrow{\mu/\lambda} q' \in \delta_2$,
- $a \xrightarrow{vrai/\lambda'} (q, q')$ si $a \rightarrow q \in \delta_1$ et $a \xrightarrow{vrai/\lambda'} q' \in \delta_2$
- $(q, q_1') \xrightarrow{\mu/\lambda} (q, q_2')$ si $q_1' \xrightarrow{\mu/\lambda} q_2' \in \delta_2$,
- $(q_1, q) \rightarrow (q_2, q)$ si $q_1 \rightarrow q_2 \in \delta_1$.

□

Il est clair que le problème du vide est indécidable pour les CTAs. Nous introduisons dans ce qui suit une sous classe décidable. Cette classe correspond aux automates où toutes les contraintes non triviales consistent à tester si *tous* les compteurs sont nuls, ou correspondent à des affectations qui sont des reset (affectations qui remettent tous les compteurs à 0).

Définition 5.6.2 *Un* **0-test automate d'arbres à compteurs** *(0-CTA) est un CTA dont les règles sont telles que :*
- *soit l'affectation λ est un reset de la forme $\vec{c} := \vec{0}$;*
- *soit μ est vrai ou une formule de la forme $\vec{c} = \vec{0}$.*

Nous montrons le résultat suivant :

Théorème 5.6.2 *L'intersection d'un langage régulier d'arbres et d'un langage reconnu par un 0-CTA est un langage reconnu par un 0-CTA. En plus, le problème du vide des 0-CTAs est décidable et peut être résolu en un temps polynômial non-déterministe.*

Preuve :

Le fait que l'intersection d'un langage régulier d'arbres et d'un langage reconnu par un 0-CTA est un langage reconnu par un 0-CTA s'obtient directement à partir de la construction du théorème 5.6.1.

Nous montrons maintenant que le problème du vide des 0-CTAs est décidable. Soit $\mathcal{A} = (Q, \Sigma, F, \vec{c}, \delta)$ un 0-CTA. Nous pouvons supposer sans perte de généralité que Q est l'union disjointe de deux ensembles d'états Q_{NT} et Q_T, où un état q est dans Q_{NT} ssi toutes les règles de la forme $f(q_1, \ldots, q_n) \stackrel{\mu/\lambda}{\rightarrow} q$ sont telles que $\mu = vrai$, c-à-d., si ces règles sont non contraintes, et un état q est dans Q_T ssi toutes les règles de la forme $f(q_1, \ldots, q_n) \stackrel{\mu/\lambda}{\rightarrow} q$ sont telles que μ est la contrainte $\vec{c} = \vec{0}$ ou λ est l'affectation $\vec{c} := \vec{0}$, c-à-d., si ces règles sont des règles contraintes du 0-CTA. Un 0-CTA peut être transformé en un 0-CTA de cette forme en dupliquant les états qui interviennent (qui sont la cible) dans les deux types de règles (contraintes et non contraintes).

Le langage accepté par $\mathcal{A}$ est vide ssi l'ensemble Acc des états accessibles de $\mathcal{A}$ ne contient pas un état final (un état q est accessible s'il existe au moins un terme clos t t.q. $t \stackrel{*}{\longrightarrow}_\delta q$). Soit $Acc = Acc^{NT} \cup Acc^T$, où Acc^{NT} et Acc^T sont les ensembles des états accessibles qui sont dans Q_{NT} et Q_T, respectivement. Ces ensembles peuvent être calculés itérativement comme suit :
- $Acc_0^{NT} = \{q \in Q_{NT} \mid \exists a \in \Sigma_0, a \stackrel{vrai/\lambda}{\rightarrow} q \in \delta\}$, $Acc_0^T = \emptyset$,
- $Acc_{i+1}^{NT} = Acc_i^{NT} \cup \{q \in Q_{NT} \mid \exists q_1, \ldots, q_n \in Acc_i, f(q_1, \ldots, q_n) \xrightarrow{vrai/\lambda} q \in \delta\} \cup \{q \in Q_{NT} \mid \exists q' \in Acc_i, q' \xrightarrow{vrai/\lambda} q \in \delta\}$,
- $Acc_{i+1}^T = \{q \in Q_T \mid \mathsf{Proc}(q, Acc_i^T)$ est satisfaisable$\}$, où $\mathsf{Proc}(q, Acc_i^T)$ est une formule linéaire définie ci-dessous.

L'idée est qu'un état q dans Q_T est rajouté à Acc_{i+1}^T ssi il existe un contexte C, n états $q_1, \ldots, q_n$ dans Acc_i^T tels que

$$C[(q_1, \vec{0}), \ldots, (q_n, \vec{0})] \stackrel{*}{\rightarrow}_\delta s \xrightarrow{\vec{c}=\vec{0}/\lambda}_\delta q \tag{5.14}$$

où s est soit de la forme p, soit de la forme $f(p_1, \ldots, p_m)$, où $p, p_1, \ldots, p_m \in Q_{NT}$, $f \in \Sigma$, tels que les dérivations $C[(q_1, \vec{0}), \ldots, (q_n, \vec{0})] \xrightarrow{*}_\delta s$ n'utilisent que des règles non contraintes, c-à-d. que pendant ces dérivations, aucun test n'est réalisé. Ceci est basé sur le fait que les états $q_1, \ldots, q_n$ sont dans Q_T, et donc nous sommes sûrs que si un noeud est annoté par l'un de ces états, alors les compteurs sont nuls à ce noeud. La formule $\mathsf{Proc}(q, Acc_i^T)$ définie ci-dessous est satisfaisable ssi il existe n états $q_1, \ldots, q_n$ dans Acc_i^T , et un contexte C tels que nous avons (5.14), c-à-d., ssi $\mathcal{A}$ admet une exécution ρ qui réalise les dérivations de (5.14)[2].

Nous définissons maintenant la formule $\mathsf{Proc}(q, Acc_i^T)$. Nous associons à chaque règle r de δ la variable w_r qui correspond au nombre d'applications de r dans ρ. D'abord, nous devons exprimer le fait que les w_r sont positifs par la formule POS :

$$\bigwedge_{r \in \delta} w_r \geq 0$$

Pour chaque règle r de δ, nous dénotons par $|r|_p$ le nombre d'occurrences de l'état p dans l'ensemble $\{p_1, \ldots, p_k\}$ si r est une règle de la forme $f(p_1, \ldots, p_k) \xrightarrow{\mu/\lambda} p'$.

Pour chaque état p, nous dénotons par :

– p^+ l'ensemble de règles de la forme $f(p_1, \ldots, p_k) \xrightarrow{\mu/\lambda} p$,

– p^- l'ensemble de règles de la forme $f(p_1, \ldots, p_k) \xrightarrow{\mu/\lambda} p'$ telles que $p \in \{p_1, \ldots, p_k\}$.

Soit δ_T l'ensemble des règles contraintes de δ, c-à-d., des règles étiquetées par $\{\vec{c} = \vec{0}/\lambda\}$ ou $\{\mu/\vec{c} := \vec{0}\}$ où les compteurs sont testés. Nous devons exprimer qu'une seule règle de δ_T est utilisée dans ρ, et que cette règle est nécessairement de la forme $s \xrightarrow{\vec{c}=\vec{0}/\lambda}_\delta q$ ou $s \xrightarrow{\mu/\vec{c}:=\vec{0}}_\delta q$. En plus, puisque $q \in Q_T$, nous devons exprimer le fait qu'il y a une seule règle qui atteind l'état q. Ceci est exprimé par la formule $\mathsf{ONETEST}$:

$$\Big(\sum_{r \in \delta_T} w_r = 1\Big) \wedge \Big(\sum_{r \in q^+} w_r = 1\Big)$$

Ecrivons maintenant les contraintes entre les w_r, ces contraintes doivent exprimer le fait que pour tout état $p \in Q_{NT}$, il y a une conservation du flot, dans le sens que le nombre de fois qu'un état $p \in Q_{NT}$ est créé par les règles $f(p1, \ldots, p_k) \xrightarrow{\mu/\lambda} p$, est égal au nombre de fois qu'il est "consommé" par les règles $f(p_1, \ldots, p, \ldots, p_k) \xrightarrow{\mu/\lambda} p'$. Ceci est exprimé par la formule FLOW :

$$\bigwedge_{p \in Q_{NT}} \Big(\sum_{r \in p^+} w_r = \sum_{r \in p^-} w_r |r|_p\Big)$$

En plus, nous devons exprimer que les seuls états de Q_T, autres que q, qui peuvent intervenir dans ρ sont les états de Acc_i^T, et que ces états ne peuvent qu'être consommés. Ceci peut être exprimé par la formule LEAVES :

$$\bigwedge_{p \in Q_T} \Big(\sum_{r \in p^-} w_r > 0\Big) \Leftrightarrow p \in Acc_i^T$$

[2]Pour simplifier, nous ne considérons ici que le cas où l'étiquette de la règle contrainte appliquée pour atteindre q est de la forme $\vec{c} = \vec{0}/\lambda$. Le cas où elle est de la forme $\mu/\vec{c} := \vec{0}$ est similaire.

Nous devons exprimer maintenant le fait que, pour être impliqué dans une "vraie" exécution ρ, un état $p \in Q_{NT}$ doit être accessible dans ρ. En effet, si nous considérons la règle $r = f(p) \xrightarrow{vrai/\lambda} p$, et si nous supposons que p n'apparaît jamais pendant l'exécution ρ, alors les contraintes précédentes restent satisfaites même si nous associons à w_r n'importe quelle valeur. Ceci ne doit pas être le cas, puisque nous voulons que les valeurs des w_r correspondent aux nombres d'applications des règles r pendant l'exécution ρ. Pour contourner ce problème, nous devons imposer que w_r soit positif ssi r est de la forme $f(p_1, \ldots, p_k) \xrightarrow{\mu/\lambda} p$, où tous les états $p_1, \ldots, p_k$ sont accessibles dans l'exécution ρ. Ceci est exprimé par la formule REACH ci-dessous. Nous allons d'abord expliquer cette formule.

Soit ρ une exécution de $\mathcal{A}$ qui arrive à la racine de v dans l'état p, où $p \in Q_{NT}$ (ρ n'utilise que des règles non contraintes). Considérons un chemin de v tel qu'il existe deux positions $p_1 < p_2$ sur ce chemin annotées par le même état p' pendant l'exécution ρ. Soit u (resp. u') le sous terme de v à la position p_2 (resp. p_1). Alors, il existe deux contextes C et C' tels que $u' = C'[u]$, et $v = C[u']$. Il est clair que, si nous enlevons le contexte C', nous obtenons un terme $v' = C[u]$ dont la racine peut être annotée avec l'état p par l'automate $\mathcal{A}$. Ceci est dû au fait que $p \notin Q_T$, et l'automate n'utilise que des règles non contraintes pendant les dérivations. Par conséquent, les valeurs des compteurs ne sont pas importantes puisque pendant l'exécution aucun test n'a été réalisé. Ainsi, si p est accessible dans une exécution ρ, alors il l'est aussi dans une autre exécution ρ' (obtenue à partir de ρ) dont les chemins ne contiennent pas deux positions annotées par le même état, de telles exécutions ρ' sont appelées exécutions *élémentaires* . Etant donné un état p, nous définissons Θ_p comme étant l'ensemble des exécutions élémentaires θ qui arrivent à l'état p après application de θ. Notons que pour chaque $p \in Q_{NT}$, Θ_p est fini et peut être effectivement calculé. Nous définissons alors la formule REACH suivante :

$$\bigwedge_{p \in Q_{NT}} \Big(\sum_{r \in p^-} w_r > 0 \Big) \Leftrightarrow \Big(\bigvee_{\theta \in \Theta_p} \bigwedge_{r \in \theta} w_r > 0 \Big)$$

Finalement, si r' est la règle qui permet l'annotation de la racine par l'état q, il faut exprimer par une formule COUNT que l'accumulation des effets des différentes règles de ρ permet l'application de r'. Il y a deux cas :

- c'est une règle r' étiquetée par $\{\vec{c} = \vec{0}/\lambda\}$ qui a été utilisée. Dans ce cas, la valeur de chaque compteur est nulle, et est égale à l'accumulation des différents effets des différentes règles qui ont été appliquées pendant l'exécution ρ. Ceci est exprimé par la formule COUNT :

$$\vec{0} = \vec{c} = \sum_{r \in \delta} w_r \vec{k_r}$$

  où $\vec{c'} := \vec{c} + \vec{k_r}$ est l'affectation correspondant à la règle r.
- c'est une règle r' étiquetée par $\{\mu/\vec{c} := \vec{0}\}$ qui a été utilisée. Dans ce cas, les valeurs des compteurs satisfont μ. Ces compteurs sont égaux à l'accumulation des différents effets des différentes règles qui ont été appliquées pendant l'exécution

ρ. Ceci est exprimé par la formule COUNT :

$$\sum_{r \in \delta} w_r \vec{k_r} \models \mu$$

où $\vec{c}' := \vec{c} + \vec{k_r}$ est l'affectation correspondant à la règle r.
Enfin, la formule $\mathsf{Proc}(q, Acc_i^T)$ est donnée par :

$$\mathsf{Proc}(q, Acc_i^T) = \mathsf{POS} \wedge \mathsf{ONETEST} \wedge \mathsf{LEAVES} \wedge \mathsf{FLOW} \wedge \mathsf{REACH} \wedge \mathsf{COUNT}.$$

Nous pouvons décider la satisfaisabilité de cette formule linéaire en un temps NP comme suit : Toutes les formules $\mathsf{POS}, \mathsf{ONETEST}, \mathsf{LEAVES}, \mathsf{FLOW}$, et COUNT ont une taille polynômiale, seule la formule REACH peut avoir une taille exponentielle puisqu'il y a un nombre exponentiel de chemins élémentaires. Au lieu de considérer toute cette formule, nous pouvons deviner, pour chaque état q, le chemin élémentaire θ' qui permet d'accéder à q, et remplacer REACH par la formule polynômiale :

$$\bigwedge_{p \in Q_{NT}} \Big(\sum_{r \in p^-} w_r > 0 \Big) \Leftrightarrow \Big(\bigwedge_{r \in \theta'} w_r > 0 \Big).$$

Donc, nous obtenons une formule linéaire de taille polynômiale. La satisfaisabilité de cette formule peut être décidée dans NP (voir le théorème 2.1.8).

□

5.6.2 Calcul d'un $\sim$-représentant de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ pour PAD

Notre but maintenant est de calculer un $\sim$-représentant de l'ensemble $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ pour un langage régulier $\mathcal{L}$ et un PAD $\mathcal{R}$. Pour ce faire, nous définissons une transformation $\wp_{\mathcal{R}}$ telle que pour tout langage $\mathcal{L}$ et tout PRS $\mathcal{R}$, $\wp^*_{\mathcal{R}}(\mathcal{L})$ est un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$ (et donc $\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})$ est un $\sim$-représentant de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$), et nous construisons un 0-CTA qui reconnaît $\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$ et PAD $\mathcal{R}$. Nous supposons que les PAD que nous considérons ne contiennent pas de règles de la forme $0 \to t$. Ceci n'est pas restrictif. En effet, les PRS obtenus à partir des programmes ne contiennent pas ces règles (voir la section 4.4). De plus, les systèmes PRS tels que définis par Mayr ne comprennent pas ces règles. Jusqu'ici nous avons considéré ce type de règles pour traiter de manière uniforme les problèmes du calcul des représentants des ensembles $Post^*_{\mathcal{R},\equiv}(\mathcal{L})$ et $Pre^*_{\mathcal{R},\equiv}(\mathcal{L})$ en exploitant le fait que $Pre^*_{\mathcal{R},\equiv}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1},\equiv}(\mathcal{L})$. Donc si $\mathcal{R}$ contient la règle $t \to 0$ qui modélise la terminaison du processus t, $\mathcal{R}^{-1}$ doit contenir la règle $0 \to t$. Dans cette section, nous n'avons pas besoin de considérer ce type de règles puisque nous ne nous intéressons qu'au calcul des représentants de $Pre^*_{\mathcal{R},\sim}(\mathcal{L})$ pour un PAD $\mathcal{R}$. Nous verrons plus tard pourquoi nous avons besoin de cette restriction.

Soit $\mathcal{R}$ un PRS quelconque, pour définir la transformation $\wp_{\mathcal{R}}$, nous avons besoin de la notion de *paral-contexte* :

Définition 5.6.3 *Un* paral-contexte *est un contexte $C[x_1, \ldots, x_n]$ tel que tous les ancêtres des variables $x_1, \ldots, x_n$ sont étiquetés par "$||$".*

Modulo $\sim$, la difficulté majeure vient des règles $X||Y \to t$ qui peuvent être appliquées de manière non locale. Plus précisément, modulo $\sim$, cette règle peut être appliquée à tous les termes de la forme $C[X, Y]$, où $C[x_1, x_2]$ est un paral-contexte, pour produire un terme qui est $\sim$-équivalent à $C[0, t]$. Pour des raisons techniques, nous introduisons une nouvelle constante spéciale "$\tilde{0}$" qui est considérée comme $\sim$-équivalente au processus nul 0, et nous considérons le terme $C[\tilde{0}, t]$ (qui est $\sim$-équivalent à $C[0, t]$). Soit alors $\mathcal{T}'$ l'ensemble des termes de processus contenant "$\tilde{0}$". La différence entre "$\tilde{0}$" et "0" est que "$\tilde{0}$" n'est jamais ni simplifié ni réécrit. Nous introduisons ce nouveau symbole pour pouvoir préserver la structure de l'arbre après l'application non locale d'une règle de la forme $X||Y \to t$. Nous devons donc tenir compte de la présence de ces "$\tilde{0}$" dans les termes comme suit : D'abord, nous définissons un *null-contexte* :

Définition 5.6.4 *Un* null-contexte *est un contexte à une variable $C[x]$ tel que toutes les feuilles autres que la variable x sont étiquetées par "$\tilde{0}$".*

Ceci veut dire que si C_0 est un null-contexte, alors $C_0[X]$ est $\sim$-équivalent à X. Par conséquent, si C est un paral-contexte et C_0, C_0' sont des null-contextes, alors la règle $X||Y \to t$ doit être appliquée à tous les termes de la forme $C\big[C_0[X], C_0'[Y]\big]$ pour produire un terme $\sim$-équivalent à $C\big[C_0[\tilde{0}], C_0'[t]\big]$. De la même manière, si C_s est un seq-contexte, alors une règle de la forme $X \cdot Y \to t$ doit être appliquée à chaque terme de la forme $\cdot(C_0[X], C_s[Y])$, pour produire un terme $\sim$-équivalent à $\cdot(C_0[\tilde{0}], C_s[t])$. Observons que le fils droit $C_s[Y]$ ne contient pas de $\tilde{0}$. Ceci est dû à la stratégie de réécriture préfixe du "$\cdot$" et au fait que les $\tilde{0}$ sont absents du langage initial.

Nous définissons maintenant la relation $\wp_{\mathcal{R}}$ qui réalise ces transformations pour un PRS arbitraire $\mathcal{R}$. Cette relation est définie comme la plus petite relation sur $\mathcal{T}'$ qui contient $\zeta_{\mathcal{R}}$ (puisque $\sim$ inclut $\sim_s$) et telle que pour tout contexte C' nous avons :

(A1) Pour toute règle $X||Y \to t$ dans $\mathcal{R}$, s'il existe un paral-contexte C et deux null-contextes C_0 et C_0' tels que $t = C'\Big[C\big[C_0[X], C_0'[Y]\big]\Big]$ et

$$t' = C'\Big[C\big[C_0[\tilde{0}], C_0'[t]\big]\Big]$$

alors $(t, t') \in \wp_{\mathcal{R}}$.

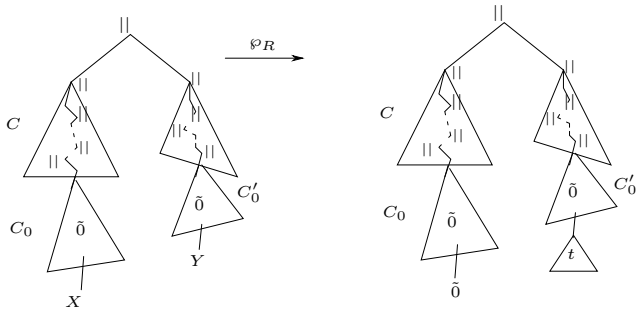


FIG. 5.4 – La règle $X||Y \to t$ modulo $\sim$

(A2) Pour toute règle $X \cdot Y \to t$ dans $\mathcal{R}$, s'il existe un seq-contexte C et un null-contexte C_0 tels que $t_1 = C'\big[\cdot\,(C_0[X], C[Y])\big]$ et $t_2 = C'\big[\cdot\,(C_0[\tilde{0}], C[t])\big]$, alors $(t_1, t_2) \in \wp_{\mathcal{R}}$.

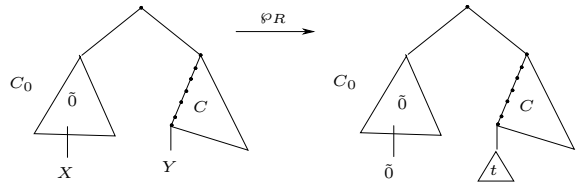


FIG. 5.5 – La règle $X \cdot Y \to t$ modulo $\sim$

Pour tout langage $\mathcal{L}$, $\wp^*_{\mathcal{R}}(\mathcal{L})$ est un $\sim$-représentant de $Post^*_{\mathcal{R},\sim}(\mathcal{L})$:

Proposition 5.6.1 *Pour tout langage d'arbres $\mathcal{L}$ et tout PRS $\mathcal{R}$, $Post^*_{\mathcal{R},\sim}(\mathcal{L}) = [\wp^*_{\mathcal{R}}(\mathcal{L})]_\sim$ et $Pre^*_{\mathcal{R},\sim}(\mathcal{L}) = [\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})]_\sim$.*

Notre but est alors de calculer $\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})$ pour un PAD $\mathcal{R}$ et un langage régulier d'arbres $\mathcal{L}$. Nous montrons le résultat suivant :

Théorème 5.6.3 *Soient $\mathcal{L}$ un langage régulier d'arbres et $\mathcal{R}$ un PAD, alors $\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})$ peut être effectivement caractérisé par un 0-CTA. En plus, à partir d'un automate à k états et τ transitions qui reconnaît $\mathcal{L}$, il est possible de construire un 0-CTA acceptant $\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})$ ayant $\mathcal{O}\big(2^{|Var|^2} \cdot |Var|^2 \cdot (k + |Sub_r(\mathcal{R}^{-1})|)\big)$ états, et $\mathcal{O}\big(2^{|Var|^2} \cdot |Var|^4 \cdot (k + |Sub_r(\mathcal{R}^{-1})|) \cdot |Sub_r(\mathcal{R}^{-1})| + k \cdot |Var|^2 \cdot 4^{|Var|^2}\big)$ transitions.*

Tout le reste de cette section est consacré à l'explication et la preuve de ce théorème. Nous fixons alors un PAD $\mathcal{R}$ et langage régulier d'arbres $\mathcal{L}$.

5.6.3 Principe de la construction

Avant de donner la construction de l'automate à compteurs, nous expliquons son idée principale qui est basée sur deux cas :
Cas B1 : Voyons d'abord comment les règles de la forme $X||Y \to t$ sont traitées. Soient u et u' deux termes tels que $u' \in \wp^*_{\mathcal{R}^{-1}}(u)$ et il existe un paral-contexte C, n variables de processus $A_1, \ldots, A_n$, n termes $t_1, \ldots, t_n$, et n null-contextes $C'_1, \ldots, C'_n$ tels que $u = C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]$, et $u' = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$. u' est obtenu de u après plusieurs réécritures dans les feuilles comme suit : Il existe des termes intermédiaires $u'_0, \ldots, u'_k$, et une séquence de paires d'indices appartenant à $\{1, \ldots, n\}$: $(i_1, j_1), \ldots, (i_k, j_k)$ où ces indices ne sont pas nécessairement distincts, mais pour chaque $l \leq k$, $i_l \neq j_l$, tels que $u'_0 = u$, $u'_k = u'$, et u'_{l+1} est un successeur de u'_l par $\wp_{\mathcal{R}^{-1}}$ obtenu comme suit : Si u'_l est de la forme $u'_l = C\big[C'_1[s_1], \ldots, C'_n[s_n]\big]$, où les s_i sont des termes, alors il existe dans $\mathcal{R}^{-1}$ une règle de la forme $Z_l||Y_l \to t$ (ou $Y_l||Z_l \to t$) telle que $s_{i_l} = Y_l$, $s_{j_l} = Z_l$, et $u'_{l+1} = C\big[C'_1[s'_1], \ldots, C'_n[s'_n]\big]$, où $s'_{i_l} = \tilde{0}$,

$s'_{j_l} = t$, et $s'_i = s_i$ pour les autres indices. Ceci veut dire que u'_{l+1} est obtenu en appliquant la règle $Z_l||Y_l \to t$ (ou $Y_l||Z_l \to t$) aux positions (i_l, j_l) dans le terme u'_l. Observons que t est soit égal à u_{j_l}, soit égal à une variable de processus B qui sera réécrite plus tard à une étape ultérieure $l' > l$ (ceci veut dire qu'il existe une règle $B||B' \to s$ (ou $B'||B \to s$) qui peut être appliquée aux positions $(i_{l'}, j_{l'})$, où j_l est soit égal à $i_{l'}$, soit à $j_{l'}$.

L'automate doit reconnaître le terme u' comme un successeur de u. Il commence par annoter les feuilles et remonte jusqu'à la racine. A chaque position i_l (resp. j_l), il doit deviner que ce nœud a interagit avec le nœud j_l (resp. i_l) comme décrit ci-dessus. Il doit mémoriser toutes ces prédictions et les valider quand il arrive à la racine de u'. Pour mémoriser toutes ces prédictions (il y en a un nombre non borné), l'automate se comporte de la manière suivante : Il considère un compteur c_X pour chaque variable de processus X dans Var, et quand il devine qu'aux positions (i_l, j_l) une règle $Z_l||Y_l \to t$ (ou $Y_l||Z_l \to t$) a été appliquée (en transformant le Y_l de la position i_l en $\tilde{0}$, et le Z_l de la position j_l en t), il décrémente le compteur c_{Y_l} à la position i_l et l'incrémente à la position j_l. En plus, il doit mémoriser à chaque position $p \in \{1, \ldots, n\}$ un graphe G_p dont les arêtes sont dans $Var \cup \{\bot\} \times Var$ tel que :

- $\bot \to Y_l$ est dans G_p ssi $p = j_l$ et $u_p \neq \tilde{0}$,
- $Y_l \to Y_{l'}$ est dans G_p ssi $p = i_l = j_{l'}$ (donc nécessairement $u_p = \tilde{0}$).

Après avoir fait toutes ces prédictions aux feuilles, l'automate remonte ce qu'il a deviné à la racine en annotant les nœuds internes comme suit : Le graphe G_p et la valeur des compteurs devinés à la position p sont remontés jusqu'à la racine du contexte C'_p, tout en vérifiant que C'_p est un null-contexte. Ensuite, ces informations sont remontées à la racine de C de la manière suivante : si le terme v_1 (resp. v_2) est décoré par le graphe G_1 (resp. G_2) et la valeur $\vec{c}_1$ des compteurs (resp. $\vec{c}_2$) (nous considérons le vecteur des compteurs $\vec{c} = (c_{X_1}, \ldots, c_{X_m})$, où $Var = \{X_1, \ldots, X_m\}$), alors $||(v_1, v_2)$ est décoré par $G_1 \cup G_2$ et $\vec{c} = \vec{c}_1 + \vec{c}_2$. Notons qu'il existe un nombre fini de tels graphes (au plus $2^{O(|Var|^2)}$ graphes possibles).

Le point clé de la construction est que nous montrons que les prédictions faites par l'automate aux feuilles sont correctes ssi l'automate atteint la racine avec des compteurs nuls ($\vec{c} = \vec{0}$), et un graphe G ayant un chemin allant du sommet $\bot$ à chaque sommet de Var intervenant dans au moins une arête de G (lemmes 5.6.2 et 6.5.2).

De manière plus précise, soit $\vec{c}$ un vecteur de $|Var|$ compteurs ($dim(\vec{c}) = |Var|$). Pour associer un compteur à chaque variable de processus X, nous numérotons les variables de processus de Var de 1 à $|Var|$. Soit donc $Var = \{X_1, X_2, \ldots, X_{|Var|}\}$. Soit X un élément quelconque de Var, nous notons par $indice(X)$ l'indice j tel que $X = X_j$. La i^{ème} composante de $\vec{c}$ correspond au compteur de la variable X_i. Soient $\vec{k}_i$ (resp. $\vec{k}'_i$) des vecteurs de dimension $|Var|$ ayant 1 (resp. -1) à la i^{ème} composante et des 0 dans les autres composantes, et soit λ_i (resp. λ'_i) l'affectation $\vec{c}' := \vec{c} + \vec{k}_i$ (resp. $\vec{c}' := \vec{c} + \vec{k}'_i$). Notons que λ_i (resp. λ'_i) incrémente (resp. décrémente) le compteur correspondant à la variable X_i.

Soit $\mathcal{G} = 2^{Var \cup \{\bot\} \times Var}$ l'ensemble des graphes ayant des arêtes dans $Var \cup \{\bot\} \times Var$. Pour valider les prédictions, nous avons besoin de définir la sous classe $\mathcal{G}_c$ des graphes G de $\mathcal{G}$ ayant un chemin partant du sommet $\bot$ à chaque sommet de Var intervenant dans au moins une arête de G. Soit alors un graphe $G \in \mathcal{G}$. Nous définissons

$Noeuds(G) = \{X \in Var \mid \exists A \to X \in G \text{ ou } X \to A \in G\}$ l'ensemble des sommets de Var intervenants dans au moins une arête de G. $\mathcal{G}_c$ est l'ensemble des graphes G ayant un chemin partant de $\perp$ à chaque variable dans $Noeuds(G)$:

$$\mathcal{G}_c = \{G \in \mathcal{G} \mid \forall X \in Noeuds(G), \exists \text{ un chemin dans } G \text{ menant de } \perp \text{ à } X\}.$$

Soit alors $\left(Y_j^p||Z_j^p \to t_j^p \text{ (ou } Z_j^p||Y_j^p \to t_j^p \text{)}\right)_{1\leq j\leq h_p}$ la séquence de règles de $\mathcal{R}^{-1}$ qui ont été appliquées de manière non locale à la position p, pour $p \in \{1, \ldots, n\}$, telle que la règle d'indice j soit la j^{ème} règle qui a été appliquée à la position p de manière non locale. La même règle peut apparaître plusieurs fois dans la séquence, puisqu'une règle peut s'appliquer plusieurs fois à une position donnée. Nous considérons par convention de notation que si $Y_j^p||Z_j^p \to t_j^p$ ou $Z_j^p||Y_j^p \to t_j^p$ s'applique entre deux positions (dont l'une est p), alors c'est la position étiquetée par Y_j^p qui se transforme en $\tilde{0}$ (c'est pour cette raison que nous considérons deux formes de règles qui sont toutes les deux symétriques : (1) $Y_j^p||Z_j^p \to t_j^p$ et (2) $Z_j^p||Y_j^p \to t_j^p$ pour exprimer que si une règle $X||Y \to t$ a été appliquée à la position p telle que c'est X qui s'est transformé en $\tilde{0}$, alors cette règle est de la forme $Y_j^p||Z_j^p \to t_j^p$. Si c'est le Y qui s'est transformé en $\tilde{0}$, alors cette règle est de la forme $Z_j^p||Y_j^p \to t_j^p$). Si nous reprenons les séquences (u'_l) et (i_l, j_l), $1 \leq l \leq k$; ceci veut dire que pour chaque j, $1 \leq j \leq h_p$, il existe un indice $k_{j,p}$, $1 \leq k_{j,p} \leq k$ tel que $k_{j,p} < k_{j+1,p}$, $u'_{k_{j,p}} = C\big[C'_1[s_1], \ldots, C'_n[s_n]\big]$, et $u'_{k_{j,p}+1} = C\big[C'_1[s'_1], \ldots, C'_n[s'_n]\big]$ tels que $s_{i_{k_{j,p}}} = Y_j^p$, $s_{j_{k_{j,p}}} = Z_j^p$, $s'_{i_{k_{j,p}}} = \tilde{0}$, $s'_{j_{k_{j,p}}} = t_j^p$, $s'_i = s_i$ pour les autres indices, et $p = j_{k_{j,p}}$ ou $p = i_{k_{j,p}}$. Plus précisément :

- si $j \neq h_p$, alors $p = j_{k_{j,p}}$ puisque s'_p doit être différent de $\tilde{0}$ pour que les autres règles d'indice supérieur à j puissent être appliquées à la position p. De plus, si $j = 1$ alors $s_{j_{k_{1,p}}} = Z_1^p = A_p$: pour que la première règle puisse être appliquée à la position p, il faut que A_p soit égale à Z_1^p ; sinon, si $j > 1$ alors $Z_j^p \in \zeta^*_{\mathcal{R}^{-1}}(t_{j-1}^p)$: après chaque application d'une règle $Y_{j-1}^p||Z_{j-1}^p \to t_{j-1}^p$ ou $Z_{j-1}^p||Y_{j-1}^p \to t_{j-1}^p$, Z_{j-1}^p se transforme en t_{j-1}^p. Après, pour pouvoir appliquer la règle $Y_j^p||Z_j^p \to t_j^p$ ou $Z_j^p||Y_j^p \to t_j^p$ à cette position, il faut que t_{j-1}^p se transforme d'abord en Z_j^p par les règles de $\zeta_{\mathcal{R}^{-1}}$.
- Si $j = h_p$, alors :
  - si $u_p \neq \tilde{0}$, alors $p = j_{k_{h_p,p}}$, $Z_{h_p}^p \in \zeta^*_{\mathcal{R}^{-1}}(t_{h_p-1}^p)$, et $u_p \in \zeta^*_{\mathcal{R}^{-1}}(t_{h_p}^p)$: après l'application de la dernière règle, nous obtenons $t_{h_p}^p$ qui doit alors se transformer en u_p par $\zeta_{\mathcal{R}^{-1}}$. De plus, comme dans le cas précédent, si $h_p = 1$ alors $s_{j_{k_{1,p}}} = Z_1^p = A_p$;
  - si $u_p = \tilde{0}$, alors $p = i_{k_{j,p}}$ et si $h_p = 1$ alors $s_{i_{k_{1,p}}} = Y_1^p = A_p$ (il y a une seule règle qui est appliquée à la position p pour la réécrire en $\tilde{0}$), sinon $Y_{h_p}^p \in \zeta^*_{\mathcal{R}^{-1}}(t_{h_p-1}^p)$: après l'application de l'avant dernière règle, nous obtenons $t_{h_p-1}^p$ qui doit alors se transformer en $Y_{h_p}^p$ par $\zeta_{\mathcal{R}^{-1}}$. La dernière règle transforme alors ce $Y_{h_p}^p$ en $\tilde{0}$.

Soit $l_{j,p} = indice(Y_j^p)$. L'automate décore chaque position $p \in \{1, \ldots, n\}$ par un graphe G_p et une valeur des compteurs v_p tels que :

- Si $u_p \neq \tilde{0}$, alors

- $v_p = \sum_{1 \leq j \leq h_p} \vec{k}_{l_{j,p}}$: puisque pour chaque j, $1 \leq j \leq h_p$, $p = j_{k_{j,p}}$. A la position p l'automate doit donc incrémenter à chaque fois le compteur de Y_j^p, ce qui revient à rajouter à chaque fois le vecteur $\vec{k}_{l_{j,p}}$;
- $G_p = \{\bot \rightarrow Y_j^p \mid 1 \leq j \leq h_p\}$: puisque $u_p \neq \tilde{0}$ et pour chaque j, $1 \leq j \leq h_p$, $p = j_{k_{j,p}}$. Le "$\bot$" indique que ce nœud est différent de $\tilde{0}$.

- Si $u_p = \tilde{0}$, alors :
  1. Si $h_p > 1$, puisque pour $j < h_p$, $p = j_{k_{j,p}}$ et $p = i_{k_{h_p,p}}$, nous avons :
     - $v_p = \sum_{1 \leq j \leq h_p - 1} \vec{k}_{l_{j,p}} + \vec{k}'_{l_{h_p,p}}$: L'automate doit incrémenter les compteurs de Y_j^p pour $1 \leq j \leq h_p - 1$, et décrémenter celui de $Y_{h_p}^p$;
     - $G_p = \{Y_{h_p}^p \rightarrow Y_j^p \mid 1 \leq j \leq h_p - 1\}$;
  2. Si $h_p = 1$, alors il y a une seule règle qui s'est appliquée (de manière non locale) à cette position et l'a transformée en $\tilde{0}$. Nous avons seulement que $p = i_{k_{1,p}}$, et donc :
     - $v_p = \vec{k}'_{l_{h_p,p}}$: L'automate doit décrémenter le compteur de $Y_{h_p}^p$;
     - $G_p = \emptyset$.

Comme nous l'avons décrit précédemment, l'automate remonte ces prédictions des feuilles jusqu'à la racine en mémorisant à chaque fois l'union des graphes et la somme des compteurs. Par conséquent, la racine du contexte C est décorée par le graphe $G = \bigcup_{1 \leq p \leq n} G_p$ et la valeur du compteur $v = \sum_{1 \leq p \leq n} v_p$. Il est facile de vérifier que $\bigcup_{1 \leq p \leq n} G_p \in \mathcal{G}_c$ et que $\sum_{1 \leq p \leq n} v_p = \vec{0}$ (par induction sur le nombre de $\tilde{0}$ dans le multiensemble $\{\!\{u_1, \ldots, u_n\}\!\}$). De plus, nous montrons que si $G \in \mathcal{G}_c$ et $v = \vec{0}$, alors les prédictions de l'automate sont correctes. Pour ce faire, nous définissons d'abord la notion de *derived system* qui résume tout ce qui précède comme suit :

Définition 5.6.5 *Un* derived system *est un sytème* $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$ *tels que pour chaque* i, $1 \leq i \leq n$, A_i *est une variable de processus,* u_i *est un terme,* $G_i \in \mathcal{G}$, v_i *est une valuation de* $\vec{c}$, *et* $\mathcal{S}_i$ *est une séquence de règles de* $\mathcal{R}^{-1}$ *de la forme*

$$\left(Y_j^i || Z_j^i \rightarrow t_j^i \text{ (ou } Z_j^i || Y_j^i \rightarrow t_j^i \text{)}\right)_{1 \leq j \leq h_i}$$

tels que (nous posons $l_{j,i} = indice(Y_j^i)$*) :*

- *Si* $u_i \neq \tilde{0}$, *alors :*
  - $A_i = Z_1^i$,
  - $v_i = \sum_{1 \leq j \leq h_i} \vec{k}_{l_{j,i}}$,
  - $G_i = \{\bot \rightarrow Y_j^i \mid 1 \leq j \leq h_i\}$,
  - $\forall 1 \leq j < h_i$, $Z_{j+1}^i \in \zeta^*_{\mathcal{R}^{-1}}(t_j^i)$,
  - $u_i \in \zeta^*_{\mathcal{R}^{-1}}(t_{h_i}^i)$.
- *Si* $u_i = \tilde{0}$, *alors :*
  1. *Si* $h_i > 1$, *alors :*

- $A_i = Z^i_1$,
- $v_i = \sum\limits_{1 \leq j \leq h_i - 1} \vec{k}_{l_{j,i}} + \vec{k}'_{l_{h_i,i}}$,
- $G_i = \{Y^i_{h_i} \to Y^i_j \mid 1 \leq j \leq h_i - 1\}$,
- $\forall 1 \leq j < h_i - 1$, $Z^i_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t^i_j)$,
- $Y^i_{h_i} \in \zeta^*_{\mathcal{R}^{-1}}(t^i_{h_i - 1})$.

2. *Si* $h_i = 1$, *alors :*
   - $A_i = Y^i_1$,
   - $v_i = \vec{k}'_{l_{h_i,i}}$,
   - $G_i = \emptyset$.

Nous montrons que (voir le lemme 5.6.1) si $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$ est un derived system tel que $\bigcup\limits_{1 \leq i \leq n} G_i \in \mathcal{G}_c$ et $\sum\limits_{1 \leq i \leq n} v_i = \vec{0}$, alors

$$u' = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big] \in \wp^*_{\mathcal{R}^{-1}}\Big(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]\Big).$$

Avant de prouver ce résultat, nous montrons d'abord comment sont traitées les règles de la forme $X \cdot Y \to t$.

Cas B2 : Supposons que les termes u et u' ci-dessus sont tels qu'il existe un indice m, $1 \leq m \leq n$ tel que $u_m = X$ et $\forall i \neq m$, $u_i = \tilde{0}$. Soit $C_s[x]$ un seq-contexte. Il est clair que $\cdot(u', C_s[Y]) \in \wp^*_{\mathcal{R}^{-1}}\big(\cdot\,(u, C_s[Y])\big)$ puisque $u' \in \wp^*_{\mathcal{R}^{-1}}(u)$. De plus, puisque u' est $\sim$-équivalent à X, la règle $X \cdot Y \to t$ peut être appliquée au terme $\cdot(u', C_s[Y])$ qui est égal à

$$\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_{m-1}[\tilde{0}], C'_m[X], C'_{m+1}[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[Y])$$

comme décrit sur la figure 5.5 pour donner $\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t])$. Il s'en suit que

$$\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t]) \in \wp^*_{\mathcal{R}^{-1}}\big(\cdot\,(u', C_s[Y])\big).$$

Si nous reprenons les séquences (u'_l) et (i_l, j_l), $1 \leq l \leq k$; les indices $p \neq m$ vérifient les mêmes conditions que précédemment. Pour l'indice m, pour chaque j, $1 \leq j \leq h_m$, nous avons $m = j_{k_{j,m}}$, puisque le $\tilde{0}$ à cette position est dû à l'application de la règle $X \cdot Y \to t$ et non aux règles de la forme $X'||Y' \to t'$. De plus :

- $A_m = Z^m_1$;
- $\forall j < h_m$, $Z^i_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t^m_j)$ (comme précédemment) ;
- $X \in \zeta^*_{\mathcal{R}^{-1}}(t^m_{h_m})$: après avoir appliqué la dernière règle, $t^m_{h_m}$ se transforme en X par $\zeta_{\mathcal{R}^{-1}}$.

Pour reconnaître le terme $\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t])$ comme successeur de $\cdot(u, C_s[Y])$ par $\wp_{\mathcal{R}^{-1}}$, l'automate procède comme suit : Il devine tout d'abord qu'à la position m il y avait un X qui a été remplacé par un $\tilde{0}$ en appliquant la règle $X \cdot Y \to t$. Ce nœud est alors décoré par le label "$-X$" pour dire qu'à cette place un X a disparu en appliquant une règle de la forme ci-dessus. Ensuite, l'automate doit deviner qu'aux positions (i_l, j_l) une règle $Z_l||Y_l \to t$ a été appliquée en utilisant des compteurs et des

graphes comme décrit dans le **cas B1**. Pendant que l'automate remonte toutes ces prédictions à la racine, il doit vérifier que tous les sous termes u_i où les réécritures non locales ont eu lieu sont égaux à $\tilde{0}$ (ceci est bien sûr crucial pour que la règle $X \cdot Y \to t$ puisse être appliquée comme décrit dans la figure 5.5). Toutes les prédictions qui permettent de deviner que u' est successeur de u sont validées à la racine du contexte C comme décrit dans le **cas B1**. Quant à la prédiction "$-X$", elle est validée à la racine "$\cdot$" du terme, après avoir vérifié que $C_s[t]$ est annoté par un état de la forme (q, X) comme décrit dans la section 5.5.

De manière plus précise, pour annoter le terme $\cdot(C\left[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\right], C_s[t])$, l'automate décore les positions p de $C\left[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\right]$ qui sont différentes de m de la même manière que précédemment dans le **cas B1**. Quant à la position m, il la décore tout d'abord par le label "$-X$", et ensuite, par un graphe G_m et une valeur des compteurs v_m tels que (puisque pour chaque j, $1 \leq j \leq h_m$, $m = j_{k_{j,m}}$) :

- $v_m = \sum_{1 \leq j \leq h_m} \vec{k}_{l_{j,m}}$;
- $G_m = \{\bot \to Y^i_s \mid 1 \leq s \leq h_m\}$.

Comme précédemment, l'automate remonte ces prédictions des feuilles jusqu'à la racine en mémorisant à chaque fois l'union des graphes et la somme des compteurs. Par conséquent, la racine du contexte C est décorée par le label "$-X$", le graphe $G = \bigcup_{1 \leq p \leq n} G_p$ et la valeur du compteur $v = \sum_{1 \leq p \leq n} v_p$. Comme dans le cas précédent, si $G \in \mathcal{G}_c$ et $v = \vec{0}$, alors l'automate valide les prédictions mémorisées dans G et v, et continue à mémoriser la prédiction "$-X$" qu'il validera à la racine "$\cdot$" du terme.

Nous devons alors montrer que si $G \in \mathcal{G}_c$ et $v = \vec{0}$, alors les prédictions de l'automate sont correctes. Pour ce faire, nous définissons d'abord la notion de *seq-derived system* qui résume les conditions précédentes comme suit :

Définition 5.6.6 *Un* seq-derived system *est un sytème* $\left(X, (A_i, u_i, G_i, v_i, \mathcal{S}_i,)_{1 \leq i \leq n}\right)$ *tels que pour chaque* i, $1 \leq i \leq n$, A_i *est une variable de processus,* u_i *est un terme,* $G_i \in \mathcal{G}$, v_i *est une valuation de* $\vec{c}$, *et* $\mathcal{S}_i$ *est une séquence de règles de* $\mathcal{R}^{-1}$ *de la forme*

$$\left(Y^i_j || Z^i_j \to t^i_j \text{ (ou } Z^i_j || Y^i_j \to t^i_j \text{)}\right)_{1 \leq j \leq h_i}$$

tels que (nous posons $l_{j,i} = indice(Y^i_j)$*) :*

- *Si* $i \neq m$, *alors :*
  1. *Si* $h_i > 1$, *alors :*
     - $A_i = Z^i_1$,
     - $v_i = \sum_{1 \leq j \leq h_i - 1} \vec{k}_{l_{j,i}} + \vec{k}'_{l_{h_i,i}}$,
     - $G_i = \{Y^i_{h_i} \to Y^i_j \mid 1 \leq j \leq h_i - 1\}$,
     - $\forall j < h_i - 1$, $Z^i_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t^i_j)$,
     - $Y^i_{h_i} \in \zeta^*_{\mathcal{R}^{-1}}(t^i_{h_i - 1})$.
  2. *Si* $h_i = 1$, *alors :*
     - $A_i = Y^i_1$,
     - $v_i = \vec{k}'_{l_{h_i,i}}$,

- $G_i = \emptyset$.

– *Si* $i = m$, *alors :*
  - $A_m = Z_1^m$,
  - $v_m = \sum_{1 \leq j \leq h_m} \vec{k}_{l_{j,m}}$,
  - $G_m = \{\bot \rightarrow Y_j^i \mid 1 \leq j \leq h_m\}$,
  - $\forall j < h_m$, $Z_{j+1}^i \in \zeta^*_{\mathcal{R}^{-1}}(t_j^m)$,
  - $X \in \zeta^*_{\mathcal{R}^{-1}}(t_{h_m}^m)$.

Nous montrons que (voir le lemme 5.6.1) si $\big(X, (A_i, u_i, G_i, v_i, \mathcal{S}_i,)_{1 \leq i \leq n}\big)$ est un seq-derived system tel que $\bigcup_{1 \leq i \leq n} G_i \in \mathcal{G}_c$ et $\sum_{1 \leq i \leq n} v_i = \vec{0}$, alors

$$C\big[C'_1[\tilde{0}], \ldots, C'_m[X], \ldots, C'_n[\tilde{0}]\big] \in \wp^*_{\mathcal{R}^{-1}}\Big(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]\Big).$$

Lemme 5.6.1 *Soient* C *un paral-contexte, et* $C'_1, \ldots, C'_n$, n *null-contextes :*

– *Soit* $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$ *un derived system tel que* $\bigcup_{1 \leq i \leq n} G_i \in \mathcal{G}_c$ *et* $\sum_{1 \leq i \leq n} v_i = \vec{0}$, *alors*

$$u' = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big] \in \wp^*_{\mathcal{R}^{-1}}\Big(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]\Big).$$

– *Soit* $\big(X, (A_i, u_i, G_i, v_i, \mathcal{S}_i,)_{1 \leq i \leq n}\big)$ *un seq-derived system tel que* $\bigcup_{1 \leq i \leq n} G_i \in \mathcal{G}_c$ *et* $\sum_{1 \leq i \leq n} v_i = \vec{0}$, *alors*

$$C\big[C'_1[\tilde{0}], \ldots, C'_m[X], \ldots, C'_n[\tilde{0}]\big] \in \wp^*_{\mathcal{R}^{-1}}\Big(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]\Big).$$

Preuve : Nous avons tout d'abord besoin de quelques définitions préliminaires. Nous notons par $\{\!\{a_1, \ldots, a_n\}\!\}$ le multiensemble contenant $a_1, \ldots, a_n$. Soit $\mathcal{M}$ un multiensemble, nous notons par $set(\mathcal{M})$ l'ensemble des éléments dans $\mathcal{M}$. Par abus de notation, nous notons par $\bigcup$ l'union des multiensembles. Soit $\mathcal{G}'$ l'ensemble des multiensembles dont les éléments sont dans $Var \cup \{\bot\} \times Var$. $\mathcal{G}'$ peut être vu comme l'ensemble des multi-graphes dont les arêtes sont dans $Var \cup \{\bot\} \times Var$, et qui peuvent avoir plusieurs arêtes entre deux sommets donnés. Soit G un multi-graphe de $\mathcal{G}'$. Nous définissons l'ensemble $Noeuds(G)$ des sommets de G intervenant dans au moins une arête comme précédemment. Soit $\mathcal{G}'_c$ l'ensemble des multi-graphes G de $\mathcal{G}'$ tels qu'il existe un chemin menant de $\bot$ à chaque nœud de $Noeuds(G)$.

Nous avons alors de manière directe :

Proposition 5.6.2 *Soient* $G \in \mathcal{G}$ *et* $G' \in \mathcal{G}'$ *tels que* $G = set(G')$. *Alors* $G \in \mathcal{G}_c$ *ssi* $G' \in \mathcal{G}'_c$.

Dans ce qui suit, nous montrons le premier point du lemme 5.6.1, le deuxième étant très similaire. Soient C un paral-contexte, et $C'_1, \ldots, C'_n$ n null-contextes. Et soit $(A_i, u_i, \widetilde{G}_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$ un derived system tel que $\bigcup_{1 \leq i \leq n} \widetilde{G}_i \in \mathcal{G}_c$ et $\sum_{1 \leq i \leq n} v_i = \vec{0}$. Soient $(G_i)_{1 \leq i \leq n}$ des multi-graphes de $\mathcal{G}'$ définis comme suit :

– si $u_i \neq \tilde{0}$, alors $G_i = \{\!\{\bot \rightarrow Y_s^i \mid 1 \leq s \leq h_i\}\!\}$,
– si $u_i = \tilde{0}$, alors :
  1. si $h_i > 1$, $G_i = \{\!\{Y_{h_i}^i \rightarrow Y_s^i \mid 1 \leq s \leq h_i - 1\}\!\}$,
  2. si $h_i = 1$, $G_i = \emptyset$.

Il est clair d'après cette définition que pour tout i, $1 \leq i \leq n$, $\widetilde{G}_i = set(G_i)$, et $\bigcup_{1\leq i\leq n} \widetilde{G}_i = set\big(\bigcup_{1\leq i\leq n} G_i\big)$. Par conséquent, il s'en suit par la proposition 5.6.2 que pour montrer le lemme, il suffit de montrer que si $\bigcup_{1\leq i\leq n} G_i \in \mathcal{G}'_c$ et $\sum_{1\leq i\leq n} v_i = \vec{0}$, alors $u' = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big] \in \wp^*_{\mathcal{R}^{-1}}\Big(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]\Big)$.

Soit N le nombre de termes u_i qui sont égaux à $\tilde{0}$. Nous procédons par induction sur N :

• $N = 1$. Alors nécessairement, n est égal à deux. En effet, $N = 1$ implique qu'il existe exactement un vecteur de la forme $\vec{k}'_{\ldots}$ dans la somme qui calcule v, et puisque $v = \vec{0}$, il existe aussi exactement un vecteur de la forme $\vec{k}_{\ldots}$ dans la somme qui calcule v. Donc, n est soit égal à un ou deux. Supposons que $n = 1$, alors nécessairemant $u_1 = \tilde{0}$, $h_1 = 2$, et $G = G_1 = \{\!\{Y_1^1 \rightarrow Y_1^1\}\!\}$. Ceci est dû au fait que $v = v_1 = \vec{0}$, et donc forcément $\vec{0} = v_1 = \vec{k}_{l_{1,1}} + \vec{k}'_{l_{2,1}}$, ce qui implique que $l_{1,1} = l_{2,1}$, et donc que $Y_1^1 = Y_2^1$. Ceci contredit le fait que $G \in \mathcal{G}'_c$. Intuitivement, il n'est pas possible d'avoir $n = 1$ puisque pour appliquer $\wp_{\mathcal{R}^{-1}}$, nous avons besoin de deux feuilles qui interagissent ensemble. Par conséquent, $n = 2$. Soit alors $u = C\big[C'_1[u_1], C'_2[u_2]\big]$ tel que un des u_i est égal à $\tilde{0}$. Soit par exemple $u_1 = \tilde{0}$ (l'autre cas est symétrique). Alors :
– $h_1 = 1$, $G_1 = \emptyset$, $A_1 = Y_1^1$, et $v_1 = \vec{k}'_{l_{1,1}}$ (puisqu'il y a exactement un vecteur de la forme $\vec{k}_{\ldots}$ dans la somme qui calcule v),
– Puisque $v = v_1 + v_2 = \vec{0}$, il s'en suit que $v_2 = \vec{k}_{l_{1,1}}$, ce qui veut dire que $h_2 = 1$ et $l_{1,1} = l_{1,2}$, puisque v_2 est par définition égal à $\vec{k}_{l_{1,2}}$. Donc, nous avons que $Y_1^2 = Y_1^1$, $G_2 = \{\!\{\bot \rightarrow Y_1^2\}\!\}$, $A_2 = Z_1^2$, et $u_2 \in \zeta^*_{\mathcal{R}^{-1}}(t_1^2)$ (nous avons dans $\mathcal{R}^{-1}$ la règle $\mathcal{R}' : Y_1^2||Z_1^2 \rightarrow t_1^2$ ou $\mathcal{R}' : Z_1^2||Y_1^2 \rightarrow t_1^2$).

En appliquant la règle $\mathcal{R}'$ à $C\big[C'_1[A_1], C'_2[A_2]\big]$ qui est ègal à $C\big[C'_1[Y_1^2], C'_2[Z_1^2]\big]$, nous obtenons $C\big[C'_1[\tilde{0}], C'_2[t_1^2]\big]$ (puisque C est un paral-contexte et C'_1, C'_2 sont des null-contextes). Puisque $u_2 \in \zeta^*_{\mathcal{R}^{-1}}(t_1^2)$, nous obtenons que

$$u = C\big[C'_1[\tilde{0}], C'_2[u_2]\big] \in \wp^*_{\mathcal{R}^{-1}}(C\big[C'_1[A_1], C'_2[A_2]\big]).$$

Donc, la propriété est vraie pour $N = 1$.

• $N > 1$. Puisque $G \in \mathcal{G}'_c$, il existe nécessairement un indice r, $1 \leq r \leq n$ tel que $G_r = \{\!\{\bot \rightarrow Y_g^r \mid 1 \leq g \leq h_r\}\!\}$ (r est un nœud tel que u_r est différent de $\tilde{0}$, et tel que la dernière prédiction de l'automate à cette position concerne la variable $Y_{h_r}^r$). Comme $v_r = \sum_{1\leq j\leq h_r} \vec{k}_{l_{j,r}}$, et $\sum_p v_p = \vec{0}$, il existe au moins un indice s tel que $v_s = \sum_{1\leq j\leq h_s} \vec{k}_{l_{j,s}} + \vec{k}'_{l_{h_s,s}}$ (et donc $u_s = \tilde{0}$) et $\vec{k}'_{l_{h_s,s}} = \vec{k}'_{l_{h_r,r}}$; c-à-d. $l_{h_s,s} = l_{h_r,r}$. Soient alors $s_1, \ldots, s_k$ des indices tels que $\forall j, 1 \leq j \leq k$, $u_{s_j} = \tilde{0}$, $Y_{h_r}^r = Y_{h_{s_j}}^{s_j}$ (c-à-d., $l_{h_r,r} = l_{h_{s_j},s_j}$),

et $G_{s_j} = \{\!\{Y^r_{h_r} \to Y^{s_j}_g \mid 1 \leq g < h_{s_j}\}\!\}$ (ce qui veut dire qu'à chaque position s_j, il y avait un $Y^r_{h_r}$ qui s'est réécrit en $\tilde{0}$). Il y a deux cas selon que k est égal à un ou non (c-à-d., s'il y a seulement un seul nœud où $Y^r_{h_r}$ s'est directement transformé en $\tilde{0}$) :

– $k = 1$. Alors, intuitivement, il y a une règle $Y^r_{h_r}||Z^r_{h_r} \to t^r_{h_r}$ (ou $Z^r_{h_r}||Y^r_{h_r} \to t^r_{h_r}$) qui s'est appliquée aux positions r et $s = s_1$ pour transformer $Y^r_{h_r}$ en $\tilde{0}$ à la position s, et $Z^r_{h_r}$ en $t^r_{h_r}$ à la position r. Supposons que $r < s$ (l'autre cas est symétrique). Soit

$$u' = C\big[C'_1[u_1], \ldots, C'_r[Z^r_{h_r}], \ldots, C'_s[Y^r_{h_r}] \ldots, C'_n[u_n]\big]$$

Alors, en appliquant la règle $Y^r_{h_r}||Z^r_{h_r} \to t^r_{h_r}$ (ou $Z^r_{h_r}||Y^r_{h_r} \to t^r_{h_r}$), nous obtenons

$$C\big[C'_1[u_1], \ldots, C'_r[t^r_{h_r}], \ldots, C'_s[\tilde{0}] \ldots, C'_n[u_n]\big].$$

Puisque $u_r \in \zeta^*_{\mathcal{R}^{-1}}(t^r_{h_r})$, il s'en suit que $u \in \wp^*_{\mathcal{R}^{-1}}(u')$. Et donc, u' peut être réécrit sous la forme :

$$u' = C\big[C'_1[u'_1], \ldots, C'_n[u'_n]\big]$$

tels que :

– si $j \notin \{r, s\}, u'_j = u_j$,

– $u'_r = Z^r_{h_r}$,$u'_s = Y^r_{h_r}$

Posons pour $j \notin \{r, s\}$, $v'_j = v_j$ et $G'_j = G_j$. Soient $G'_r = \{\!\{\bot \to Y^r_g \mid 1 \leq g \leq h_r - 1\}\!\}$, $h'_r = h_r - 1$, $v'_r = \sum_{1 \leq j \leq h'_r} \vec{k}_{l_{j,r}}$. De même, soit $G'_s = \{\!\{\bot \to Y^s_g \mid 1 \leq g \leq h_s - 1\}\!\}$, $h'_s = h_s - 1$, $v'_s = \sum_{1 \leq j \leq h'_s} \vec{k}_{l_{j,s}}$. Il est facile de voir que :

– $\sum_{1 \leq i \leq n} v'_i = \vec{0}$, puisque $\sum_{1 \leq i \leq n} v_i = \vec{0}$, $v'_i = v_i$ si $i \notin \{r, s\}$, $v_r = v'_r + \vec{k}_{l_{h_r,r}}$, $v_s = v'_s + \vec{k}'_{l_{h_s,s}}$, et $l_{h_s,s} = l_{h_r,r}$, ce qui veut dire que $\vec{k}_{l_{h_r,r}} + \vec{k}'_{l_{h_s,s}} = \vec{0}$.

– $G' = \bigcup_{1 \leq i \leq n} G'_i$ est dans $\mathcal{G}'_c$. En effet, soit V un sommet de G', nous allons montrer qu'il existe dans G' un chemin menant de $\bot$ à V. V est aussi un sommet de G, puisque G' contient les mêmes (hyper)-arêtes que G, à part l'arête $\bot \to Y^r_{h_r}$ qui a été enlevée de G_r pour donner G'_r et les arêtes $Y^r_{h_r} \to Y^s_g$, $1 \leq g \leq h_s$, qui ont été enlevées de G_s, et ont été remplacées par les arêtes $\bot \to Y^s_g$, $1 \leq g \leq h_s - 1$, pour donner G'_s. Soit ρ un chemin de G menant de $\bot$ à V (un tel chemin existe puisque $G \in \mathcal{G}'_c$). Il y a deux cas :
  – ρ ne passe pas par $Y^r_{h_r}$. Il est alors clair que ρ est aussi un chemin de G'.
  – ρ est de la forme $\bot \to Y^r_{h_r} \rightsquigarrow V$ (où $\rightsquigarrow V$ représente un chemin menant à V). Alors, puisque $k = 1$, il s'en suit que les seules arêtes sortant de $Y^r_{h_r}$ sont les arêtes de G_s, c-à-d., ρ est nécessairement de la forme $\bot \to Y^r_{h_r} \to Y^s_g \rightsquigarrow V$, $1 \leq g \leq h_s - 1$. Comparé à G, dans G' nous avons enlevé les arêtes $\bot \to Y^r_{h_r}$, mais nous avons rajouté les arêtes $\bot \to Y^s_g$ pour tout g, $1 \leq g \leq h_s - 1$. Donc $\bot \to Y^s_g \rightsquigarrow V$ est un chemin de G' qui relie $\bot$ à V.

Par conséquent, $G' \in \mathcal{G}'_c$. Soient les séquences de règles $(\mathcal{S}'_i)_{1 \leq i \leq n}$ telles que

– si $i \notin \{r, s\}$, $\mathcal{S}'_i = \mathcal{S}_i$,

– $\mathcal{S}'_r = \Big(Y^r_j||Z^r_j \to t^r_j \text{ (ou } Z^r_j||Y^r_j \to t^r_j \text{)}\Big)_{1 \leq j \leq h'_r}$, et

– $\mathcal{S}'_s = \Big(Y^s_j || Z^s_j \to t^s_j \text{ (ou } Z^s_j || Y^s_j \to t^s_j \text{)} \Big)_{1 \le j \le h'_s}$.

Alors, le système $(A_i, u'_i, G'_i, v'_i, \mathcal{S}'_i)_{1 \le i \le n}$ est un derived system tel que $\bigcup\limits_{1 \le i \le n} G'_i \in \mathcal{G}'_c$ et $\sum\limits_{1 \le i \le n} v'_i = \vec{0}$. En plus, le nombre de u'_i égaux à $\tilde{0}$ est $N - 1$. Nous déduisons alors par induction que $u' \in \wp^*_{\mathcal{R}^{-1}}(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]$. Et puisque $u \in \wp^*_{\mathcal{R}^{-1}}(u')$, nous obtenons que $u \in \wp^*_{\mathcal{R}^{-1}}(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]$. Donc, la propriété est vraie dans ce cas.

– $k > 1$. Pour j, $1 \le j \le k$, nous avons $v_{s_j} = \sum\limits_{1 \le j \le h_{s_j} - 1} \vec{k}_{l_{j,s_j}} + \vec{k}'_{l_{h_r,r}}$, puisque $l_{h_r,r} = l_{h_{s_j},s_j}$. Comme $v = \sum\limits_{1 \le i \le n} v_i$, et $k \ge 2$, $v = \vec{k}'_{l_{h_r,r}} + \vec{k}'_{l_{h_r,r}} + \cdots$. Comme $v = \vec{0}$, et que $v_r = \vec{k}_{l_{h_r,r}} + \cdots$, il y a deux cas :

– soit $v_r = \vec{k}_{l_{h_r,r}} + \vec{k}_{l_{h_r,r}} + \cdots$, et l'hyper-graphe G_r contient au moins deux arêtes $\bot \to Y^r_{h_r}$;
– soit il existe un indice $g \ne r$ tel que $v_g = \vec{k}_{l_{h_r,r}} + \cdots$ et l'hyper-graphe G_g contient une arête $Y^g_{h_g} \to Y^r_{h_r}$ ou $\bot \to Y^r_{h_r}$.

Dans ces deux cas, l'hyper-graphe G contient, en plus de l'arête $\bot \to Y^r_{h_r}$ une autre arête de la forme $V_g \to Y^r_{h_r}$, où V_g peut être égale à $\bot$. Considérons un tel sommet V_g. Il y a deux cas :

(A) $V_g Y^r_{h_r}$, ou il existe un chemin $\bot \rightsquigarrow V_g$ dans G qui ne passe pas par $Y^r_{h_r}$. Dans ce cas, soit s un indice quelconque de $\{s_1, \ldots, s_k\}$.
(B) Tous les chemins menant de $\bot$ à V_g sont de la forme $\bot \to Y^r_{h_r} \rightsquigarrow V_g$. Il existe alors j, $1 \le j \le k$, un indice g' tels que $\bot \to Y^r_{h_r} \to Y^{s_j}_{g'} \rightsquigarrow V_g$. Soit s un tel indice s_j.

Nous définissons G'_i, h'_i, v'_i, et $\mathcal{S}'_i$ comme précédemment dans le cas $k = 1$. Alors, $\sum\limits_{1 \le i \le n} v'_i = \vec{0}$, et $G' = \bigcup\limits_{1 \le i \le n} G'_i \in \mathcal{G}'_c$. En effet, soit V un sommet de G'. Alors V est aussi un sommet de G, puisque G' contient les mêmes (hyper)-arêtes que G, à part l'arête $\bot \to Y^r_{h_r}$ qui a été enlevée de G_r pour donner G'_r et les arêtes $Y^r_{h_r} \to Y^s_{h_j}$, $1 \le j \le h_s$, qui ont été enlevées de G_s, et ont été remplacées par les arêtes $\bot \to Y^s_{h_j}$, pour j, $1 \le j \le h_s - 1$, pour donner G'_s. Soit ρ un chemin de G menant de $\bot$ à V (un tel chemin existe puisque $G \in \mathcal{G}'_c$). Il y a deux cas :

– ρ ne passe pas par $Y^r_{h_r}$. Il est alors clair que ρ est aussi un chemin de G'.
– ρ est de la forme $\bot \to Y^r_{h_r} \rightsquigarrow V$ (rappelons que $\rightsquigarrow V$ représente un chemin quelconque qui mène à V). Alors :

  1. S'il existe un $s_i \ne s$ tel que ρ est de la forme $\bot \to Y^r_{h_r} \to Y^{s_i}_m \rightsquigarrow V$,
     – Dans le premier cas (A), il existe un chemin $\bot \rightsquigarrow V_g$ tel que G' contient un chemin de la forme $\bot \rightsquigarrow V_g \to Y^r_{h_r} \to Y^{s_i}_m \rightsquigarrow V$.
     – Dans le second cas (B), il existe un chemin $Y^s_{g'} \rightsquigarrow V_g$ tel que G' contient un chemin de la forme $\bot \to Y^s_{g'} \rightsquigarrow V_g \to Y^r_{h_r} \to Y^{s_i}_m \rightsquigarrow V$.
  2. Si ρ est de la forme $\bot \to Y^r_{h_r} \to Y^s_m \rightsquigarrow V$, alors $\bot \to Y^s_m \rightsquigarrow V$ est un chemin de G'.

Alors, le système $(A_i, u'_i, G'_i, v'_i, \mathcal{S}'_i)_{1\leq i\leq n}$ est un derived system tel que $\bigcup_{1\leq i\leq n} G'_i \in \mathcal{G}'_c$ et $\sum_{1\leq i\leq n} v'_i = \vec{0}$. En plus, le nombre de u'_i égaux à $\tilde{0}$ est $N-1$. Nous déduisons alors par induction que $u' \in \wp^*_{\mathcal{R}^{-1}}(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]$. Et puisque $u \in \wp^*_{\mathcal{R}^{-1}}(u')$, nous obtenons que $u \in \wp^*_{\mathcal{R}^{-1}}(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]$. Donc, la propriété est vraie.

□

5.6.4 Construction

Nous donnons maintenant la construction de l'automate. Soit $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini qui reconnaît le langage régulier $\mathcal{L}$, et soit $\mathcal{A}^*_{\zeta_{\mathcal{R}^{-1}}} = (\tilde{Q}, \Sigma, \tilde{F}, \tilde{\delta})$ l'automate d'arbres défini dans le théorème 5.5.1 qui reconnaît $\zeta^*_{\mathcal{R}^{-1}}(\mathcal{L})$. Nous construisons un automate à compteurs $\mathcal{A}^*_{\wp_{\mathcal{R}^{-1}}} = (\hat{Q}, \Sigma, \hat{F}, \vec{c}, \hat{\delta})$ qui se comporte comme décrit dans la section précédente. Pour réaliser tous les types de prédictions décrits plus haut, $\hat{Q}$ contient, en plus des états de $\tilde{Q}$, des états de la forme $(q, -X)$, où q est un état de Q_{null}, pour mémoriser les prédictions "$-X$", et des quadruplets de la forme (q, a, G, w) où :

- q représente un état de $Q_{null} \cup Q_{null} \times Var$,
- $a \in Var \cup \{\tilde{0}\} \cup \{\bot\}$ permet de mémoriser, pour chaque position $p \in \{1, \ldots, n\}$ si ou $u_p = \tilde{0}$ (auquel cas G_p contient des arêtes de la forme $Y^p_{h_p} \to Y^p_j$, et $a = Y^p_{h_p}$ permet de mémoriser $Y^p_{h_p}$ pour construire le graphe G_p), ou non (auquel cas G_p contient des arêtes de la forme $\bot \to Y^p_j$ et $a = \bot$). Soit $C'\big[C'_{p_1}[u_{p_1}], \ldots, C'_{p_k}[u_{p_k}]\big]$ un sous terme de u'. A la racine de ce terme la composante "a" est soit égale à $\bot$ s'il existe au moins un indice i, $1 \leq i \leq k$ tel que $u_{p_i} \neq \tilde{0}$, soit égale à $\tilde{0}$ si pour tout i, $1 \leq i \leq k$ $u_{p_i} = \tilde{0}$. Nous avons besoin de cette information pour vérifier, pendant l'annotation de $\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t]\Big)$ que tous les sous termes u_i sont égaux à $\tilde{0}$.
- G est un graphe appartenant à l'ensemble $\mathcal{G} = 2^{Var \cup \{\bot\} \times Var}$. $Var \cup \{\bot\} \times Var$.
- $w \in \{\emptyset\} \cup \{-X \mid X \in Var\}$ permet de mémoriser les prédictions "$-X$". Nous avons "$\emptyset$" dans cette composante de l'état si aucune prédiction de cette forme n'a été faite.

Ces états permettent de stocker ce que l'automate a deviné pour pouvoir les valider plus tard. Pour valider les prédictions relatives aux règles de la forme $X||Y \to t$, l'automate passe d'un état de la forme $(q, a, G, \emptyset)$ à q^T, et d'un état de la forme $(q, a, G, -Z)$ à $(q, -Z)$.

Maintenant, nous sommes prêts pour donner la construction de $\mathcal{A}^*_{\wp_{\mathcal{R}^{-1}}}$: $\mathcal{A}^*_{\wp_{\mathcal{R}^{-1}}} = (\hat{Q}, \Sigma, \hat{F}, \vec{c}, \hat{\delta})$ tels que :

- $\hat{Q} = \tilde{Q} \cup \{(p, a, G, w) \mid p \in Q_{null} \cup Q_{null} \times Var, a \in Var \cup \{\tilde{0}\} \cup \{\bot\}, G \in \mathcal{G}, w \in \{\emptyset\} \cup \{-X \mid X \in Var\}\} \cup \{(p, -X) \mid p \in Q_{null} \cup Q_{null} \times Var\}$,
- $\hat{F} = \tilde{F}$,
- $\hat{\delta}$ est le plus petit ensemble de règles qui contient $\tilde{\delta}$ et qui satisfait :

(γ_1) Si $X_i||X_j \to t$ ou $X_j||X_i \to t$ est une règle de $\mathcal{R}^{-1}$, alors :

(a) Si $X_i \xrightarrow{*}_{\tilde{\delta}} q^T$, alors :

i. $\tilde{0} \overset{vrai/\lambda'_i}{\to} (q, X_i, \emptyset, \emptyset) \in \hat{\delta}$,

ii. $(q_t, a, G, w) \overset{vrai/\lambda_j}{\to} (q, a, G \cup \{a \to X_j\}, w) \in \hat{\delta}$, $a \in \{\bot\} \cup Var$,

iii. $q_t^T \overset{vrai/\lambda_j}{\to} (q, \bot, \{\bot \to X_j\}, \emptyset) \in \hat{\delta}$,

iv. $(q_t, -X) \overset{vrai/\lambda_j}{\to} \big(q, \bot, \{\bot \to X_j\}, -X\big) \in \hat{\delta}$,

(b) Si $X_i \xrightarrow{*}_{\tilde{\delta}} (q, W)$, alors :

i. $\tilde{0} \overset{vrai/\lambda'_i}{\to} ((q, W), X_i, \emptyset, \emptyset) \in \hat{\delta}$,

ii. $(q_t, a, G, w) \overset{vrai/\lambda_j}{\to} \big((q, W), a, G \cup \{a \to X_j\}, w\big) \in \hat{\delta}$, $a \in \{\bot\} \cup Var$,

iii. $q_t^T \overset{vrai/\lambda_j}{\to} \big((q, W), \bot, \{\bot \to X_j\}, \emptyset\big) \in \hat{\delta}$,

iv. $(q_t, -X) \overset{vrai/\lambda_j}{\to} \big((q, W), \bot, \{\bot \to X_j\}, -X\big) \in \hat{\delta}$,

(γ_2) Si $X \cdot Y \to t$ est une règle de $\mathcal{R}^{-1}$, alors :

(a) Si $X \xrightarrow{*}_{\tilde{\delta}} q^T$, alors $\tilde{0} \to (q, -X) \in \hat{\delta}$,

(b) Si $X \xrightarrow{*}_{\tilde{\delta}} (q, W)$, alors $\tilde{0} \to \big((q, W), -X\big) \in \hat{\delta}$,

(γ_3) Si $p_1 \xrightarrow{*}_{\tilde{\delta}} p_2$, alors $(p'_1, a, G, w) \to (p'_2, a, G, w) \in \hat{\delta}$ et $(p'_1, -X) \to (p'_2, -X) \in \hat{\delta}$, où $p'_i = q_i$ si $p_i = q_i^T$, et $p'_i = (q_i, W_i)$ si $p_i = (q_i, W_i)$,

(γ_4) $(q, a, G, \emptyset) \overset{\vec{c}=\vec{0}/\lambda}{\to} q^T \in \hat{\delta}$, et $(q, a, G, -X) \overset{\vec{c}=\vec{0}/\lambda}{\to} (q, -X) \in \hat{\delta}$, où λ est l'affectation nulle, $a \in \{\bot, \tilde{0}\}$ et $G \in \mathcal{G}_c$,

(γ_5) Si $||(q_1, q_2) \to q \in \delta_{null}$, alors :

(a) $||\big((q_1, a_1, G_1, \emptyset), q_2^T\big) \to (q, \bot, G_1, \emptyset) \in \hat{\delta}$,

(b) $||\big(q_1^T, (q_2, a_2, G_2, \emptyset)\big) \to (q, \bot, G_2, \emptyset) \in \hat{\delta}$,

(c) $||\big((q_1, a_1, G_1, \emptyset), (q_2, a_2, G_2, \emptyset)\big) \to (q, \bot, G_1 \cup G_2, \emptyset) \in \hat{\delta}$,

(d) $||\big((q_1, a_1, G_1, \emptyset), (q_2, a_2, G_2, \emptyset)\big) \to (q, \tilde{0}, G_1 \cup G_2, \emptyset) \in \hat{\delta}$, si $a_1 \neq \bot$ et $a_2 \neq \bot$,

(e) $||\big((q_1, \bot, G_1, -X), (q_2, a_2, G_2, \emptyset)\big) \to (q, \bot, G_1 \cup G_2, -X) \in \hat{\delta}$, si $a_2 \in \{\tilde{0}\} \cup Var$,

(f) $||\big((q_1, a_1, G_1, \emptyset), (q_2, \bot, G_2, -X)\big) \to (q, \bot, G_1 \cup G_2, -X) \in \hat{\delta}$, si $a_1 \in \{\tilde{0}\} \cup Var$,

(γ_6) Si $\cdot(q_1, q_2) \to q \in \delta_{null}$, alors :

(a) $\cdot\Big((q_1, -X), \big((q_2, X), a, G, w\big)\Big) \to (q, a, G, w) \in \hat{\delta}$, $a \in Var \cup \{\bot\}$, tel que si $w = -Z$, alors $a = \bot$,

(b) $\cdot\big((q_1, -X), (q_2, X)\big) \to q^T \in \hat{\delta}$,

(c) $\cdot\Big((q_1,-X),\big((q_2,X),-W\big)\Big) \to (q,-W) \in \hat{\delta}$.

Notons que $\mathcal{A}^*_{\wp_{\mathcal{R}^{-1}}}$ est un 0-CTA puisque les seules règles contraintes sont les règles γ_4 qui testent si tous les compteurs sont nuls. Nous avons alors le résultat suivant :

Théorème 5.6.4 *Soit $\mathcal{R}$ un PAD, $\mathcal{L}$ un ensemble régulier de termes, et $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini qui reconnaît $\mathcal{L}$. Alors, $\wp^*_{\mathcal{R}^{-1}}(\mathcal{L})$ est reconnu par le 0-CTA $\mathcal{A}^*_{\wp_{\mathcal{R}^{-1}}}$ ci-dessus.*

Avant de prouver ce théorème, nous expliquons informellement les règles ci-dessus.

5.6.5 Explication des règles

Pour expliquer les règles, nous reprenons les cas précédents **B1** et **B2** :

5.6.5.1 Cas B1

Rappelons que dans ce cas nous avons la situation suivante :
- Si $u_p \neq \tilde{0}$, nous avons :
  - $A_p = Z^p_1$,
  - $\forall 1 \leq j < h_p$, $Z^p_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t^p_j)$,
  - $u_p \in \zeta^*_{\mathcal{R}^{-1}}(t^p_{h_p})$.
  - De plus, l'automate doit deviner cette valeur des compteurs : $v_p = \sum_{1 \leq j \leq h_p} \vec{k}_{l_{j,p}}$, et le graphe suivant : $G_p = \{\perp \to Y^p_j \mid 1 \leq j \leq h_p\}$.
- Si $u_p = \tilde{0}$, alors :
  1. Si $h_p = 1$, alors :
     - $A_p = Y^p_1$.
     - L'automate doit deviner cette valeur des compteurs : $v_p = \vec{k}'_{l_{h_p,p}}$, et le graphe $G_p = \emptyset$.
  2. Si $h_p > 1$, alors :
     - $A_p = Z^p_1$,
     - $\forall 1 \leq j < h_p - 1$, $Z^p_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t^p_j)$,
     - $Y^p_{h_p} \in \zeta^*_{\mathcal{R}^{-1}}(t^p_{h_p-1})$.
     - De plus, l'automate doit deviner cette valeur des compteurs :

$$v_p = \sum_{1 \leq j \leq h_p - 1} \vec{k}_{l_{j,p}} + \vec{k}'_{l_{h_p,p}}$$

     ainsi que le graphe $G_p = \{Y^p_{h_p} \to Y^p_j \mid 1 \leq j \leq h_p - 1\}$.

Pour ce faire, l'automate utilise les règles ci-dessus comme suit :
- Si $u_p \neq \tilde{0}$, l'automate annote ce terme u_p par l'état

$$\big((q_{A_p}, \perp, \{\perp \to Y^p_j \mid 1 \leq j \leq h_p\}, \emptyset), \sum_{1 \leq j \leq h_p} \vec{k}_{l_{j,p}}\big).$$

Pour ce faire, il procède comme suit : Comme $u_p \in \zeta^*_{\mathcal{R}^{-1}}(t^p_{h_p})$, l'automate utilise les règles de $\tilde{\delta}$ pour annoter u_p par $q^T_{t^p_{h_p}}$ ($t^p_{h_p} \in Sub_r(\mathcal{R}^{-1})$). Ensuite, comme $Z^p_{h_p} \xrightarrow{*}_{\tilde{\delta}} q^T_{Z^p_{h_p}}$, il utilise les règles $\gamma_1$aiii pour passer à l'état

$$\big((q_{Z^p_{h_p}}, \bot, \{\bot \to Y^p_{h_p}\}, \emptyset), \vec{k}_{l_{h_p,p}}\big).$$

Comme $Z^p_{h_p} \in \zeta^*_{\mathcal{R}^{-1}}(t^p_{h_{p-1}})$, nous avons dans $\tilde{\delta}$ la règle $q^T_{Z^p_{h_p}} \to_{\tilde{\delta}} q^T_{t^p_{h_{p-1}}}$. Ceci est dû aux règles ($\alpha_3 a$). En effet, $t^p_{h_{p-1}} \in Sub_r(\mathcal{R}^{-1})$ et puisque $Z^p_{h_p} \in \zeta^*_{\mathcal{R}^{-1}}(t^p_{h_{p-1}})$, $Z^p_{h_p}$ est aussi dans $Sub_r(\mathcal{R}^{-1})$. Donc en appliquant la règle γ_3, l'automate passe à l'état

$$\big((q_{t^p_{h_{p-1}}}, \bot, \{\bot \to Y^p_{h_p}\}, \emptyset), \vec{k}_{l_{h_p,p}}\big).$$

Ensuite, il utilise les règles $\gamma_1$aii pour passer à l'état

$$\big((q_{Z^p_{h_{p-1}}}, \bot, \{\bot \to Y^p_{h_p}, \bot \to Y^p_{h_{p-1}}\}, \emptyset), \vec{k}_{l_{h_p,p}} + \vec{k}_{l_{h_{p-1},p}}\big).$$

Et ainsi de suite, l'automate utilise les règles précédentes jusqu'à arriver à l'état

$$\big((q_{Z^p_1}, \bot, \{\bot \to Y^p_j \mid 1 \le j \le h_p\}, \emptyset), \sum_{1 \le j \le h_p} \vec{k}_{l_{j,p}}\big),$$

et comme $Z^p_1 = A_p$, nous avons ce que nous voulions.

– Si $u_p = \tilde{0}$

1. Si $h_p = 1$, l'automate applique $\gamma_1$ai pour annoter le $\tilde{0}$ à cette position par

$$\big((q_{Y^p_{h_p}}, Y^p_{h_p}, \emptyset, \emptyset), \vec{k}'_{l_{h_p,p}}\big).$$

2. Si $h_p > 1$, l'automate annote ce terme u_p par l'état

$$\big((q_{A_p}, Y^p_{h_p}, \{Y^p_{h_p} \to Y^p_j \mid 1 \le j \le h_p - 1\}, \emptyset), \sum_{1 \le j \le h_p - 1} \vec{k}_{l_{j,p}} + \vec{k}'_{l_{h_p,p}}\big).$$

Pour ce faire, il applique $\gamma_1$ai pour annoter le $\tilde{0}$ à cette position par

$$\big((q_{Y^p_{h_p}}, Y^p_{h_p}, \emptyset, \emptyset), \vec{k}'_{l_{h_p,p}}\big).$$

Comme $Y^p_{h_p} \in \zeta^*_{\mathcal{R}^{-1}}(t^p_{h_{p-1}})$, nous avons dans $\tilde{\delta}$ la règle $q^T_{Y^p_{h_p}} \to_{\tilde{\delta}} q^T_{t^p_{h_{p-1}}}$, et donc en appliquant la règle γ_3, l'automate passe à l'état

$$\big((q_{t^p_{h_{p-1}}}, Y^p_{h_p}, \emptyset, \emptyset), \vec{k}'_{l_{h_p,p}}\big).$$

Ensuite, il utilise les règles $\gamma_1$aii pour passer à l'état

$$\big((q_{Z^p_{h_{p-1}}}, Y^p_{h_p}, \{Y^p_{h_p} \to Y^p_{h_{p-1}}\}, \emptyset), \vec{k}_{l_{h_{p-1},p}} + \vec{k}'_{l_{h_p,p}}\big).$$

Et nous voyons ici que pour considérer l'arête $Y^p_{h_p} \to Y^p_{h_{p-1}}$ du graphe, l'automate a utilisé l'information stockée dans la deuxième composante exprimant qu'à cette position $Y^p_{h_p}$ s'est transformé en $\tilde{0}$. L'automate va maintenant utiliser les règles $\gamma_1$aii et γ_3 jusqu'à arriver à l'état

$$\big((q_{Z^p_1}, Y^p_{h_p}, \{Y^p_{h_p} \to Y^p_j \mid 1 \leq j \leq h_p - 1\}, \emptyset), \sum_{1 \leq j \leq h_p - 1} \vec{k}_{l_{j,p}} + \vec{k}'_{l_{h_p,p}}\big),$$

c-à-d. l'état voulu puisque $Z^p_1 = A_p$.

Après avoir deviné ceci à chaque position p, toutes ces prédictions sont remontées à la racine du paral-contexte C en mémorisant les sommes de tous les compteurs et l'union des graphes en utilisant les règles γ_5a, γ_5b, et γ_5c. La racine de u' sera alors annotée par un état de la forme

$$\big((q, \bot, \bigcup_{1 \leq p \leq n} G_p, \emptyset), \sum_{1 \leq p \leq n} v_p\big).$$

Comme montré précédemment, ces prédictions sont correctes ssi $\bigcup_{1 \leq p \leq n} G_p \in \mathcal{G}_c$ et $\sum_{1 \leq p \leq n} v_p = \vec{0}$. Donc, les règles γ_4 permettent de valider ce que l'automate a deviné, et d'annoter le nœud par l'état q^T exprimant que c'est un successeur de L_q par $\wp_{\mathcal{R}^{-1}}$.

5.6.5.2 Cas B2

Dans ce cas, pour toute position différente de m, l'automate se comporte comme décrit ci-dessus. Pour la position m, nous avons la situation suivante :

- $A_m = Z^m_1$,
- $\forall j < h_m$, $Z^i_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t^m_j)$,
- $X \in \zeta^*_{\mathcal{R}^{-1}}(t^m_{h_m})$.
- De plus, l'automate doit deviner la valeur du compteur $v_m = \sum_{1 \leq j \leq h_m} \vec{k}_{l_{j,m}}$ et le graphe $G_m = \{\bot \to Y^i_s \mid 1 \leq s \leq h_m\}$.

En fait, il doit annoter cette position par l'état

$$\big((q_{A_m}, \bot, \{\bot \to Y^m_j \mid 1 \leq j \leq h_m\}, -X), \sum_{1 \leq j \leq h_m} \vec{k}_{l_{j,m}}\big).$$

Pour ce faire, il commence par annoter le $\tilde{0}$ à cette position par $(q_X, -X)$ en utilisant les règles γ_2a. Ensuite, puisque $X \in \zeta^*_{\mathcal{R}^{-1}}(t^m_{h_m})$, nous avons dans $\tilde{\delta}$ la règle $q^T_X \to_{\tilde{\delta}} q^T_{t^m_{h_m}}$, et donc en appliquant les règles γ_3, l'automate passe à l'état $(q_{t^m_{h_m}}, -X)$. Ensuite, en appliquant les règles $\gamma_1$aiv, l'automate passe à l'état

$$\big((q_{Z^m_{h_m}}, \bot, \{\bot \to Y^m_{h_m}\}, -X), \vec{k}_{l_{h_m,m}}\big).$$

Ensuite, il procède comme décrit dans le **Cas B1** jusqu'à arriver à l'état

$$\big((q_{Z_1^m}, \bot, \{\bot \to Y_j^m \mid 1 \leq j \leq h_m\}, -X), \sum_{1 \leq j \leq h_m} \vec{k}_{l_{j,m}}\big),$$

où $A_m = Z_1^m$.

Comme précédemment, toutes ces prédictions sont remontées à la racine du paral-contexte C en mémorisant les sommes de tous les compteurs et l'union des graphes en utilisant les règles γ_5d, γ_5e, et γ_5f. Ces règles assurent que tous les u_i sont égaux à $\tilde{0}$ (la deuxième composante "a" est soit $\tilde{0}$ soit dans Var). La racine de $C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big]$ sera alors annotée par un état de la forme

$$\big((q, \bot, \bigcup_{1 \leq p \leq n} G_p, -X), \sum_{1 \leq p \leq n} v_p\big),$$

où $\bigcup_{1 \leq p \leq n} G_p \in \mathcal{G}_c$ et $\sum_{1 \leq p \leq n} v_p = \vec{0}$. Donc, les règles γ_4 permettent de valider tout ce que l'automate a deviné, et d'annoter le nœud par l'état $(q, -X)$.

Maintenant, il faut valider cette prédiction "$-X$". Soient alors des états q' et q'' tels que $C_s[Y]$ est annoté par q', et $\cdot(q, q') \to q''$ est une règle de δ_{null}. Il y a alors quatre cas différents en fonction du contexte dans lequel se trouve le terme $\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t])$ et des différentes réécritures possibles qu'il pourrait encore faire :

1. Si le terme $\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t])$ se transforme en $\cdot(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], U)$ tel que $U \in \zeta^*_{\mathcal{R}^{-1}}(C_s[t])$, alors comme vu dans la section 5.5, U est annoté par (q', X), et les règles γ_6b permettent de valider la prédiction "$-X$" et d'annoter la racine du terme par ${q''}^T$.
2. Si ce terme se trouve dans un contexte de la forme

$$\cdot\Big(\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t]\Big), C'_s[Z]\Big)$$

où $C'_s[x]$ est un autre seq-context, et Z est une variable tels qu'il existe une variable W et une règle dans $\mathcal{R}^{-1}$ de la forme $W \cdot Z \to t'$, et tels que $C_s[t]$ se transforme par les règles de $\zeta_{\mathcal{R}^{-1}}$ en la variable W, nous obtenons alors le terme

$$\cdot\Big(\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], W\Big), C'_s[Z]\Big)$$

auquel nous pouvons appliquer la règle $W \cdot Z \to t'$ et obtenir

$$\cdot\Big(\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], \tilde{0}\Big), C'_s[t']\Big)$$

Si nous appelons ce terme T, l'automate annote T comme suit : D'abord, comme décrit précédemment, nous avons

$$T \xrightarrow{*}_{\hat{\delta}} \cdot\Big(\cdot\big((q, -X), \tilde{0}\big), C'_s[t']\Big).$$

Comme $C_s[Y]$ est reconnu par q', $W \in \zeta^*_{\mathcal{R}^{-1}}(C_s[t])$, et $X \cdot Y \to t$ est une règle de $\mathcal{R}^{-1}$, nous avons à partir du lemme 5.5.1 que $W \xrightarrow{*}_{\tilde{\delta}} (q', X)$, et donc les règles γ_2b permettent d'annoter $\tilde{0}$ par $\big((q', X), -W\big)$. Nous aurons alors

$$T \xrightarrow{*}_{\hat{\delta}} \cdot\Big(\cdot \Big((q, -X), \big((q', X), -W\big)\Big), C'_s[t']\Big).$$

Les règles γ_6c permettent de valider la prédiction "$-X$" et de garder la prédiction "$-W$". Nous obtenons alors :

$$T \xrightarrow{*}_{\hat{\delta}} \cdot\Big(\cdot \Big((q, -X), \big((q', X), -W\big)\Big), C'_s[t']\Big) \to_{\hat{\delta}} \cdot\big((q'', -W), C'_s[t']\big).$$

Finalement, $C'_s[t']$ est annoté par un état de la forme (q''', W) (lemme 5.5.1), et la prédiction "$-W$" est validée par les règles γ_6b.

3. S'il existe un paral-contexte $D[x_1, \ldots, x_l]$, l null-contextes $D_1[x], \ldots, D_l[x]$, et l variables $B_1, \ldots, B_l$ telles que l'on ait le terme

$$D\Big[D_1[B_1], \ldots, D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], C_s[t]\Big)\Big], \ldots, D_l[B_l]\Big].$$

Supposons que le terme $C_s[t]$ se réécrive en B_h par $\zeta_{\mathcal{R}^{-1}}$, et ensuite que ce B_h interagit avec les autres B_i en appliquant des règles de la forme $X'||Y' \to t'$ de manière non locale comme décrit précédemment (puisque

$$D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], x\Big)\Big]$$

est un null-contexte). Nous obtenons alors un terme de la forme

$$D\Big[D_1[s_1], \ldots, D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], s_h\Big)\Big], \ldots, D_l[s_l]\Big].$$

Comme vu précédemment, à chaque position p, $1 \leq p \leq l$, l'automate doit deviner la bonne valeur des graphes et des compteurs v_p et G_p. Pour annoter la racine de D, l'automate se comporte comme précédemment (dans le **Cas B1**) pourvu que le terme $\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], s_h\Big)$ soit annoté par un état de la forme

$$\big((q'', a, G_h, \emptyset), v_h\big).$$

Pour ce faire, puisque $B_h \xrightarrow{*}_{\tilde{\delta}} (q', X)$ (par le lemme 5.5.1, puisque $C_s[Y]$ est reconnu par q', $B_h \in \zeta^*_{\mathcal{R}^{-1}}(C_s[t])$, et $X \cdot Y \to t$ est une règle de $\mathcal{R}^{-1}$), en appliquant les règles γ_1b (comme étaient appliquées les règles γ_1a précédemment), l'automate annote s_h par un état de la forme $\Big(\big((q', X), a, G_h, \emptyset\big), v_h\Big)$. La prédiction concernant X sera alors validée par les règles γ6a, puisque $C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big]$ est annoté par $(q, -X)$, et donc $\cdot\Big(C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big], s_h\Big)$ est annoté par un état de la forme

$$\big((q'', a, G_h, \emptyset), v_h\big).$$

4. Le dernier cas est s'il existe une règle $W \cdot Z \to t'$ et un terme de la forme

$$\cdot\Big(D\Big[D_1[B_1],\ldots,D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}],\ldots,C'_n[\tilde{0}]\big],C_s[t]\Big)\Big],\ldots,D_l[B_l]\Big],C'_s[Z]\Big)$$

pour un seq-contexte C'_s et une variable Z, où D et les D_i sont comme précédemment ; tels que comme précédemment, $C_s[t]$ se réécrit en B_h qui interagit avec les autres B_i pour donner

$$D\Big[D_1[s_1],\ldots,D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}],\ldots,C'_n[\tilde{0}]\big],s_h\Big)\Big],\ldots,D_l[s_l]\Big].$$

où $s_h = W$ et tous les autres s_i sont égaux à $\tilde{0}$ de telle manière à ce que le contexte

$$D\Big[D_1[\tilde{0}],\ldots,D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}],\ldots,C'_n[\tilde{0}]\big],x\Big)\Big],\ldots,D_l[\tilde{0}]\Big]$$

soit un null-contexte. La règle $W \cdot Z \to t'$ peut alors alors s'appliquer à

$$\cdot\Big(D\Big[D_1[\tilde{0}],\ldots,D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}],\ldots,C'_n[\tilde{0}]\big],W\Big)\Big],\ldots,D_l[\tilde{0}]\Big],C'_s[Z]\Big)$$

pour donner

$$\cdot\Big(D\Big[D_1[\tilde{0}],\ldots,D_h\Big[\cdot\Big(C\big[C'_1[\tilde{0}],\ldots,C'_n[\tilde{0}]\big],\underline{\tilde{0}}\Big)\Big],\ldots,D_l[\tilde{0}]\Big],C'_s[t']\Big).$$

Dans ce cas, le $\tilde{0}$ souligné est annoté par $(q_W, -W)$ grâce aux règles γ_2a. Ensuite, en appliquant les règles γ_1a comme précédemment dans le **Cas B1** (nous utilisons $\gamma_1$aiv à la place de $\gamma_1$aiii), et la règle $\gamma_1$bii à la fin (en utilisant le fait que $B_h \xrightarrow{*}_{\tilde{\delta}} (q', X)$, l'automate annote cette position par un état de la forme $\Big(\big((q', X), \bot, G_h, -W\big), v_h\Big)$. La prédiction "$-X$" est validée par les règles γ_6a puisque $C\big[C'_1[\tilde{0}],\ldots,C'_n[\tilde{0}]\big]$ est annoté par $(q, -X)$, et ensuite, les prédictions concernant les compteurs, les graphes et "$-W$" sont remontées à la racine de D en vérifiant que tous les s_i sont des $\tilde{0}$ (ceci se fait en utilisant les règles γ_5d, γ_5e, et γ_5f). La racine de D est alors annotée par un état de la forme $\Big(\big(p, \bot, \bigcup G_p, -W\big), \sum v_p\Big)$ tels que $\bigcup G_p \in \mathcal{G}_c$, et $\sum v_p = \vec{0}$. Les règles γ_4 permettent alors de valider ces prédictions et de passer à l'état $(p, -W)$. Comme $C'_s[t']$ est annoté par un état de la forme (p', W) (lemme 5.5.1), la prédiction "$-W$" est validée par les règles γ_6b.

5.6.5.3 Quelques remarques

Observons que le fait que les règles de $\mathcal{R}^{-1}$ ne contiennent pas de $||$ dans leur membre droit est crucial pour que la construction ci-dessus marche. En effet, si c'était

le cas, notre raisonnement précédent basé sur la notion de *derived system* n'est plus valable puisqu'au cours d'une dérivation, les paral-contextes auquels les règles de la forme $X||Y \to t$ sont appliquées croissent. En effet prenons l'exemple suivant : $\mathcal{R}^{-1}$ contient les règles $\mathcal{R}_1 : X||Z \to W$, $W \to A||B$, et $A||Y \to D$. Considérons le terme $t_1 = ||\big(||(X,Y),Z\big)$. En appliquant $\mathcal{R}_1$, nous obtenons $t_2 = ||\big(||(\tilde{0},Y),W\big)$. Ensuite, en appliquant $\mathcal{R}_2$, nous obtenons $t_3 = ||\big(||(\tilde{0},Y),||(A,B)\big)$, et enfin la règle $\mathcal{R}_3$ nous donne $t_4 = ||\big(||(\tilde{0},\tilde{0}),||(D,B)\big)$. Nous voyons alors que nous sommes partis d'un paral-contexte $C[x_1,x_2,x_3] = ||\big(||(x_1,x_2),x_3\big)$ ($t_1 = ||\big(||(X,Y),Z\big)$) et nous obtenons un terme $t_3 = ||\big(||(\tilde{0},Y),||(A,B)\big) = C'[Y,A,B]$, où C' est le paral-contexte $C'[x_1,x_2,x_3] = ||\big(||(\tilde{0},x_1),||(x_2,x_3)\big)$, auquel la règle $\mathcal{R}_3$ peut s'appliquer.

Voyons maintenant pourquoi est-ce que nous avons besoin de garder la structure de l'arbre (en termes de nœuds étiquetés par "||") en remplaçant X par $\tilde{0}$ et Y par t en appliquant une règle de la forme $X||Y \to t$ (ou $Y||X \to t$), et pourquoi nous ne procédons pas comme dans la section 5.5 en éliminant tout simplement le nœud correspondant à X (ou celui de Y). Ceci est dû au fait que dans ce cas, nous aurons besoin de règles du même genre que $\alpha_4 e$ pour valider et accumuler les prédictions : Nous aurons besoin d'une règle d'inférence de la forme : Si $||(q_1,q_2) \to q$ est une règle de l'automate initial, et si $X_i \xrightarrow{*}_{\hat{\delta}} \big((q_1,...),v\big)$, exprimant que X_i peut être obtenu à partir de L_{q_1} en faisant des prédictions correspondant aux valeurs v des compteurs, alors nous devons considérer les règles suivantes $(q_2,...) \xrightarrow{vrai/v+\vec{k}'_i}_{\hat{\delta}} (q,...)$, exprimant que si u est un terme reconnu par $\big((q_2,...),v'\big)$, et si X_i a été éliminé (si nous ne considérons pas $\tilde{0}$) en utilisant une règle de la forme $X_i||X_j \to t$, alors le terme u devrait être reconnu par $\big((q_2,...),v'+v+\vec{k}'_i\big)$: l'automate doit mémoriser que X_i était d'abord obtenu après certaines prédictions correspondant à v, et ensuite il a disparu $(\vec{k}'_i)$. Le problème est alors que nous avons besoin d'un nombre infini de règles de ce genre puisqu'il existe un nombre infini de valuations v possibles des compteurs.

C'est pour ces mêmes raisons que nous imposons qu'il n'y ait pas de règles de la forme $0 \to u$ dans $\mathcal{R}$. En effet, dans ce cas, nous n'avons pas de règles de la forme $u \to 0$ dans $\mathcal{R}^{-1}$, et donc après avoir réécrit X en $\tilde{0}$ et Y en t après l'application d'une règle de la forme $X||Y \to t$ (ou $Y||X \to t$), nous sommes sûrs que le terme t ne va pas être réécrit en 0 et donc disparaître ($\sim$ tient compte des équivalences $\sim_0$). Par conséquent, à la place de t il y aura toujours soit t soit un successeur de t. C'est ce qui assure que l'automate peut toujours deviner à cette place qu'il y a eu une application non locale de la règle $X||Y \to t$.

5.6.6 Exemple

Nous expliquons comment l'automate se comporte concrètement sur un exemple : Soit $\mathcal{A} = (Q,\Sigma,F,\delta)$ un automate d'arbres fini qui reconnaît le terme t_1 de la figure 5.6, où $Q = \{q_1,\ldots,q_{11}\}$, $F = \{q_{11}\}$, et δ contient les règles suivantes :

- $X_i \to q_i, i \in \{1,\ldots,5,10\}$,
- $||(q_1,q_2) \to q_6, ||(q_6,q_3) \to q_7, \cdot(q_7,q_4) \to q_8, \cdot(q_8,q_5) \to q_9, ||(q_9,q_{10}) \to q_{11}$.

Soit $\mathcal{R}$ un PAD tel que $\mathcal{R}^{-1}$ contient les règles suivantes :

- $\mathcal{R}_1 : X_1||X_3 \to X_6$,

– $\mathcal{R}_2 : X_2||X_6 \to X_7$,
– $\mathcal{R}_3 : X_7 \cdot X_4 \to X_8$,
– $\mathcal{R}_4 : X_8 \cdot X_5 \to X_{11}$,
– $\mathcal{R}_5 : X_{11}||X_{10} \to X_{12}$.

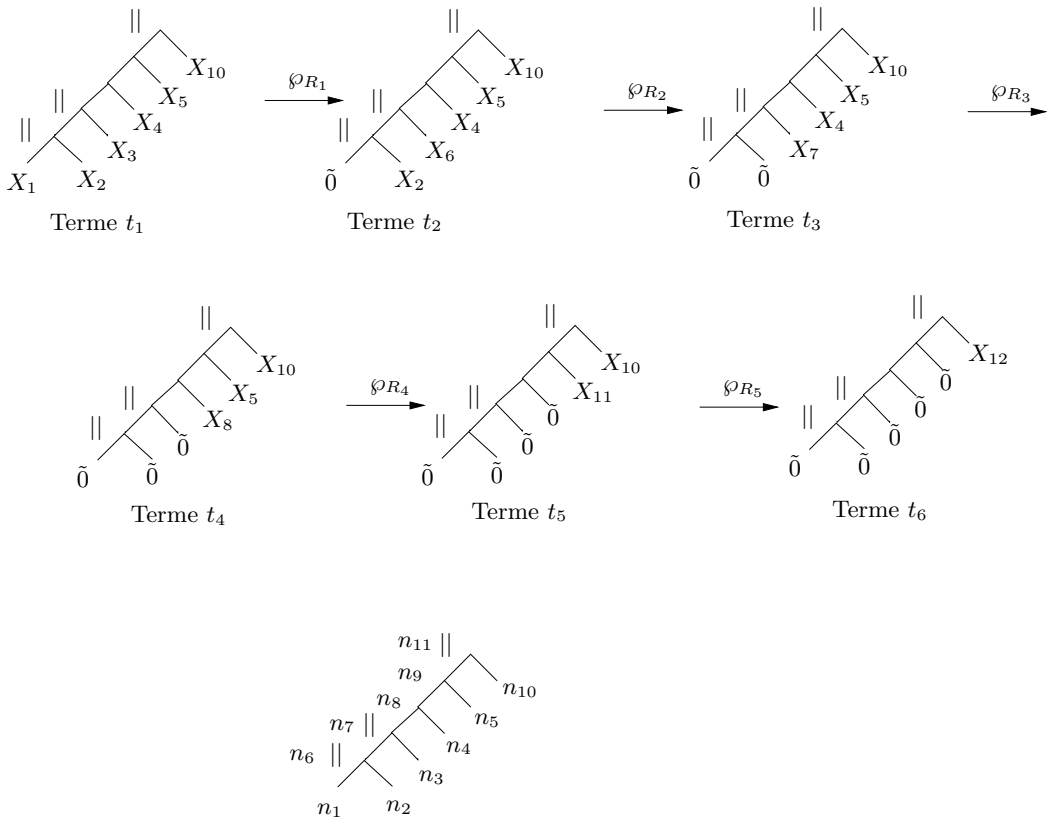


FIG. 5.6 – Exemple

Pour reconnaître le terme t_6 comme successeur de t_1, l'automate se comporte comme suit :

– Il devine qu'au nœud n_1, il y avait un X_1 qui a été réécrit par $\mathcal{R}_1$. n_1 est alors annoté par $\left((q_1, X_1, \emptyset, \emptyset), \vec{k}'_1\right)$ (le compteur de X_1 est décrémenté).
– De la même manière, il devine qu'au nœud n_2 il y avait un X_2 qui a été réécrit par $\mathcal{R}_2$. n_2 est alors annoté par $\left((q_2, X_2, \emptyset, \emptyset), \vec{k}'_2\right)$ (le compteur de X_2 est décrémenté).
– A n_3, il devine qu'il y avait un X_7 qui a été réécrit par $\mathcal{R}_3$. n_3 est alors annoté par $(q_{X_7}, -X_7)$. Ensuite, l'automate devine qu'à cette place, il y avait un X_6 qui a été transformé en X_7 en utilisant la règle $\mathcal{R}_2$. Le nœud est alors annoté par $\left((q_{X_6}, \perp, \perp \to X_2, -X_7), \vec{k}_2\right)$ (le compteur de X_2 est incrémenté). Finalement,

l'automate devine qu'à ce nœud il y avait un X_3 qui a été réécrit en X_6 en utilisant $\mathcal{R}_1$: ce nœud est alors annoté par $\Big((q_3, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7), \vec{k}_1 + \vec{k}_2\Big)$ (le compteur de X_1 est incrémenté, et X_3 est reconnu par q_3).

- Le nœud n_6 est alors annoté par $\Big((q_6, \tilde{0}, \emptyset, \emptyset), \vec{k}'_1 + \vec{k}'_2\Big)$ à cause des annotations de n_1 et n_2 et du fait que $||(q_1, q_2) \to q$ est une règle de l'automate initial. Le $\tilde{0}$ dans la deuxième composante exprime que le terme ayant n_6 comme racine est équivalent à $\tilde{0}$, c-à-d., il n'a que des $\tilde{0}$ aux feuilles.
- n_7 est annoté par $\Big((q_7, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7), \vec{k}'_1 + \vec{k}'_2 + \vec{k}_1 + \vec{k}_2\Big)$. Ceci est dû aux annotations de n_6 et n_3 et au fait que $||(q_6, q_3) \to q_7$ est une règle de l'automate initial. Alors, puisque la valeur du compteur est nulle ($\vec{k}'_1 + \vec{k}'_2 + \vec{k}_1 + \vec{k}_2 = \vec{0}$), et puisque dans le graphe $G = \{\bot \to X_1, \bot \to X_2\}$, il existe un chemin menant de $\bot$ à chaque sommet de $Noeuds(G) = \{X_1, X_2\}$, c-à-d. $G \in \mathcal{G}_c$; les prédictions peuvent être validée à ce nœud, et l'automate annote n_7 par $(q_7, -X_7)$.
- A n_4, l'automate devine qu'il y avait un X_8 qui a été réécrit par $\mathcal{R}_4$. n_4 est alors annoté par $(q_{X_8}, -X_8)$. Ensuite, l'automate devine qu'à cette place il y avait un X_4 qui a été transformé en X_8 en utilisant la règle $\mathcal{R}_3$. Le nœud est alors annoté par $\big((q_4, X_7), -X_8\big)$ puisque X_4 est reconnu par q_4.
- Par conséquent, n_8 est annoté par $(q_8, -X_8)$: la prédiction concernant X_7 est validée.
- A n_5, l'automate devine qu'il y avait un X_{11} qui a été réécrit par $\mathcal{R}_5$. n_5 est alors annoté par $\big((q_{X_{11}}, X_{11}, \emptyset, \emptyset), \vec{k}'_{11}\big)$ (le compteur de X_{11} est décrémenté). Ensuite, l'automate devine qu'à la place de ce X_{11}, il y avait un X_5 qui a été réécrit par $\mathcal{R}_4$. Le nœud est alors annoté par $\Big(\big((q_5, X_8), X_{11}, \emptyset, \emptyset\big), \vec{k}'_{11}\Big)$ puisque X_5 est reconnu par q_5.
- D'où, n_9 est annoté par $\Big(\big(q_9, \bot, \emptyset, \emptyset\big), \vec{k}'_{11}\Big)$: la prédiction concernant X_8 est validée.
- En ce qui concerne n_{10}, l'automate devine qu'à ce nœud, il y avait un X_{10} qui a été réécrit en utilisant $\mathcal{R}_5$. Le nœud est donc annoté par $\Big((q_{10}, \bot, \{\bot \to X_{11}\}, \emptyset), \vec{k}_{11}\Big)$: le compteur de X_{11} est incrémenté.
- Finalement, n_{11} est annoté par $\Big((q_{11}, \bot, \{\bot \to X_{11}\}, \emptyset), \vec{k}_{11} + \vec{k}'_{11}\Big)$. Les prédictions peuvent être validées puisque $\vec{k}_{11} + \vec{k}'_{11} = \vec{0}$, et dans le graphe $\{\bot \to X_{11}\}$, il existe un chemin menant de $\bot$ à chaque sommet de $Noeuds(\{\bot \to X_{11}\})$, (c-à-d., X_{11}).

Maintenant, nous calculons les règles $\hat{\delta}$ (nous ne considérons que les règles dont nous avons besoin pour annoter les termes de la figure 5.6) :

(r_1) Puisque $X_1 \xrightarrow{*}_{\tilde{\delta}} q_1^T$ et $X_1||X_3 \to X_6$ est une règle de $\mathcal{R}^{-1}$, les règles $\gamma_1 ai$ impliquent que $\hat{\delta}$ contient la règle

$$\tilde{0} \overset{vrai/\lambda'_1}{\to} (q_1, X_1, \emptyset, \emptyset)$$

(r_2) Puisque $X_2 \xrightarrow{*}_{\tilde{\delta}} q_2^T$ et $X_2||X_6 \to X_7$ est une règle de $\mathcal{R}^{-1}$, les règles $\gamma_1 a$i impliquent que $\hat{\delta}$ contient la règle

$$\tilde{0} \overset{vrai/\lambda_2'}{\to} (q_2, X_2, \emptyset, \emptyset)$$

(r_3) Puisque $X_7 \xrightarrow{*}_{\tilde{\delta}} q_{X_7}^T$ ($X_7 \in Sub_r(\mathcal{R}^{-1})$) et $X_7 \cdot X_4 \to X_8$ est une règle de $\mathcal{R}^{-1}$, les règles ($\gamma_2 a$) impliquent que $\hat{\delta}$ contient la règle

$$\tilde{0} \to (q_{X_7}, -X_7)$$

(r_4) Puisque $X_6 \xrightarrow{*}_{\tilde{\delta}} q_{X_6}^T$ et $X_2||X_6 \to X_7$ est une règle de $\mathcal{R}^{-1}$, les règles $\gamma_1 a$iv impliquent que $\hat{\delta}$ contient la règle

$$(q_{X_7}, -X_7) \overset{vrai/\lambda_2}{\to} (q_{X_6}, \bot, \bot \to X_2, -X_7)$$

(r_5) Puisque $X_3 \xrightarrow{*}_{\tilde{\delta}} q_3^T$ et $X_1||X_3 \to X_6$ est une règle de $\mathcal{R}^{-1}$, les règles $\gamma_1 a$ii impliquent que $\hat{\delta}$ contient la règle

$$(q_{X_6}, \bot, \bot \to X_2, -X_7) \overset{vrai/\lambda_1}{\to} (q_3, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7)$$

(r_6) Puisque $X_4 \xrightarrow{*}_{\tilde{\delta}} q_4$ et $X_7 \cdot X_4 \to X_8$ est une règle de $\mathcal{R}^{-1}$, les règles (α_2) impliquent que $\hat{\delta}$ contient la règle $X_8 \to (q_4, X_7)$. Puisque $X_8 \cdot X_5 \to X_{11}$ est une règle de $\mathcal{R}^{-1}$, les règles ($\gamma_2 b$) impliquent que $\hat{\delta}$ contient la règle

$$\tilde{0} \to \big((q_4, X_7), -X_8\big)$$

(r_7) Puisque $X_5 \xrightarrow{*}_{\tilde{\delta}} q_5$ et $X_8 \cdot X_5 \to X_{11}$ est une règle de $\mathcal{R}^{-1}$, les règles (α_2) impliquent que $\hat{\delta}$ contient la règle $X_{11} \to (q_5, X_8)$. Puisque $X_{11}||X_{10} \to X_{12}$ est une règle de $\mathcal{R}^{-1}$, les règles $\gamma_1 b$i impliquent que $\hat{\delta}$ contient la règle

$$\tilde{0} \overset{vrai/\lambda_{11}'}{\to} \big((q_5, X_8), X_{11}, \emptyset, \emptyset\big)$$

(r_8) Puisque $X_{10} \xrightarrow{*}_{\tilde{\delta}} q_{10}^T$ et $X_{11}||X_{10} \to X_{12}$ est une règle de $\mathcal{R}^{-1}$, les règles $\gamma_1 a$iii impliquent que $\hat{\delta}$ contient la règle

$$q_{X_{12}}^T \overset{vrai/\lambda_{11}}{\to} (q_{10}, \bot, \bot \to X_{11}, \emptyset)$$

(r_9) Puisque $\{\bot \to X_1, \bot \to X_2\} \in \mathcal{G}_c$, les règles ($\gamma_4$) impliquent que $\hat{\delta}$ contient la règle

$$(q_7, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7) \overset{\vec{c}=\vec{0}/\lambda}{\to} (q_7, -X_7)$$

(r_{10}) Puisque $\{\bot \to X_{11}\} \in \mathcal{G}_c$, les règles ($\gamma_4$) impliquent que $\hat{\delta}$ contient la règle

$$(q_{11}, \bot, \bot \to X_{11}, \emptyset) \overset{\vec{c}=\vec{0}/\lambda}{\to} q_{11}^T$$

(r_{11}) Puisque $||(q_1, q_2) \to q_6$ est une règle de δ, les règles ($\gamma_5 d$) impliquent que $\hat{\delta}$ contient la règle

$$||\big((q_1, X_1, \emptyset, \emptyset), (q_2, X_2, \emptyset, \emptyset)\big) \to (q_6, \tilde{0}, \emptyset, \emptyset)$$

(r_{12}) Puisque $||(q_6, q_3) \to q_7$ est une règle de δ, les règles ($\gamma_5 f$) impliquent que $\hat{\delta}$ contient la règle

$$||\big((q_6, \tilde{0}, \emptyset, \emptyset), (q_3, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7)\big) \to (q_7, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7)$$

(r_{13}) Puisque $\cdot(q_7, q_4) \to q_8$ est une règle de δ, les règles ($\gamma_6 c$) impliquent que $\hat{\delta}$ contient la règle

$$\cdot\Big((q_7, -X_7), \big((q_4, X_7), -X_8\big)\Big) \to (q_8, -X_8)$$

(r_{14}) Puisque $\cdot(q_8, q_5) \to q_9$ est une règle de δ, les règles ($\gamma_6 a$) impliquent que $\hat{\delta}$ contient la règle

$$\cdot\Big((q_8, -X_8), \big((q_5, X_8), X_{11}, \emptyset, \emptyset\big)\Big) \to \big(q_9, X_{11}, \emptyset, \emptyset\big)$$

(r_{15}) Puisque $||(q_9, q_{10}) \to q_{11}$ est une règle de δ, les règles ($\gamma_5 c$) impliquent que $\hat{\delta}$ contient la règle

$$||\Big(\big(q_9, X_{11}, \emptyset, \emptyset\big), (q_{10}, \bot, \bot \to X_{11}, \emptyset)\Big) \to \big(q_{11}, \bot, \bot \to X_{11}, \emptyset\big)$$

Avec ces règles, tous les termes de la figure (5.6) peuvent être reconnus comme successeurs du terme t_1. Plus précisément, tous ces termes peuvent être annotés par l'état final q_{11}^T. Nous montrons ci-dessous comment le terme t_6 est reconnu par cet état :

- n_1 est annoté par $\big((q_1, X_1, \emptyset, \emptyset), \vec{k}_1'\big)$ grâce aux règles r_1.
- n_2 est annoté par $\big((q_2, X_2, \emptyset, \emptyset), \vec{k}_2'\big)$ grâce aux règles r_2.
- En utilisant les règles r_3, r_4, et r_5, n_3 peut être annoté par

$$\Big((q_3, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7), \vec{k}_1 + \vec{k}_2\Big)$$

- n_4 est annoté par $\big((q_4, X_7), -X_8\big)$ grâce aux règles r_6.
- n_5 est annoté par $\Big(\big((q_5, X_8), X_{11}, \emptyset, \emptyset\big), \vec{k}_{11}'\Big)$ en utilisant r_7.
- r_{11} annote n_6 avec $\Big((q_6, \tilde{0}, \emptyset, \emptyset), \vec{k}_1' + \vec{k}_2'\Big)$.
- n_7 est d'abord annoté par $\Big((q_7, \bot, \{\bot \to X_1, \bot \to X_2\}, -X_7), \vec{k}_1' + \vec{k}_2' + \vec{k}_1 + \vec{k}_2\Big)$ en utilisant r_{12}, et ensuite par $(q_7, -X_7)$ en utilisant r_9 puisque $\vec{k}_1' + \vec{k}_2' + \vec{k}_1 + \vec{k}_2 = \vec{0}$.
- Grâce à r_{13}, n_8 est annoté par $(q_8, -X_8)$.
- n_9 est annoté par $\Big(\big(q_9, X_{11}, \emptyset, \emptyset\big), \vec{k}_{11}'\Big)$ grâce à r_{14}.
- n_{10} est annoté par $\Big((q_{10}, \bot, \{\bot \to X_{11}\}, \emptyset), \vec{k}_{11}\Big)$ grâce à r_8.
- Finalement n_{11} est d'abord annoté par $\Big(\big(q_{11}, \bot, \bot \to X_{11}, \emptyset\big), \vec{k}_{11} + \vec{k}_{11}'\Big)$ grâce à r_{15}, et ensuite par q_{11}^T en utilisant r_{10} puisque $\vec{k}_{11} + \vec{k}_{11}' = \vec{0}$.

5.6.7 La preuve du théorème 5.6.4

Pour montrer le théorème 5.6.4, nous montrons les lemmes 5.6.2 et 6.5.2 :

Lemme 5.6.2 *Pour chaque $u \in \mathcal{T}'$, nous avons :*

- $u \xrightarrow{*}_{\hat{\delta}} \big((q, a, G, w), v\big)$ *alors :*
  - *Si $w = \emptyset$, il existe un paral-contexte C, n null-contextes $C'_1, \ldots, C'_n$, et un derived system $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$ tels que $v = \sum_{1 \leq i \leq n} v_i$, $G = \bigcup_{1 \leq i \leq n} G_i$, $u = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$, et $C\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_q)$. En plus, si $a = \tilde{0}$, alors pour tout i, $u_i = \tilde{0}$, et si $a \in Var$, alors C et C'_1 sont le contexte trivial ($n = 1$), et $a = Y^1_{h_1}$.*
  - *Si $w = -X$, alors $a = \perp$ et il existe un paral-contexte C, n null-contextes $C'_1, \ldots, C'_n$, et un seq-derived system $\big(X, (A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}\big)$ tels que $v = \sum_{1 \leq i \leq n} v_i$, $G = \bigcup_{1 \leq i \leq n} G_i$, $u = C\big[C'_1[\tilde{0}], \ldots, C'_n[\tilde{0}]\big]$, et $C\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_q)$.*
- $u \xrightarrow{*}_{\hat{\delta}} \Big(\big((q, W), a, G, w\big), v\Big)$ *alors il existe $A \in Var$ tel que $A \xrightarrow{*}_{\tilde{\delta}} (q, W)$ et :*
  - *si $w = \emptyset$, il existe une séquence de règles $\mathcal{S}$ telle que $(A, u, G, v, \mathcal{S})$ soit un derived system. Si $u = \tilde{0}$, alors $a = Y^1_{h_1}$ ($n = 1$), et si $u \neq \tilde{0}$, alors $a = \perp$.*
  - *Si $w = -X$, il existe $\mathcal{S}$ tel que $\big(X, (A, u, G, v, \mathcal{S})\big)$ soit un seq-derived system. Dans ce cas, $a = \perp$.*
- $u \xrightarrow{*}_{\hat{\delta}} q^T$ *alors $u \in \wp^*_{\mathcal{R}^{-1}}(L_q)$.*
- $u \xrightarrow{*}_{\hat{\delta}} (q, -X)$ *alors il existe un paral-contexte $C[x]$, tel que C est aussi un null-contexte tel que :*
  - *soit il existe un null-contexte $C_0[x]$ tel que $u = C[\cdot(C_0[\tilde{0}], \tilde{0})]$, et $C[\cdot(C_0[\tilde{0}], X)] \in \wp^*_{\mathcal{R}^{-1}}(L_q)$,*
  - *soit $u = C[\tilde{0}]$ et $C[X] \in \wp^*_{\mathcal{R}^{-1}}(L_q)$.*
- $u \xrightarrow{*}_{\hat{\delta}} \big((q, W), -X\big)$ *alors $u = \tilde{0}$ et $X \in L_{(q,W)}$.*

Preuve : Nous procédons par induction sur k, le nombre de fois qu'une règle dans $\hat{\delta} \setminus \tilde{\delta}$ est appliquée pendant les dérivations. Nous montrons toutes les propriétés simultanément :

• $k = 1$. Il y a alors cinq cas en fonction de la règle de $\hat{\delta}$ qui a été appliquée :

- La règle $\gamma_1 ai$ a été appliquée. Alors $u = \tilde{0} \rightarrow_{\hat{\delta}} \big((q, X_i, \emptyset, \emptyset), v\big)$, où $v = \vec{k}'_i$. Ceci veut dire qu'il existe une règle $X_i || X_j \rightarrow t$ (ou $X_j || X_i \rightarrow t$) telle que $X_i \xrightarrow{*}_{\tilde{\delta}} q^T$. Il est facile de voir que la propriété est vraie en prenant $n = 1$, C et C'_1 sont le contexte trivial, $A_1 = X_i$, et $h_1 = 1$. En effet, puisque $X_i \xrightarrow{*}_{\tilde{\delta}} q^T$, nous déduisons à partir du lemme 5.5.1 que $X_i \in \zeta^*_{\mathcal{R}^{-1}}(L_q) \subseteq \wp^*_{\mathcal{R}^{-1}}(L_q)$.
- Le cas où la règle $\gamma_1 bi$ a été appliquée est similaire au cas précédent.
- La règle $\gamma_1 aiii$ a été appliquée. Alors $u \xrightarrow{*}_{\tilde{\delta}} q^T_t \rightarrow_{\hat{\delta}} \big((q, \perp, \{\perp \rightarrow X_i\}, \emptyset), v\big)$, où $v = \vec{k}_i$. Ceci implique qu'il existe une règle $X_i || X_j \rightarrow t$ (ou $X_j || X_i \rightarrow t$) telle que $X_j \xrightarrow{*}_{\tilde{\delta}} q^T$. Il est facile de voir que la propriété est vraie en prenant $n = 1$, C et C'_1 sont le contexte trivial, $A_1 = X_j$, et $h_1 = 1$. En effet, puisque $X_j \xrightarrow{*}_{\tilde{\delta}} q^T$,

nous avons par le lemme 5.5.1 que $X_j \in \zeta^*_{\mathcal{R}^{-1}}(L_q) \subseteq \wp^*_{\mathcal{R}^{-1}}(L_q)$. En plus, puisque $u \xrightarrow{*}_{\tilde{\delta}} q_t^T$, nous avons $u \in \zeta^*_{\mathcal{R}^{-1}}(t)$.
- Le cas où la règle $\gamma_1 b$iii a été appliquée est similaire au cas précédent.
- La règle $\gamma_2 a$ a été appliquée, c-à-d., $u = \tilde{0} \rightarrow_{\hat{\delta}} (q, -X)$. La propriété est vraie en prenant C le contexte trivial. En effet, si cette règle a été appliquée, alors $X \xrightarrow{*}_{\tilde{\delta}} q^T$, c-à-d., $X \in \zeta^*_{\mathcal{R}^{-1}}(L_q) \subseteq \wp^*_{\mathcal{R}^{-1}}(L_q)$.
- La règle $\gamma_2 b$ a été appliquée, c-à-d., $u \rightarrow_{\hat{\delta}} \big((q, W), -X\big)$. Alors nécessairement, $u = \tilde{0}$, et $X \xrightarrow{*}_{\tilde{\delta}} (q, W)$. La propriété est vérifiée.

• $k > 1$.
⋆ Soit $u \xrightarrow{k}_{\hat{\delta}} \big((q, a, G, w), v\big)$. Il y a alors différents cas selon la règle de l'automate qui a été appliquée à la fin.

- La règle $\gamma_1 a$ii a été appliquée à la fin : $u \xrightarrow{k-1}_{\hat{\delta}} \big((q_t, a, G', w), v'\big) \rightarrow_{\hat{\delta}} \big((q, a, G, w), v\big)$ tel qu'il existe une règle $X_i||X_j \rightarrow t$ (ou $X_j||X_i \rightarrow t$) telle que $X_j \xrightarrow{*}_{\hat{\delta}} q^T$, $v = v' + \vec{k}_i$, et $G = G' \cup \{a \rightarrow X_i\}$. Alors, par induction nous avons :
  - Si $w = \emptyset$, il existe un derived system $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$ tel que $u = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$, $G' = \bigcup_{1 \leq i \leq n} G_i$, $v' = \sum_{1 \leq i \leq n} v_i$, et $C\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_{q_t}) = \wp^*_{\mathcal{R}^{-1}}(t)$. Nous avons que $\wp^*_{\mathcal{R}^{-1}}(t) = \zeta^*_{\mathcal{R}^{-1}}(t)$ puisque $t \in Sub_r(\mathcal{R}^{-1})$, et $\mathcal{R}^{-1}$ ne contient aucune règle ayant $X||Y$ dans le côté droit. C-à-d., $\mathcal{R}^{-1}$ ne rajoute pas de "||" dans les termes. Donc, C et C'_1 sont le contexte trivial ($n = 1$). En effet, C est le contexte trivial et C'_1 est un null-context, c-à-d., un contexte qui ne contient que des $\tilde{0}$ aux feuilles. Mais les $\tilde{0}$ ne sont rajoutés que lorsque nous avons l'opérateur $||$.
    Nous allons donc enlever l'indice i. Soit $A = A_1 \in \wp^*_{\mathcal{R}^{-1}}(t) = \zeta^*_{\mathcal{R}^{-1}}(t)$ (ici $u = u_1$), et soit $\mathcal{S}$ une séquence de règles

$$\big(Y_j||Z_j \rightarrow t_j \text{ (ou } Z_j||Y_j \rightarrow t_j \text{)}\big)_{1 \leq j \leq h}$$

    telle que :
    - Si $u \neq \tilde{0}$, alors $a = \perp$ et :
      - $A = Z_1$,
      - $v' = \sum_{1 \leq j \leq h} \vec{k}_{l_j}$,
      - $G' = \{\perp \rightarrow Y_s \mid 1 \leq s \leq h\}$,
      - $\forall 0 < j < h$, $Z_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t_j)$,
      - $u \in \zeta^*_{\mathcal{R}^{-1}}(t_h)$.
    - Si $u = \tilde{0}$, alors $a = Y_h$ et :
      1. Si $h > 1$, alors :
         - $A = Z_1$,
         - $v' = \sum_{1 \leq j \leq h-1} \vec{k}_{l_j} + \vec{k}'_{l_h}$,
         - $G' = \{Y_h \rightarrow Y_s \mid 1 \leq s \leq h-1\}$,
         - $\forall 0 < j < h-1$, $Z_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t_j)$,
         - $Y_h \in \zeta^*_{\mathcal{R}^{-1}}(t_{h-1})$.
      2. Si $h = 1$, alors $a = Y_1$ et :

- $A = Y_1$,
- $v' = \vec{k}'_{l_h}$,
- $G' = \emptyset$.

Considérons la séquence de règles $\mathcal{S}'$:

$$\big(Y'_j||Z'_j \to t'_j \text{ (ou } Z'_j||Y'_j \to t'_j \text{)}\big)_{1\leq j\leq h+1}$$

telle que la règle $Y'_1||Z'_1 \to t'_1$ est la règle $X_i||X_j \to t$ (ou $Z'_1||Y'_1 \to t'_1$ est la règle $X_j||X_i \to t$), et pour tout $1 < j \leq h+1$, $Y'_j = Y_{j-1}$, $Z'_j = Z_{j-1}$, et $t'_j = t_{j-1}$. Soient $(e_m)_m$ les indices tels que $e_m = \mathit{indice}(Y'_m)$ $(e_1 = i)$. Nous avons $v = v' + \vec{k}_i = v' + \vec{k}_{e_1}$, et $G = G' \cup \{a \to X_i\}$. Soit $A' = X_j = Z'_1 \in \zeta^*_{\mathcal{R}^{-1}}(L_q) \subseteq \wp^*_{\mathcal{R}^{-1}}(L_q)$ (puisque $X_j \xrightarrow{*}_{\tilde{\delta}} q^T$). Nous avons déjà que $Z'_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t'_j)$, pour tout j, $1 < j < h+1$. En plus,

$$Z'_2 = Z_1 = A \in \zeta^*_{\mathcal{R}^{-1}}(t)$$

Donc, puisque $t = t'_1$, nous avons que pour tout $0 < j < h+1$, $Z'_{j+1} \in \zeta^*_{\mathcal{R}^{-1}}(t'_j)$.

- Si $u \neq \tilde{0}$, nous avons :
  - $v = \sum_{1\leq j\leq h+1} \vec{k}_{e_j}$,
  - $G = \{\bot \to Y'_s \mid 1 \leq s \leq h+1\}$ $(Y'_1 = X_i)$,
  - $u \in \zeta^*_{\mathcal{R}^{-1}}(t'_{h+1})$, puisque $t'_{h+1} = t_h$.
- Si $u = \tilde{0}$, alors :
  - $v = \sum_{1\leq j\leq h} \vec{k}_{e_j} + \vec{k}'_{e_{h+1}}$,
  - $G = \{Y'_{h+1} \to Y'_s \mid 1 \leq s \leq h\}$ $(a = Y_h = Y'_{h+1})$,
  - $Y'_{h+1} \in \zeta^*_{\mathcal{R}^{-1}}(t'_h)$.

  Donc, $(A', u, G, v, \mathcal{S}')$ est un derived system, et la propriété est satisfaite.
- Le cas où $w = -Y$ est similaire.

- La règle γ_3 a été appliquée à la fin. Alors $u \overset{k-1}{\to}_{\hat{\delta}} \big((q', a, G, w), v\big) \to_{\hat{\delta}} \big((q, a, G, w), v\big)$ parce que ${q'}^T \to_{\tilde{\delta}} q^T$, c-à-d., $\zeta^*_{\mathcal{R}^{-1}}(L_{q'}) \subseteq \zeta^*_{\mathcal{R}^{-1}}(L_q)$.
  Nous n'allons considérer que le cas où $w = \emptyset$. L'autre cas est similaire. Par induction, nous déduisons qu'il existe un derived system $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1\leq i\leq n}$, un paral-contexte C, et n null-contextes $C'_1, \ldots, C'_n$ tels que $G = \bigcup_{1\leq i\leq n} G_i$, $v = \sum_{1\leq i\leq n} v_i$, $u = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$, et $C\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_{q'})$ qui est inclu dans $\wp^*_{\mathcal{R}^{-1}}(L_q)$, puisque $\zeta^*_{\mathcal{R}^{-1}}(L_{q'}) \subseteq \zeta^*_{\mathcal{R}^{-1}}(L_q)$.
- La règle γ_5 a été appliquée à la fin. Il existe alors deux cas selon que ce sont les deux premières règles qui ont été appliquées ou les autres (dans ce qui suit, nous n'allons pas tenir compte de la nature de la composante a, qui est facile à vérifier) :
  - Une des règles $\gamma_5 a$ ou $\gamma_5 b$ a été appliquée. Ces règles sont symétriques. Supposons par exemple que la règle $\gamma_5 a$ a été considérée. Alors

$$u = ||(u', u'') \overset{k-1}{\to}_{\hat{\delta}} ||\Big(\big((q', a', G', \emptyset), v'\big), {q''}^T\Big) \to_{\hat{\delta}} \big((q, a, G, \emptyset), v\big)$$

Il s'en suit que $||(q', q'') \to q$ est une règle de δ_{null}, $v = v'$, et $G = G'$. Prenons le cas où $w = \emptyset$ (l'autre cas étant similaire). Par induction, nous avons que $u'' \in \wp^*_{\mathcal{R}^{-1}}(L_{q''})$, et il existe un derived system $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \le i \le n}$, un paral-contexte C, et n null-contextes $C'_1, \ldots, C'_n$ tels que $G = \bigcup_{1 \le i \le n} G_i$, $v = \sum_{1 \le i \le n} v_i$, $u' = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$, et

$$C\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_{q'}).$$

Soit D le paral-contexte $D[x_1, \ldots, x_n] = ||(C[x_1, \ldots, x_n], u'')$. Alors

$$D\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_q),$$

$u = D\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$, et $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \le i \le n}$ est un derived system qui satisfait la propriété.

– Une des quatre dernières règles a été appliquée :

$$u \overset{k-1}{\to}_{\hat{\delta}} ||\Big(\big((q', a', G', w'), v'\big), \big((q'', a'', G'', w''), v''\big)\Big) \to_{\hat{\delta}} \big((q, a, G, w), v\big)$$

Alors $||(q', q'') \to q$ est une règle de δ_{null}, $v = v' + v''$, $G = G' \cup G''$, et a et w dépendent des valeurs de a', a'', w', et w''. Supposons par exemple que la règle $\gamma_5 c$ a été considérée. Nous obtenons par induction qu'il existe des derived systems $(A'_i, u'_i, G'_i, v'_i, \mathcal{S}'_i)_{1 \le i \le n'}$ et $(A''_i, u''_i, G''_i, v''_i, \mathcal{S}''_i)_{1 \le i \le n''}$, deux paral-contextes C' et C'', et $n' + n''$ null-contextes $D'_1, \ldots, D'_{n'}$ et $D''_1, \ldots, D''_{n''}$ tels que $G' = \bigcup_{1 \le i \le n'} G'_i$, $v' = \sum_{1 \le i \le n'} v'_i$, $G'' = \bigcup_{1 \le i \le n''} G''_i$, $v'' = \sum_{1 \le i \le n''} v''_i$, $u' = C'\big[D'_1[u'_1], \ldots, D'_{n'}[u'_{n'}]\big]$, $u'' = C''\big[D''_1[u''_1], \ldots, D''_{n''}[u''_{n''}]\big]$,

$$C'\big[D'_1[A'_1], \ldots, D'_{n'}[A'_{n'}]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_{q'}),$$

et

$$C''\big[D''_1[A''_1], \ldots, D''_{n''}[A''_{n''}]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_{q''}).$$

Posons $n = n' + n''$, et C le paral-contexte

$$C[x_1, \ldots, x_n] = ||(C'[x_1, \ldots, x_{n'}], C''[x_{n'+1}, \ldots, x_{n''}]).$$

Alors, il est facile de voir que $u = C\big[D_1[u_1], \ldots, D_n[u_n]\big]$, et que $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \le i \le n}$ est un derived system, où les nouvelles séquences $(\Box_i)_{1 \le i \le n}$ ($\Box \in \{D, u, A, \mathcal{S}\}$) sont définies comme suit :

– $\Box_i = \Box'_i$ si $1 \le i \le n'$,
– $\Box_i = \Box''_{n'+i}$ sinon.

En plus, nous avons que

$$C\big[D_1[A_1], \ldots, D_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_q).$$

Donc, la propriété est satisfaite.

– La règle $\gamma_6 a$ a été appliquée. Alors

$$u = \cdot(u', u'') \overset{k-1}{\rightarrow}_{\hat{\delta}} \cdot\Big((q', -X), \big(((q'', X), a'', G'', w''), v''\big)\Big) \rightarrow_{\hat{\delta}} \big((q, a, G, w), v\big)$$

Il s'en suit que $\cdot(q', q'') \rightarrow q$ est une règle de δ_{null}, $v = v''$, et $G = G''$. Par induction, nous déduisons qu'il existe un paral-contexte $C'[x]$ qui soit aussi un null-contexte, et un null-contexte C_0 tel que $C_0[\tilde{0}] = u_0$, $u' = C'[\cdot(u_0, \tilde{0})]$, et $C'[\cdot(u_0, X)] \in \wp^*_{\mathcal{R}^{-1}}(L_{q'})$ (le cas où $u' = C'[\tilde{0}]$, et $C'[X] \in \wp^*_{\mathcal{R}^{-1}}(L_{q'})$ est identique). Supposons que $w = \emptyset$ (l'autre cas se traite de la même manière), nous avons par induction qu'il existe $A \in Var$ tel que $A \overset{*}{\rightarrow}_{\tilde{\delta}} (q'', X)$ et il existe une séquence de règles $\mathcal{S}$ telle que $(A, u'', G, v, \mathcal{S})$ soit un derived system.
Soit C'_1 le null-contexte $C'_1[x] = \cdot(u', x)$. Il est facile de voir que $(A, u'', G, v, \mathcal{S})$ est un derived system tel que $u = C'_1[u'']$. En plus, $C'_1[A] \in \wp^*_{\mathcal{R}^{-1}}(L_q)$. En effet, nous avons que $A \overset{*}{\rightarrow}_{\tilde{\delta}} (q'', X)$, alors le lemme 5.5.1 implique qu'il existe un seq-contexte $C_s[x]$, une règle $X \cdot Y \rightarrow t$ dans $\mathcal{R}^{-1}$ tels que $A \in \zeta^*_{\mathcal{R}^{-1}}(C_s[t])$, et $C_s[Y] \in L_{q''}$. Donc,

$$\cdot(C'[\cdot(u_0, X)], C_s[Y]) \in \wp^*_{\mathcal{R}^{-1}}(L_q).$$

Puisque u_0 est un terme nul, nous pouvons appliquer la règle $X \cdot Y \rightarrow t$ à ce terme, et obtenir que

$$\cdot(C'[\cdot(u_0, \tilde{0})], C_s[t]) \in \wp^*_{\mathcal{R}^{-1}}(L_q)$$

Puisque $A \in \zeta^*_{\mathcal{R}^{-1}}(C_s[t])$, nous obtenons que

$$\cdot(C'[\cdot(u_0, \tilde{0})], A) \in \wp^*_{\mathcal{R}^{-1}}(L_q).$$

Observons que dans le terme précédent, le fils droit $C_s[t]$ peut être réécrit puisque le fils gauche est nul (u_0 est nul). Donc, la propriété est satisfaite dans ce cas.

$\star$ Le cas où $u \overset{k}{\rightarrow}_{\hat{\delta}} \Big(\big((q, X), a, G, w\big), v\Big)$ peut se traiter de la même manière.

$\star$ Soit $u \overset{k}{\rightarrow}_{\hat{\delta}} q^T$. Le cas le plus intéressant est lorsque nous avons :

$$u \overset{k-1}{\rightarrow}_{\hat{\delta}} \big((q, a, G, w), v\big) \rightarrow_{\hat{\delta}} q^T.$$

Alors nécessairement, la règle γ_4 a été appliquée. Nous obtenons que $G \in \mathcal{G}_c$, et $v = \vec{0}$. Par induction, nous déduisons qu'il existe un derived system $(A_i, u_i, G_i, v_i, \mathcal{S}_i)_{1 \leq i \leq n}$, un paral-contexte C, et n null-contextes $(C'_i)_{1 \leq i \leq n}$ tels que $G = \bigcup_{1 \leq i \leq n} G_i$, $v = \sum_{1 \leq i \leq n} v_i$, $u = C\big[C'_1[u_1], \ldots, C'_n[u_n]\big]$ et

$$C\big[C'_1[A_1], \ldots, C'_n[A_n]\big] \in \wp^*_{\mathcal{R}^{-1}}(L_q).$$

En plus, puisque $G \in \mathcal{G}_c$, et $v = \vec{0}$, nous déduisons à partir du lemme 5.6.1 que

$$u \in \wp^*_{\mathcal{R}^{-1}}(C\big[C'_1[A_1], \ldots, C'_n[A_n]\big]).$$

Ceci implique que $u \in \wp^*_{\mathcal{R}^{-1}}(L_q)$.
$\star$ Les cas où $u \xrightarrow{k}_{\hat{\delta}} \big(q, -X\big)$ et $u \xrightarrow{k}_{\hat{\delta}} \big((q, Y), -X\big)$ sont faciles à vérifier. Grâce aux règles γ_4, nous utilisons comme précédemment le lemme 5.6.1 pour traiter le premier cas. □

Lemme 5.6.3 *Soit $u \in \mathcal{T}'$, alors $u \in \wp^*_{\mathcal{R}^{-1}}(L_q) \Rightarrow u \xrightarrow{*}_{\hat{\delta}} q^T$.*

Preuve : Puisque $L_q \subseteq L_{q^T}$, nous montrons que si $u' \xrightarrow{*}_{\hat{\delta}} q^T$ et $u \in \wp_{\mathcal{R}^{-1}}(u')$, alors $u \xrightarrow{*}_{\hat{\delta}} q^T$. Nous n'allons considérer que les réécritures décrites sur les figures 5.4 et 5.5 puisqu'elles sont les plus pertinentes. Nous pouvons traiter les autres règles comme nous l'avons fait dans les lemmes précédents qui ne considèrent pas l'équivalence $\sim$. Il y a deux cas cas selon le type de la règle qui a été appliquée de manière non locale :
• Une règle $X_i||X_j \to t$ a été appliquée non localement comme décrit sur la figure 5.4. Soit alors C un paral-contexte ; C_1 et C_2 des null-contextes tels que chacun a soit "$\cdot$" comme racine, soit est le contexte trivial ; et des termes u_i tels que chacun est soit une variable de processus de Var, soit a "$\cdot$" comme racine (c-à-d., comme nous allons le voir, C est un paral-contexte "maximal") tels que :

Cas 1 : soit $u' = C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}]$ et

$$u = C[u_1, \ldots, u_{i_1}, C_1[\tilde{0}], u_{i_2}, \ldots, u_{i_3}, C_2[t], u_{i_4}, \ldots, u_{i_5}].$$

Dans ce cas, il faut montrer que si

$$C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}] \xrightarrow{*}_{\hat{\delta}} q^T$$

alors

$$C[u_1, \ldots, u_{i_1}, C_1[\tilde{0}], u_{i_2}, \ldots, u_{i_3}, C_2[t], u_{i_4}, \ldots, u_{i_5}] \xrightarrow{*}_{\hat{\delta}} q^T$$

Cas 2 : soit il existe un contexte C' et un terme u'' tels que

$$u' = C'\big[\cdot\,(C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}], u'')\big]$$

et

$$u = C'\big[\cdot\,(C[u_1, \ldots, u_{i_1}, C_1[\tilde{0}], u_{i_2}, \ldots, u_{i_3}, C_2[t], u_{i_4}, \ldots, u_{i_5}], u'')\big].$$

Dans ce cas, il faut montrer que si s est un état tel que

$$C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}] \xrightarrow{*}_{\hat{\delta}} s$$

alors

$$C[u_1, \ldots, u_{i_1}, C_1[\tilde{0}], u_{i_2}, \ldots, u_{i_3}, C_2[t], u_{i_4}, \ldots, u_{i_5}] \xrightarrow{*}_{\hat{\delta}} s$$

auquel cas, il est facile de déduire que si

$$u' \xrightarrow{*}_{\hat{\delta}} C'[\cdot(s, s')] \xrightarrow{*}_{\hat{\delta}} q^T$$

alors nous avons aussi

$$u \xrightarrow{*}_{\hat{\delta}} C'[\cdot(s, s')] \xrightarrow{*}_{\hat{\delta}} q^T$$

Dans ce cas, à partir des règles γ_6, $\alpha_4 a$, et $\alpha_4 b$, nous avons que s est soit de la forme $(p, -Z)$, soit de la forme p^T, ou (p, Z).

1. si $s = (p, -Z)$, le lemme 5.6.2 implique que

$$C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}]$$

est un terme nul. Nous obtenons donc une contradiction. Par conséquent, s ne peut pas être de la forme $(p, -Z)$.

2. si $s = (p, Z)$, le lemme 5.5.1 implique qu'il existe un seq-contexte C_s et une règle $Z \cdot T \to t'$ tels que

$$C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}] \in \zeta^*_{\mathcal{R}^{-1}}(t')$$

et $C_s[T] \in L_q$. Ceci n'est pas possible puisque $\mathcal{R}^{-1}$ ne contient aucune règle avec l'opérateur "||" dans la partie droite, donc, $\zeta^*_{\mathcal{R}^{-1}}(t')$ ne contient pas de nœuds étiquetés par "||". Ceci contredit le fait que C n'a que des nœuds étiquetés par "||". Donc, s ne peut pas être de la forme (p, Z).

D'où, la seule possibilité pour s est d'être de la forme p^T.

Par conséquent, il faut montrer dans les deux cas que si u, u' et q sont tels que

$$u = C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}] \stackrel{*}{\to}_{\hat{\delta}} q^T$$

alors

$$u'C[u_1, \ldots, u_{i_1}, C_1[\tilde{0}], u_{i_2}, \ldots, u_{i_3}, C_2[t], u_{i_4}, \ldots, u_{i_5}] \stackrel{*}{\to}_{\hat{\delta}} q^T$$

Supposons que pendant la dérivation

$$u' = C[u_1, \ldots, u_{i_1}, C_1[X_i], u_{i_2}, \ldots, u_{i_3}, C_2[X_j], u_{i_4}, \ldots, u_{i_5}] \stackrel{*}{\to}_{\hat{\delta}} q^T$$

les règles γ_4 ont été appliquées plusieurs fois pour valider certaines prédictions à la volée, et annoter les nœuds internes correspondants avec des états q_i^T, et utiliser ensuite les règles $\gamma_5 a$ et $\gamma_5 b$ pour annoter le reste de l'arbre. Alors, puisque C est un paral-contexte, ces validations peuvent être faites plus tard à la racine de l'arbre, et nous pouvons utiliser les règles $\gamma_5 c$ et $\gamma_5 d$ à la place de $\gamma_5 a$ et $\gamma_5 b$ pour annoter le reste de l'arbre. Par conséquent, nous supposons que pendant la dérivation ci-dessus, les règles γ_4 ne sont appliquées qu'à la racine de C. Nous pouvons alors distinguer entre deux cas (il y en a deux à cause des formes des règles γ_5) :

1. Pendant la dérivation $u' \stackrel{*}{\to}_{\hat{\delta}} q^T$, tous les termes u_k, $C_1[X_i]$, et $C_2[X_j]$ sont annotés par des états de la forme p^T. Supposons par exemple que C_1 n'est pas le contexte trivial, et que C_2 est le contexte trivial, les autres cas étant similaires. Soit alors $C_1[X_i] = \cdot(U_1, U_2)$. Soient q_1, q_2 tels que

$$C_1[X_i] = \cdot(U_1, U_2) \stackrel{*}{\to}_{\hat{\delta}} q_1^T,$$

et

$$C_2[X_j] = X_j \stackrel{*}{\to}_{\hat{\delta}} q_2^T.$$

Alors nécessairement, grâce au lemme 5.6.2, nous obtenons que $X_j \in \wp^*_{\mathcal{R}^{-1}}(L_{q_2})$, ce qui veut dire que $X_j \in \zeta^*_{\mathcal{R}^{-1}}(L_{q_2})$ puisque les règles de $\wp_{\mathcal{R}^{-1}} \backslash \zeta_{\mathcal{R}^{-1}}$ introduisent $\tilde{0}$. D'où,

$$X_j \stackrel{*}{\to}_{\tilde{\delta}} q_2^T.$$

A partir des règles $\gamma_1 a$iii, nous obtenons que

$$q_t^T \to_{\hat{\delta}} \Big(\big(q_2, \bot, \{\bot \to X_i\}, \emptyset \big), \vec{k}_i \Big) \tag{5.15}$$

En ce qui concerne $C_1[X_i]$, le lemme 5.6.2 implique que $C_1[X_i] \in \wp^*_{\mathcal{R}^{-1}}(L_{q_1})$. Soit $C_1[X_i] = \cdot(U_1, U_2)$, alors nous avons nécessairement que U_1 est un terme nul, et $U_2 = X_i$. En effet, puisque C_1 est un null-contexte, il contient nécessairement des $\tilde{0}$'s et donc, des nœuds étiquetés par "||". Donc, il existe un terme $\cdot(U_1', U_2')$ dans L_{q_1} tel que $\cdot(U_1, U_2) \in \wp^*_{\mathcal{R}^{-1}}\big(\cdot\, (U_1', U_2') \big)$. A cause de la règle de priorité de l'opérateur "$\cdot$", U_1' est réécrit d'abord, et ensuite, quand il devient nul, U_2' est réécrit. Ce scénario n'est pas possible puisqu'il n'y a pas de règles de la forme $t \to 0$. Donc, il n'est pas possible de réécrire U_1' en un terme nul. Par conséquent, la seule possibilité qui reste est de réécrire U_1' jusqu'à obtenir un terme de la forme $D[Z]$, où D est un null-contexte. Dans ce cas, s'il existe un seq-contexte C_s et une variable Y, tels que $U_2' = C_s[Y]$, alors s'il existe une règle $Z \cdot Y \to t'$, nous pouvons l'appliquer comme décrit sur la figure 5.5 et obtenir $\cdot(U_1, C_s[t'])$, où $U_1 = D[\tilde{0}]$. Puisque U_1 est nul, $C_s[t']$ peut être réécrit en U_2. Puisque $C_1[X_i] = \cdot(U_1, U_2)$, et C_1 est un null-contexte, $C_s[t']$ doit être réécrit en U_2, qui est supposé être un terme équivalent à X_i, avec éventuellement, quelques $\tilde{0}$. Nous pouvons montrer par induction structurelle sur C_s que $U_2 = X_i$:

- C_s est le contexte trivial, alors puisque t' ne peut pas être réécrit en un terme contenant des $\tilde{0}$ (il n'y a pas de règle avec "||" dans le côté droit). Par conséquent, t' doit être réécrit en X_i.
- $C_s[t'] = \cdot(C_s'[t'], v)$ où C_s' est un seq-contexte. Alors, par induction, $C_s'[t']$ peut seulement être réécrit en une variable. Puisqu'il ne peut pas devenir nul (pas de règles de la forme $t \to 0$), v ne peut donc pas être réécrit tout seul, il doit attendre (s'il est sous la bonne forme $(C_s''[Z'])$) que $C_s'[t']$ devienne une variable X' avec qui il peut interagir en utilisant une règle de la forme $X' \cdot Z' \to v'$. Nous obtenons alors, sans introduire de $\tilde{0}$, le terme $C_s''[v']$ avec C_s'' est un seq-contexte de teille strictement inférieure à celle de C_s.

Concernant l'annotation de $C_1[X_i] = \cdot(U_1, X_i)$, la seule règle qui peut être appliquée pour annoter la racine est $\gamma_6 b$. Soient donc p_1 et p_2 tels que $\cdot(U_1, X_i) \xrightarrow{*}_{\hat{\delta}} \cdot\big((p_1, -Z), (p_2, Z)\big)$. Ceci veut dire que $\cdot(p_1, p_2) \to q_1$ est une règle de δ_{null}, et que $X_i \xrightarrow{*}_{\tilde{\delta}} (p_2, Z)$, et donc les règles $\gamma_1 b$i impliquent que

$$\tilde{0} \to_{\hat{\delta}} \Big(\big((p_2, Z), X_i, \emptyset, \emptyset \big), \vec{k}_i' \Big)$$

Nous déduisons donc, à partir des règles $\gamma_6 a$ que :

$$C_1[\tilde{0}] = \cdot(U_1, \tilde{0}) \xrightarrow{*}_{\hat{\delta}} \cdot \Big((p_1, -Z), \Big(\big((p_2, Z), X_i, \emptyset, \emptyset \big), \vec{k}_i' \Big) \Big) \xrightarrow{*}_{\hat{\delta}} \big((q_1, X_i, \emptyset, \emptyset), \vec{k}_i' \big) \tag{5.16}$$

A partir de 5.15 et 5.16, nous pouvons montrer par induction structurelle sur C, en utilisant les règles γ_5a et γ_5b que

$$u \xrightarrow{*}_{\hat{\delta}} \big((q, \bot, \{\bot \to X_i\}, \emptyset), \vec{k}_i + \vec{k}_i' \big) \to_{\hat{\delta}} q^T$$

puisque $\vec{k}_i + \vec{k}_i' = \vec{0}$, et le graphe $\{\bot \to X_i\} \in \mathcal{G}_c$.

2. Pendant la dérivation $u' \xrightarrow{*}_{\hat{\delta}} q^T$, il y a au moins un terme u_k, $C_1[X_i]$, ou $C_2[X_j]$ qui soit annoté par un état de la forme $(p, a, G, \emptyset)$.
Alors
$$u' \xrightarrow{*}_{\hat{\delta}} \big((q, \bot, G, \emptyset), \vec{0}\big) \xrightarrow{\gamma_4}_{\hat{\delta}} q^T$$
c-à-d., $G \in \mathcal{G}_c$. Considérons le cas le moins trivial où C_1 n'est pas le contexte trivial, et où pendant la dérivation ci-dessus, nous avons (les autres cas sont plus simples et se traitent en suivant le même schéma) :
$$C_1[X_i] \xrightarrow{*}_{\hat{\delta}} \big((q_1, a_1, G_1, \emptyset), v_1\big)$$
Alors, puisque C_1 a "$\cdot$" comme racine, la remarque précédente (concernant la forme de $\cdot(U_1, U_2)$) implique qu'il existe un terme nul u_0 tel que $C_1[X_i] = \cdot(u_0, X_i)$. Puisque
$$\cdot(u_0, X_i) \xrightarrow{*}_{\hat{\delta}} \big((q_1, a_1, G_1, \emptyset), v_1\big),$$
et u_0 contient des $\tilde{0}$, les règles $\gamma_1$aiii n'ont pas pû être appliquées à cet endroit, puisque nous ne pouvons pas avoir $\cdot(u_0, X_i) \xrightarrow{*}_{\hat{\delta}} q_t^T$. En effet, ceci voudrait dire que $\cdot(u_0, X_i) \in \wp^*_{\mathcal{R}^{-1}}(t)$ qui est égal à $\zeta^*_{\mathcal{R}^{-1}}(t)$ puisque les membres droits des règles de $\mathcal{R}^{-1}$ ne comprennent pas de "||". Seulement, cet ensemble ne contient pas de $\tilde{0}$ alors que $\cdot(u_0, X_i)$ en contient. Donc, les seules règles qui ont pû être appliquées à cet endroit pour annoter $\cdot(u_0, X_i)$ par $\big((q_1, a_1, G_1, \emptyset), v_1\big)$ sont les règles $\gamma_6 a$. Ces règles impliquent qu'il existe $Z \in Var$ tel que $u_0 \xrightarrow{*}_{\tilde{\delta}} (p_1, -Z)$, et que
$$X_i \xrightarrow{*}_{\tilde{\delta}} \Big(\big((p_2, Z), a_1, G_1, \emptyset\big), v_1\Big).$$
le lemme 5.6.2 implique que $a_1 = \bot$, et G_1 est de la forme $\{\bot \rightarrow Y_k \mid 1 \leq k \leq n\}$, où les Y_k sont tels que $\mathcal{S}$ est une séquence de règles de la forme
$$\Big(Y_k || Z_k \rightarrow t_k \text{ (or } Z_k || Y_k \rightarrow t_k \text{)} \Big)_{1 \leq k \leq n}$$
telle que si $l_k = indice(Y_k)$, nous avons :
– $Z_1 \xrightarrow{*}_{\tilde{\delta}} (p_2, Z)$,
– $v_1 = \sum_{1 \leq s \leq n} \vec{k}_{l_s}$,
– $G_1 = \{\bot \rightarrow Y_k \mid 1 \leq k \leq n\}$,
– $\forall 0 < k < n$, $Z_{k+1} \in \zeta^*_{\mathcal{R}^{-1}}(t_k)$,
– $X_i \in \zeta^*_{\mathcal{R}^{-1}}(t_n)$.
Nous pouvons montrer que
$$\tilde{0} \xrightarrow{*}_{\hat{\delta}} \Big(\big((p_2, Z), X_i, \{X_i \rightarrow Y_k \mid 1 \leq k \leq n\}, \emptyset\big), v_1 + \vec{k}'_i\Big).$$
En effet, les règles $\gamma_1 a$i impliquent que
$$\tilde{0} \rightarrow_{\hat{\delta}} \big((q_{X_i}, X_i, \emptyset, \emptyset), \vec{k}'_i\big).$$

Puisque $X_i \in \zeta^*_{\mathcal{R}^{-1}}(t_n)$, nous avons $q^T_{X_i} \xrightarrow{*}_{\tilde{\delta}} q^T_{t_n}$, et donc, les règles γ_3 impliquent que

$$\big((q_{X_i}, X_i, \emptyset, \emptyset), \vec{k}'_i\big) \rightarrow_{\hat{\delta}} \big((q_{t_n}, X_i, \emptyset, \emptyset), \vec{k}'_i\big).$$

Des règles $\gamma_1 a$ii, nous déduisons que

$$\big((q_{t_n}, X_i, \emptyset, \emptyset), \vec{k}'_i\big) \rightarrow_{\hat{\delta}} \big((q_{Z_n}, X_i, X_i \rightarrow Y_n, \emptyset), \vec{k}'_i + \vec{k}_{l_n}\big).$$

Puisque $\forall k < n$, $Z_{k+1} \in \zeta^*_{\mathcal{R}^{-1}}(t_k)$, nous avons $q^T_{Z_{k+1}} \xrightarrow{*}_{\tilde{\delta}} q^T_{t_k}$, et donc

$$\big((q_{Z_{k+1}}, a, G, \emptyset), v\big) \rightarrow_{\hat{\delta}} \big((q_{t_k}, a, G, \emptyset), v\big).$$

Donc, en appliquant successivement les règles γ_3 et $\gamma_1 a$ii, nous obtenons que

$$\tilde{0} \xrightarrow{*}_{\hat{\delta}} \big((q_{Z_n}, X_i, X_i \rightarrow Y_n, \emptyset), \vec{k}'_i + \vec{k}_{l_n}\big) \xrightarrow{*}_{\hat{\delta}}$$

$$\big(q_{t_1}, X_i, \{X_i \rightarrow Y_k \mid 1 < k \leq n\}, \emptyset), \vec{k}'_i + \sum_{1<s\leq n} \vec{k}_{l_s}\big),$$

et puisque $Z_1 \xrightarrow{*}_{\tilde{\delta}} (p_2, Z)$, les règles $\gamma_1 b$ii impliquent que

$$\big((q_{t_1}, X_i, \{X_i \rightarrow Y_k \mid 1 < k \leq n\}, \emptyset), \vec{k}'_i + \sum_{1<s\leq n} \vec{k}_{l_s}\big) \rightarrow_{\hat{\delta}}$$

$$\Big(\big((p_2, Z), X_i, \{X_i \rightarrow Y_k \mid 1 \leq k \leq n\}, \emptyset\big), \vec{k}'_i + \sum_{1\leq s\leq n} \vec{k}_{l_s}\Big).$$

Donc, nous avons :

$$\tilde{0} \xrightarrow{*}_{\hat{\delta}} \Big(\big((p_2, Z), X_i, \{X_i \rightarrow Y_k \mid 1 \leq k \leq n\}, \emptyset\big), \vec{k}'_i + \sum_{1\leq s\leq n} \vec{k}_{l_s}\Big),$$

c-à-d.,

$$\tilde{0} \xrightarrow{*}_{\hat{\delta}} \Big(\big((p_2, Z), X_i, \{X_i \rightarrow Y_k \mid 1 \leq k \leq n\}, \emptyset\big), \vec{k}'_i + v_1\Big).$$

Par conséquent, d'après les règles γ_6a nous obtenons :

$$C_1[\tilde{0}] \xrightarrow{*}_{\hat{\delta}} \Big(\big(q_1, X_i, \{X_i \rightarrow Y_k \mid 1 \leq k \leq n\}, \emptyset\big), \vec{k}'_i + v_1\Big). \tag{5.17}$$

Nous pouvons faire la même analyse pour $C_2[t]$. Nous pouvons montrer que si

$$C_2[X_j] \xrightarrow{*}_{\hat{\delta}} \big((q_2, a_2, G_2, \emptyset), v_2\big),$$

alors

$$C_2[t] \xrightarrow{*}_{\hat{\delta}} \big((q_2, a_2, G_2 \cup \{\perp \rightarrow X_i\}, \emptyset), v_2 + \vec{k}_i\big). \tag{5.18}$$

Soit $u_{j_k} \xrightarrow{*}_{\hat{\delta}} \big((q_{j_k}, a_{j_k}, G_{j_k}, \emptyset), v_{j_k}\big)$. En utilisant (5.17), (5.18), et les règles (γ_5), nous pouvons montrer par induction structurelle sur C que

$$u \xrightarrow{*}_{\hat{\delta}} \big((q, a, G', \emptyset), v + \vec{k}_i + \vec{k}'_i\big),$$

où $G' = (G \setminus \{\bot \to Y_k \mid 1 \leq k \leq n, \bot \to Y_k \notin (\bigcup G_{j_s} \cup G_2)\}) \cup \{X_i \to Y_k \mid 1 \leq k \leq n\} \cup \{\bot \to X_i\}$. Puisque $v = \vec{0}$, nous obtenons $v + \vec{k_i} + \vec{k'_i} = \vec{0}$. Puisque $G \in \mathcal{G}_c$, il est facile de vérifier que $G' \in \mathcal{G}_c$. Par conséquent, nous obtenons par γ_4 que :

$$u \xrightarrow{*}_{\hat{\delta}} \left((q, a, G', \emptyset), \vec{0}\right) \xrightarrow{*}_{\hat{\delta}} q^T.$$

3. Notons qu'il n'est pas possible d'avoir un sous arbre annoté par un état de la forme $(p, a, G, -Y)$ puisque pendant l'annotation de l'arbre u', nous devons arriver à la racine avec l'état q^T, c-à-d. nous devons nous débarasser de l'étiquette $-Y$. Seulement, ceci ne peut être fait que par les règles $\gamma_6 b$, et ces règles ne peuvent pas être appliquées ici puisque C ne contient pas de nœuds étiquetés avec "$\cdot$".

• Une règle $X \cdot Y \to t$ a été appliquée de manière non locale comme décrit sur la figure 5.5. Ce cas suit le même schéma que le cas précédent. Dans ce cas, nous considérons un C et deux termes u_1 et u_2 tels que $u_1 = C_1[X]$ pour un null-contexte C_1 et $u_2 = C_s[Y]$ pour un seq-contexte C_s tels que $u' = C[\cdot(C_1[X], C_s[Y])]$, et $u = C[\cdot(C_1[\tilde{0}], C_s[t])]$. Nous avons que $u' \xrightarrow{*}_{\hat{\delta}} q^T$, et nous voulons montrer que $u \xrightarrow{*}_{\hat{\delta}} q^T$. Alors, à cause des règles γ_6 et le lemme 5.6.2, nous déduisons que pendant l'annotation de u', la seule possibilité est d'avoir $C_1[X] \xrightarrow{*}_{\hat{\delta}} q_1^T$, et $C_s[Y] \xrightarrow{*}_{\hat{\delta}} q_2$. Maintenant, comme précédemment, nous pouvons montrer en utilisant les règles γ_2, $\gamma_5 e$, $\gamma_5 f$, et γ_6 que $u \xrightarrow{*}_{\hat{\delta}} q^T$ en montrant par induction structurelle sur C_1 que $C_1[\tilde{0}] \xrightarrow{*}_{\hat{\delta}} (q_1, -X)$ (nous avons déjà par le lemme 5.5.1 que $C_s[t] \xrightarrow{*}_{\tilde{\delta}} (q_2, X)$). □

5.7 Conclusion

Nous avons considéré dans ce chapitre les systèmes PRS avec différentes sémantiques induites par des équivalences structurelles différentes entre les termes de processus. Ces équivalences correspondent à la considération de différentes combinaisons des propriétés des opérateurs utilisés (associativité, commutativité, élément neutre).

Lorsque l'opérateur "||" n'est pas considéré comme associatif/commutatif, nous donnons des constructions polynômiales d'automates d'arbres finis qui reconnaissent des représentants des ensembles $Post^*_{\equiv}(\mathcal{L})$ et $Pre^*_{\equiv}(\mathcal{L})$ pour un ensemble régulier de configurations $\mathcal{L}$, où $\equiv$ est une équivalence parmi $\{=, \sim_0, \sim_s\}$. Les constructions données sont valables pour toute la classe PRS et permettent en particulier l'analyse des programmes à parallélisme binaires. Cette classe de programmes est assez générale. Elle permet de considérer la récursivité, le parallélisme, la création dynamique de processus, et la synchronisation.

Si nous considérons l'équivalence $\sim$, nous nous restreignons à la classe des systèmes PAD, qui est plus générale que la classe des systèmes à pile et celle des systèmes PA, et qui permet de représenter la communication entre les procédures. Nous montrons que nos constructions précédentes permettent de calculer en temps polynômial des représentants réguliers de l'ensemble $Post^*(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$, et de l'ensemble $Pre^*(\mathcal{L})$ dans le cas où $\mathcal{L}$ est $\sim_{||}$-compatible. Dans le cas général, nous

montrons que nous pouvons construire un représentant non régulier de cet ensemble des prédécesseurs en utilisant des 0-test automates à compteurs. Nous montrons que ces automates sont fermés par intersection avec les automates d'arbres finis standards, et que leur problème du vide est décidable. Ceci permet de les utiliser pour résoudre les problèmes d'accessibilité dans notre cadre.

Ces constructions permettent d'étendre et d'unifier les techniques existantes d'analyse des systèmes à pile et les systèmes PA [BEM97, FWW97, LS98, EP00]. D'un côté, ces résultats étendent de manière non triviale les constructions de $Post^*_=(\mathcal{L})$ et $Pre^*_=(\mathcal{L})$ pour les systèmes PA données dans [LS98, EP00]. En effet, dans le cas de PA les constructions sont plus simples puisque les règles sont de la forme $X \to t$, et donc ne peuvent s'appliquer qu'aux feuilles. Donc dans ce cas, il suffit de calculer les successeurs des variables X et les coller aux feuilles correspondantes dans le langage initial. La difficulté dans notre cas vient du fait que nous avons, en plus de ces règles, des règles de la forme $X \cdot Y \to t$ (ou $X||Y \to t$) qui peuvent tranformer radicalement la structure des termes du langage initial.

D'un autre côté, dans le cas des systèmes de réécriture préfixe, si les langages de départ ne comprennent pas de "||", les constructions que nous donnons pour des représentants des ensembles $Post^*_{\sim_s}(\mathcal{L})$ et $Pre^*_{\sim_s}(\mathcal{L})$ peuvent être vues comme de nouveaux algorithmes de calcul des ensembles des accessibles pour les systèmes de réécriture préfixe (ou les systèmes à pile). Ces algorithmes viennent s'ajouter aux techniques déjà existantes [Cau92, BEM97, FWW97, EHRS00].

Chapitre 6

Calcul de tous les accessibles pour PRS

L'approche adoptée dans le chapitre précédent permet de résoudre le problème d'accessibilité sans considérer les équivalences structurelles du "||" pour toute la classe PRS. Les constructions données sont polynômiales. Si nous considérons toutes les équivalences, l'approche précédente permet l'analyse des systèmes PAD. La plupart des algorithmes proposés sont polynômiaux, mais ne permettent pas de caractériser *tout* l'ensemble des accessibles. Dans ce chapitre nous nous attaquons au problème de construire *tout* l'ensemble des accessibles (pas juste des représentants de cet ensemble), dans le cas où toutes les équivalences structurelles entre les termes sont considérées, pour des classes de PRS plus générales que les systèmes PAD.

Comme un PRS en forme normale peut être vu comme l'union d'un système de réécriture préfixe (ou système à pile) $\mathcal{R}_s$ et d'un système de réécriture de multiensembles (ou réseau de Petri) $\mathcal{R}_p$, notre but dans ce chapitre est d'étendre et d'intégrer les techniques d'analyse d'accessibilité des automates à pile et des réseaux de Petri pour avoir une procédure générale pour les systèmes PRS. Nous proposons une procédure qui calcule *tout* l'ensemble des accessibles, et qui est *paramétrée* par un algorithme Ω qui calcule cet ensemble pour les systèmes de réécriture de multiensembles (réseaux de Petri) en utilisant les ensembles semilinéaires (ou de manière équivalente, l'arithmétique de Presburger) pour la représentation et la manipulation d'ensembles de paral-termes (ou marquages pour les réseaux de Petri). Comme les réseaux de Petri ne préservent pas la semilinéarité (les ensembles des marquages accessibles des réseaux de Petri ne sont en général pas semilinéaires), cet algorithme peut être soit *exact* (dans le cas où Ω est instancié par un algorithme qui calcule *exactement* un ensemble semilinéaire représentant les marquages accessibles), mais applicable pour des classes particulières de systèmes de réécriture de multiensembles (précisément celles qui préservent effectivement la semilinéarité), ou *approximatif* (dans le cas où Ω est instancié par un algorithme qui calcule une sur-approximation semilinéaire de l'ensemble des marquages accessibles), mais applicable pour tout système de réécriture de multiensembles.

Notre but est de donner un cadre général générique d'analyse des PRS où toutes les techniques d'analyse symbolique pour les réseaux de Petri, qui sont basées sur l'arithmétique linéaire et les ensembles semilinéaires peuvent être intégrées, en combinaison avec des techniques d'analyse des systèmes à pile.

Pour ce faire, au lieu de représenter les termes de processus par des arbres binaires comme dans le chapitre précédent, nous proposons de les représenter par des arbres ayant des largeurs arbitraires. En effet, un seq-terme $X_1 \cdot X_2 \cdots X_n$ et un paral-terme $X_1 \| X_2 \cdots \| X_n$ peuvent être représentés par des arbres dont la racine est "$\cdot$" ou "$||$" et dont les feuilles sont étiquetées par $X_1, X_2, \ldots, X_n$ (voir la figure 6). Plus précisément, le terme $\cdot(X_1, \ldots, X_n)$ représente $X_1 \cdot X_2 \cdots X_n$, et puisque "$||$" est commutatif, $X_1 \| X_2 \cdots \| X_n$ est représenté par l'ensemble de termes $||(X_{\sigma(1)}, \ldots, X_{\sigma(n)})$, où σ est une permutation de l'ensemble $\{1, \ldots, n\}$. Nous introduisons alors une classe d'automates d'arbres (les *CH-automates*) qui permettent de représenter de manière *finie* un ensemble infini de termes de processus représentés par des arbres à largeur non bornée. Ces automates combinent les langages réguliers de mots (pour représenter les ensembles de seq-termes) avec les ensembles semilinéaires, ou l'arithmétique de Presburger (pour représenter les ensembles de paral-termes) afin de représenter des ensembles de termes PRS.

Dans la première section de ce chapitre, nous définissons cette classe d'automates et nous montrons qu'elle satisfait toutes les bonnes propriétés de fermeture par les opérations booléennes et de décidabilité du problème du vide. Ensuite, nous donnons notre procédure générique qui prend en entrée (1) un algorithme Ω qui calcule un ensemble semilinéaire représentant $M^*(L)$ pour tout système de réécriture de multiensembles (ou réseau de Petri) M appartenant à une classe $\mathcal{C}$; et tout ensemble semilinéaire de marquages (ou de paral-termes) L, (2) un PRS $\mathcal{R}$ dont la partie "réécriture de multiensembles" $\mathcal{R}_p$ appartient à la classe $\mathcal{C}$, et (3) un CH-automate reconnaissant un ensemble $\mathcal{L}$ de termes de processus. L'algorithme nous calcule un CH-automate qui reconnaît $Post^*(\mathcal{L})$.

Nous montrons que notre algorithme peut être appliqué pour calculer *exactement* les ensembles des accessibles dans le cas des systèmes PAD, ce qui permet de retrouver les résultats du chapitre précédent et d'étendre tous les travaux existants sur l'analyse d'accessibilité des systèmes à pile et des systèmes PA. De plus, dans ce cas, nous montrons que notre procédure permet de décider la satisfaisabilité des formules EF pour les systèmes PAD.

Nous montrons enfin qu'en instanciant Ω par n'importe quelle procédure d'analyse d'accessibilité (exacte ou approchée) des réseaux de Petri ou des automates à compteurs qui manipule les semilinéaires, notre technique permet de calculer des surapproximations des ensembles dees accessibles pour toute la classe PRS.

Le contenu de ce chapitre fait l'objet du rapport de recherche [BT03a].

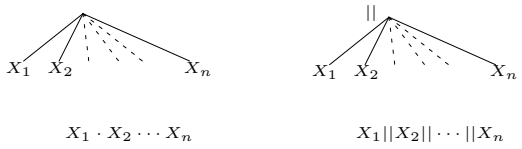


FIG. 6.1 – Représentation des termes de processus par des arbres à arité arbitraire

6.1 Représentation symbolique des ensembles de termes de processus

Pour pouvoir reconnaître des ensembles infinis de termes de processus qui sont fermés par associativité/commutativité des opérateurs séquentiel et parallèle, nous introduisons dans cette partie une classe d'automates d'arbres qui reconnaissent des ensembles d'arbres modulo associativité / associativité-commutativité de certains opérateurs. Ces automates appelés CH-automates (pour Commutative-Hedge-automates) sont une extension à la fois des automates d'arbres ascendants qui reconnaissent des arbres à arités fixes (arbres avec largeurs bornées) (voir la définition 2.1.20), et des Hedge automates qui reconnaissent des arbres ayant des largeurs non bornées en utilisant des contraintes régulières [BKMW01]. Les automates que nous introduisons reconnaissent des langages d'arbres sur des alphabets qui contiennent des symboles avec arités fixes et d'autres avec arités arbitraires et qui sont fermés par commutativité et/ou associativité de certains opérateurs. Les contraintes des règles peuvent être soit (1) régulières : nous affectons un état à un nœud étiqueté par un opérateur associatif si la séquence des états qui annotent ses fils appartient à un certain langage régulier, ou (2) semilinéaires : nous affectons un état à un nœud étiqueté par un opérateur associatif/commutatif si l'image de Parikh de la séquence des états qui annotent ses fils satisfait une certaine formule de Presburger.

Plus précisément, considérons des automates d'arbres "classiques" tels que définis dans la définition 2.1.20 qui reconnaissent des arbres à largeur fixe. Prenons par exemple le cas des arbres binaires. Pour reconnaître un arbre, l'automate doit trouver une annotation des nœuds par des états telle que (1) la racine de l'arbre soit annotée par un état final, et (2) qui soit compatible avec les règles de l'automate de la forme :

- soit $a \to q$ pour annoter les feuilles ;
- soit $f(q_1, q_2) \to q$ qui permet d'annoter un terme $f(t_1, t_2)$ avec l'état q si t_1 et t_2 sont respectivement annotés par q_1 et q_2.

Supposons maintenant que f représente un opérateur associatif. Nous considérons alors qu'un nœud correspondant à cet opérateur peut avoir un nombre arbitraire de fils. Par conséquent, nous utilisons des règles d'annotation de la forme $f(L) \to q$, où L est un langage régulier sur l'alphabet des états de contrôle de l'automate. Ces règles permettent d'annoter un arbre $f(t_1, \ldots, t_n)$ avec q si chaque sous arbre t_i est annoté par un état q_i tels que la séquence $q_1 \cdots q_n$ est dans L. Si nous supposons en plus que f est associatif *et* commutatif, l'ordre entre les fils n'est plus important. Par conséquent, nous utilisons des règles de la forme $f(\varphi) \to q$, où φ est une contrainte arithmétique. Ces règles permettent d'annoter la racine de l'arbre $f(t_1, \ldots, t_n)$ avec q si chaque sous

arbre t_i est annoté par un état q_i, et le nombre d'occurrences de chaque état dans la séquence $q_1 \cdots q_n$ satisfait la contrainte φ.

Dans ce qui suit, nous donnons d'abord la définition formelle de cette classe d'automates. Nous montrons ensuite qu'elle est effectivement fermée par toutes les opérations booléennes, et que son problème du vide est décidable. Ceci permet de l'utiliser comme représentation symbolique pour résoudre nos problèmes d'accessibilité. Nous terminons cette partie par la description d'une classe particulière de ces automates qui permet de représenter les ensembles de termes de processus.

6.1.1 Commutative Hedge Automates

Soit $\Sigma = \Sigma' \cup \Sigma_A$ un alphabet fini, où Σ' est un alphabet muni d'une fonction d'arité, et Σ_A est un ensemble fini d'opérateurs associatifs tels que Σ' et Σ_A sont disjoints. Pour $k \geq 0$, Σ_k est l'ensemble des éléments de Σ' d'arité k. Comme les symboles de Σ_A sont associatifs, ils peuvent avoir une arité arbitraire. Soit $\mathcal{X}$ un ensemble dénombrable de variables $\{x_1, x_2, \ldots\}$. Comme dans le cas des termes à arités fixes, nous définissons dans ce qui suit l'ensemble $T_\Sigma[\mathcal{X}]$ des termes sur Σ et $\mathcal{X}$, et la fonction *racine* qui détermine la racine de l'arbre représentant un terme.

Définition 6.1.1 *L'ensemble $T_\Sigma[\mathcal{X}]$ de termes sur Σ et $\mathcal{X}$ est le plus petit ensemble qui satisfait :*

- *Si $f \in \Sigma_0 \cup \mathcal{X}$, alors $f \in T_\Sigma[\mathcal{X}]$ et $racine(f) = f$.*
- *Si $k \geq 1$, $f \in \Sigma_k$, et $t_1, \ldots, t_k \in T_\Sigma[\mathcal{X}]$, alors $f(t_1, \ldots, t_k)$ est dans $T_\Sigma[\mathcal{X}]$ et $racine\big(f(t_1, \ldots, t_k)\big) = f$.*
- *Si $f \in \Sigma_A$, $t_1, \ldots, t_k \in T_\Sigma[\mathcal{X}]$ pour un certain $k \geq 1$, et $racine(t_i) \neq f$ pour chaque $1 \leq i \leq k$, alors $f(t_1, \ldots, t_k)$ est dans $T_\Sigma[\mathcal{X}]$, et $racine\big(f(t_1, \ldots, t_k)\big) = f$.*

Observons que si $f \in \Sigma_A$, alors nous ne considérons que les termes de la forme $f(t_1, \ldots, t_k)$ tels que pour chaque i, la racine de t_i est différente de f. En effet, puisque f est associatif, $f\big(t_1, \ldots, t_{i-1}, f(u_1, \ldots, u_m), t_{i+1}, \ldots, t_n\big)$ est équivalent au terme $f(t_1, \ldots, t_{i-1}, u_1, \ldots, u_m, t_{i+1}, \ldots, t_n)$.

Nous dénotons par T_Σ l'ensemble $T_\Sigma[\emptyset]$. Les termes de T_Σ sont appelés *termes clos*. Un terme t dans $T_\Sigma[\mathcal{X}]$ est *linéaire* si chaque variable apparaît au plus une fois dans t. Un *contexte* C est un terme linéaire de $T_\Sigma[\mathcal{X}]$. Soient $t_1, \ldots, t_n$ des termes de T_Σ, alors $C[t_1, \ldots, t_n]$ dénote le terme obtenu en remplaçant dans le contexte C l'occurrence de la variable x_i par le terme t_i, pour chaque $1 \leq i \leq n$. Nous notons parfois C par $C[x_1, \ldots, x_n]$ pour exprimer que c'est un contexte à n variables.

Σ_A peut contenir des opérateurs commutatifs. Nous considérons alors que $\Sigma_A = \Sigma'_A \cup \Sigma'_{AC}$ où Σ'_{AC} est un ensemble d'opérateurs associatifs et commutatifs. Nous supposons que Σ'_A et Σ'_{AC} sont disjoints.

Définition 6.1.2 *Un CH-automate est un quadruplet $\mathcal{A} = (Q, \Sigma, F, \Delta)$ où :*

- *Q est l'union d'ensembles finis disjoints d'états $Q' \cup \bigcup_{f \in \Sigma_A} Q_f$,*
- *$F \subseteq Q$ est un ensemble d'états finaux,*
- *Δ est un ensemble de règles de la forme :*

1. $a \to q$, *où* $q \in Q', a \in \Sigma_0$; *ou*
2. $f(q_1, \ldots, q_n) \to q$, *où* $f \in \Sigma_n, q \in Q'$, *et* $q_i \in Q$; *ou*
3. $q \to q'$, *où* $(q, q') \in Q' \times Q' \cup \bigcup_{f \in \Sigma_A} Q_f \times Q_f$; *ou*
4. $f(L) \to q$, *où* $f \in \Sigma'_A$, $L \subseteq (Q \setminus Q_f)^*$, *et* $q \in Q_f$; *ou*
5. $f(\varphi) \to q$, *où* $f \in \Sigma'_{AC}$, $q \in Q_f$, *et* φ *est une formule de Presburger telle que* $FV(\varphi) = \{x_q \mid q \in Q \setminus Q_f\}$.

Un CH-automate induit une relation $\to_\Delta$ entre les termes, définie comme suit : $t \to_\Delta t'$ ssi il existe un contexte C tel que $t = C[s]$, $t' = C[s']$, et
- il existe $a \to q$ dans Δ tel que $s = a$ et $s' = q$; ou
- il existe $q \to q'$ dans Δ tel que $s = q$ et $s' = q'$; ou
- il existe $f(q_1, \ldots, q_n) \to q$ dans Δ tel que $s = f(q_1, \ldots, q_n)$ et $s' = q$; ou
- il existe $f(L) \to q$ dans Δ, où $f \in \Sigma'_A$ tel que $s = f(q_1, \ldots, q_k)$, $q_1 \cdots q_k \in L$, et $s' = q$; ou
- il existe $f(\varphi) \to q$ dans Δ, où $f \in \Sigma'_{AC}$ tel que $s = f(q_1, \ldots, q_k)$, $Parikh(q_1 \cdots q_k) \models \varphi$, et $s' = q$.

Soit $\xrightarrow{*}_\Delta$ la clôture réflexive-transitive de $\to_\Delta$. Un état q est accessible s'il existe au moins un terme clos t tel que $t \xrightarrow{*}_\Delta q$. Un terme clos t est accepté par un état q si $t \xrightarrow{*}_\Delta q$. Soit $L_q = \{t \mid t \xrightarrow{*}_\Delta q\}$. Un terme clos t est accepté par un CH-automate $\mathcal{A} = (Q, \Sigma, F, \Delta)$ s'il existe un état q dans F tel que $t \xrightarrow{*}_\mathcal{A} q$. Le CH-langage de $\mathcal{A}$, dénoté par $\mathcal{L}(\mathcal{A})$, est l'ensemble de tous les termes clos acceptés par $\mathcal{A}$.

Un CH-automate $\mathcal{A}$ est déterministe (DCH-automate) ssi pour chaque terme clos t, il existe au plus un état q tel que $t \xrightarrow{*}_\Delta q$. L'automate $\mathcal{A}$ est complet si pour chaque terme clos t, il existe un état q tel que $t \xrightarrow{*}_\Delta q$.

6.1.2 Propriétés des CH-automates

Proposition 6.1.1 *Etant donnés un terme clos t et un CH-automate $\mathcal{A}$, nous pouvons décider si t est accepté par $\mathcal{A}$.*

Preuve : Soit un terme t, le CH-automate commence aux feuilles. Il annote chaque feuille a par l'ensemble des états $\{q \mid a \to q\}$, et remonte en annotant à chaque fois chaque sous terme par l'ensemble de tous les états possibles par lesquels ce terme peut être annoté. Notons que ceci est possible puisque l'arithmétique de Presburger est décidable. Le terme est alors accepté ssi l'automate arrive à la racine de t avec un ensemble d'états contenant un état final. □

6.1.2.1 Déterminiser les CH-automates

Soit $\mathcal{A} = (Q, \Sigma, F, \Delta)$ un CH-automate. Nous définissons le DCH-automate suivant $\mathcal{A}_D = (Q_D, \Sigma, F_D, \Delta_D)$ tel que :
- $Q_D = Q'_D \cup \bigcup\limits_{f \in \Sigma_A} Q_{f_D}$, où $Q'_D = 2^{Q'}$, $Q_{f_D} = 2^{Q_f}$,
- $F_D = \{s \in Q_D \mid s \cap F \neq \emptyset\}$, et

– Δ_D contient les règles suivantes :
  (b_1) $a \to s, a \in \Sigma_0$ ssi $s = \{q \in Q' \mid a \xrightarrow{*}_{\Delta} q\}$,
  (b_2) $f(s_1, \ldots, s_n) \to s, f \in \Sigma_n$ ssi

$$s = \{q \in Q' \mid \exists q_1 \in s_1, \ldots, q_n \in s_n, f(q_1, \ldots, q_n) \xrightarrow{*}_{\Delta} q\}$$

  (b_3) $f(L) \to s, f \in \Sigma'_A$, et $s \in 2^{Q_f}$ ssi

$$L = \Big(\bigcap_{q \in s} \bigcup_{\substack{f(L_i) \to q' \in \Delta \\ q' \xrightarrow{*}_{\Delta} q}} \theta(L_i)\Big) \setminus \Big(\bigcup_{q \notin s} \bigcup_{\substack{f(L_i) \to q' \in \Delta \\ q' \xrightarrow{*}_{\Delta} q}} \theta(L_i)\Big)$$

  (b_4) $f(\varphi) \to s, f \in \Sigma'_{AC}$, et $s \in 2^{Q_f}$ ssi

$$\varphi = \Big(\bigwedge_{q \in s} \bigvee_{\substack{f(\varphi_i) \to q' \in \Delta \\ q' \xrightarrow{*}_{\Delta} q}} \eta(\varphi_i)\Big) \wedge \Big(\bigwedge_{q \notin s} \bigwedge_{\substack{f(\varphi_i) \to q' \in \Delta \\ q' \xrightarrow{*}_{\Delta} q}} \neg\eta(\varphi_i)\Big)$$

où θ est une substitution telle que pour chaque $q \in Q$,

$$\theta(q) = \{s \in Q_D \mid q \in s\}$$

et η est une substitution telle que pour chaque $q \in Q$,

$$\eta(x_q) = \sum_{s \in Q_D \mid q \in s} x_s$$

où x_q est la variable associée à l'état q.

Ce processus de déterminisation peut être vu comme une adaptation de la "subset construction" standard. L'automate mémorise dans ses états tous les états où il pourrait être après avoir lu un certain terme. Ceci permet de regrouper toutes les règles ayant le même côté gauche en en seule règle.

Lemme 6.1.1 *$\mathcal{A}_D$ est un automate déterministe qui reconnaît le même langage que $\mathcal{A}$.*

Preuve : Nous montrons que

$$t \xrightarrow{*}_{\Delta_D} s \text{ ssi } s = \{q \in Q \mid t \xrightarrow{*}_{\Delta} q\}.$$

Nous procédons par induction structurelle sur t :
– $t = a$, pour un $a \in \Sigma_0$, alors la propriété se déduit directement des règles (b_1),
– $t = f(t_1, \ldots, t_n)$, pour un $f \in \Sigma_n$. D'abord, supposons que

$$t = f(t_1, \ldots, t_n) \xrightarrow{*}_{\Delta_D} f(s_1, \ldots, s_n) \to_{\Delta_D} s.$$

Il s'en suit par induction que $s_i = \{q \in Q \mid t_i \xrightarrow{*}_{\Delta} q\}$ Donc les règles (b_2) impliquent que $s = \{q \in Q \mid t \xrightarrow{*}_{\Delta} q\}$. Pour l'autre direction, soient $s = \{q \in Q \mid t \xrightarrow{*}_{\Delta} q\}$, et $s_i = \{q \in Q \mid t_i \xrightarrow{*}_{\Delta} q\}$. Nous avons par induction que pour chaque i, $t_i \xrightarrow{*}_{\Delta_D} s_i$, et par la construction des règles (b_2), nous déduisons que $f(s_1, \ldots, s_n) \to s$ est une règle de Δ_D, et par conséquent, que $t \xrightarrow{*}_{\Delta_D} s$.

– $t = f(t_1, \ldots, t_n)$, pour un $f \in \Sigma'_A$. Supposons que

$$t = f(t_1, \ldots, t_n) \xrightarrow{*}_{\Delta_D} f(s_1, \ldots, s_n) \rightarrow_{\Delta_D} s.$$

Soit alors L tel que $s_1 \cdots s_n \in L$, et $f(L) \rightarrow s$ une règle de Δ_D. Les règles (b_3) impliquent que

$$L = \Big(\bigcap_{q \in s} \bigcup_{\substack{f(L_i) \rightarrow q' \in \Delta \\ q' \xrightarrow{*}_{\Delta} q}} \theta(L_i) \Big) \setminus \Big(\bigcup_{q \notin s} \bigcup_{\substack{f(L_i) \rightarrow q' \in \Delta \\ q' \xrightarrow{*}_{\Delta} q}} \theta(L_i) \Big)$$

Puisque $s_1 \cdots s_n \in L$, ceci veut dire que pour chaque $q \in s$, il existe un langage régulier L_j, et une dérivation $f(L_j) \xrightarrow{*}_{\Delta} q$ tels que $s_1 \cdots s_n \in \theta(L_j)$, et pour chaque état $q \notin s$ il n'y a pas de langage régulier L'_j tel que $s_1 \cdots s_n \in \theta(L'_j)$, et $f(L'_j) \xrightarrow{*}_{\Delta} q$ est une dérivation de Δ. Par la définition de θ, nous déduisons qu'il existe n états $q_i \in s_i$ tels que $q_1 \cdots q_n \in L_j$, que $f(q_1, \ldots, q_n) \xrightarrow{*}_{\Delta} q$, et que pour tous les états $p_i \in s_i$, et pour chaque état $q \notin s$, il n'y a pas de langage L'_j tel que $p_1 \cdots p_n \in L'_j$, et $f(L'_j) \xrightarrow{*}_{\Delta} q$ est une dérivation de Δ. De l'autre côté, puisque $t_i \xrightarrow{*}_{\Delta_D} s_i$, nous obtenons par induction que $s_i = \{q \in Q \mid t_i \xrightarrow{*}_{\Delta} q\}$. Par conséquent, nous avons que $t \xrightarrow{*}_{\Delta} f(q_1, \ldots, q_n) \xrightarrow{*}_{\Delta} q$ pour chaque état q dans s, et qu'il n'y a pas d'état $q \notin s$ tels que $t \xrightarrow{*}_{\Delta} q$, ce qui veut dire que s est exactement l'ensemble $\{q \in Q \mid t \xrightarrow{*}_{\Delta} q\}$.
Pour l'autre direction, soit $s = \{q \in Q \mid t \xrightarrow{*}_{\Delta} q\}$. Nous montrons que $t \xrightarrow{*}_{\Delta_D} s$. Soit $s_i = \{q \in Q \mid t_i \xrightarrow{*}_{\Delta} q\}$. Par induction, nous obtenons que $t_i \xrightarrow{*}_{\Delta_D} s_i$. Et par la définition de s nous déduisons que pour chaque $q \in s$, il existe $q_i \in s_i$ tel que $f(q_1, \ldots, q_n) \xrightarrow{*}_{\Delta} q$, c-à-d., tel que $q_1 \cdots q_n \in L$ et $f(L) \xrightarrow{*}_{\Delta} q$ pour un langage régulier L. Ce qui veut dire que pour chaque $q \in s$, il existe un langage régulier L tel que $f(L) \xrightarrow{*}_{\Delta} q$, et $s_1 \cdots s_n \in \theta(L)$. La définition des règles (b_3) implique que $t \xrightarrow{*}_{\Delta_D} f(s_1, \ldots, s_n) \xrightarrow{*}_{\Delta_D} s$
– $t = f(t_1, \ldots, t_n)$, pour un $f \in \Sigma'_{AC}$. Ce cas est similaire au cas précédent.

□

Nous obtenons alors le théorème suivant :

Théorème 6.1.1 *Soit L un ensemble de termes acceptés par un CH-automate. Alors, il existe un CH-automate déterministe qui accepte L.*

De même, nous montrons qu'un CH-automate peut être complété :

Théorème 6.1.2 *Soit $\mathcal{A}$ un CH-automate. Alors il existe un CH-automate complet équivalent à $\mathcal{A}$.*

Preuve : Nous construisons un CH-automate complet $\mathcal{A}_c = (Q_c, \Sigma, F_c, \Delta_c)$ qui reconnaît le même langage que $\mathcal{A}$ en rajoutant les nouveaux états puits q, q_f pour chaque $f \in \Sigma_A$ tels que $Q'_c = Q' \cup \{q\}$, $(Q_c)_f = Q_f \cup \{q_f\}$, pour chaque $f \in \Sigma_A$, $F_c = F$, et Δ_c contient Δ et les règles suivantes :

- $a \to q$, s'il n'y a pas de règles dans Δ ayant a comme côté gauche,
- $f(q_1, \ldots, q_n) \to q, q_1, \ldots, q_n \in Q_c$, s'il n'y a pas de règles dans Δ ayant $f(q_1, \ldots, q_n)$ comme côté gauche,
- Pour chaque $f \in \Sigma'_A$, soient $L_1, \ldots, L_k$ toutes les contraintes telles qu'il existe un état q tel que $f(L_j) \to q$ est une règle de Δ. Alors Δ_c contient la règle

$$f(\overline{L_1 \cup \cdots \cup L_k}) \to q_f$$

où $\overline{L} = (Q_c \setminus (Q_c)_f)^* \setminus L$.
- Pour chaque $f \in \Sigma'_{AC}$, soit $\varphi_1, \ldots, \varphi_k$ toutes les contraintes telles qu'il existe un état q où $f(\varphi_j) \to q$ est une règle de Δ. Alors si la contrainte $\bigvee_j \varphi_j$ n'est pas équivalente à $vrai$, Δ_c contient la règle

$$f\big(\neg(\varphi_1 \vee \cdots \vee \varphi_k) \wedge (x_q = 0, q \in (Q_c)_f)\big) \to q_f$$

Nous expliquons dans ce qui suit l'intuition exprimée par ces règles. Par exemple, pour $f \in \Sigma'_A$, la règle

$$f(\overline{L_1 \cup \cdots \cup L_k}) \to q_f$$

assure que pour chaque terme $t = f(t_1, \ldots, t_n)$ tel que

$$t = f(t_1, \ldots, t_n) \xrightarrow{*}_\Delta f(q_1, \ldots, q_n),$$

il y a deux cas :

1. soit il existe une règle $f(L) \to q$ dans Δ telle que $q_1 \cdots q_n \in L$, auquel cas, t est annoté par q,
2. si une telle règle n'existe pas, t est annoté par q_f.

De la même manière, pour $f \in \Sigma'_{AC}$, la règle $f(\neg(\varphi_1 \vee \cdots \vee \varphi_k) \to q_f$ assure que pour chaque terme $t = f(t_1, \ldots, t_n)$ tel que

$$t = f(t_1, \ldots, t_n) \xrightarrow{*}_\Delta f(q_1, \ldots, q_n),$$

il y a deux cas :

1. soit il existe une règle $f(\varphi) \to q$ dans Δ telle que $Parikh(q_1 \cdots q_n) \models \varphi$, auquel cas, t est annoté par q,
2. si une telle règle n'existe pas, t est annoté par q_f.

Notons que si $\mathcal{A}$ est déterministe, alors $\mathcal{A}_c$ reste déterministe. □

Nous montrons que la classe des CH-langages est effectivement fermée par les opérations booléennes.

Théorème 6.1.3 *La classe des CH-langages est effectivement fermée par union, intersection, et complémentation.*

Preuve :

Union : Soient L_1 et L_2 deux CH-langages reconnus respectivement par les CH-automates $\mathcal{A}_1 = (Q_1, \Sigma, F_1, \Delta_1)$, et $\mathcal{A}_2 = (Q_2, \Sigma, F_2, \Delta_2)$. Alors $L_1 \cup L_2$ est reconnu par le CH-automate $\mathcal{A} = (Q, \Sigma, F_1 \cup F_2, \Delta_1 \cup \Delta_2)$, tel que $Q' = Q'_1 \cup Q'_2$, et $Q'_f = Q_{1_f} \cup Q_{2_f}$.

Complémentation : Soit L un CH-language, et soit $\mathcal{A} = (Q, \Sigma, F, \Delta)$ un CH-automate complet et déterministe qui reconnaît L. Alors, il est facile de voir que $\mathcal{A}' = (Q, \Sigma, \bar{F}, \Delta)$ reconnaît le complément de L ($\bar{F}$ dénotes le complément de F, c-à-d., les états non finaux de $\mathcal{A}$) .

Intersection : Puisque les CH-langages sont fermés par union et complémentation, et puisque

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}},$$

nous déduisons que CH-languages sont clos par intersection.

Nous donnons aussi une construction directe de l'intersection de deux CH-langages comme suit :

Soient L_1 et L_2 deux CH-langages reconnus respectivement par les CH-automates $\mathcal{A}_1 = (Q_1, \Sigma, F_1, \Delta_1)$, et $\mathcal{A}_2 = (Q_2, \Sigma, F_2, \Delta_2)$. Alors $L_1 \cap L_2$ est reconnu par le CH-automate $\mathcal{A} = (\tilde{Q}, \Sigma, F_1 \times F_2, \Delta)$, tel que $\tilde{Q}^0 = Q_1^0 \times Q_2^0$, et $\tilde{Q}_f = Q_{1_f} \times Q_{2_f}$, et Δ est défini comme suit :

- $f\big((q_1, q'_1), \ldots, (q_n, q'_n)\big) \to (q, q') \in \Delta$ si $f \in \Sigma_n$, $f(q_1, \ldots, q_n) \to q \in \Delta_1$, et $f(q'_1, \ldots, q'_n) \to q' \in \Delta_2$,
- Soit α_1 la substitution suivante définie sur Q_1 :

  $$\alpha_1(q) = \{(q, q') \mid q' \in Q_2\},$$

  et soit α_2 la substitution suivante définie sur Q_2 :

  $$\alpha_2(q') = \{(q, q') \mid q \in Q_1\}.$$

  Alors, si $f \in \Sigma_A$, $f(L_1) \to q_1 \in \Delta_1$, et $f(L_2) \to q_2 \in \Delta_2$, Δ contient la règle

  $$f\big(\alpha_1(L_1) \cap \alpha_2(L_2)\big) \to (q_1, q_2)$$

- Soit α'_1 la substitution de variables suivante définie sur Q_1 :

  $$\alpha'_1(x_q) = \sum_{q' \in Q_2} x_{(q,q')}$$

  et soit α'_2 la substitution de variables suivante définie sur Q_2 :

  $$\alpha'_2(x_{q'}) = \sum_{q \in Q_1} x_{(q,q')}$$

  Alors, si $f \in \Sigma_{AC}$, $f(\varphi_1) \to q_1 \in \Delta_1$, et $f(\varphi_2) \to q_2 \in \Delta_2$, Δ contient la règle

  $$f\big(\alpha'_1(\varphi_1) \wedge \alpha'_2(\varphi_2)\big) \to (q_1, q_2)$$

□

Finalement, nous montrons que le problème du vide est décidable pour les CH-automates :

Théorème 6.1.4 *Le problème du vide des CH-langages est décidable.*

Preuve : Soit $\mathcal{A} = (Q, \Sigma, F, \Delta)$ un CH-automate. Le langage accepté par $\mathcal{A}$ est vide ssi l'ensemble des états accessibles de $\mathcal{A}$ ne contient pas d'état final. Cet ensemble est le plus petit ensemble qui contient $Acc = \{q \mid \exists a \in \Sigma_0, a \to q \in \Delta\}$, et qui est fermé par les règles d'inférence suivantes :

- si $q \in Acc$ et $q \to q' \in \Delta$, alors $q' \in Acc$,
- si $q_1, \ldots, q_n \in Acc$ et il existe une règle $f(q_1, \ldots, q_n) \to q$ dans Δ pour un $f \in \Sigma_n$, alors $q \in Acc$,
- si $q_1, \ldots, q_n \in Acc$ et il existe une règle $f(L) \to q$ dans Δ (pour un certain $f \in \Sigma'_A$) tel que $(q_1 + \cdots + q_n)^* \cap L \neq \emptyset$, alors $q \in Acc$,
- si $q_1, \ldots, q_n \in Acc$ et il existe une règle $f(\varphi) \to q$ dans Δ (pour un $f \in \Sigma'_{AC}$) tel que la formule
$$(\bigwedge_{q \notin \{q_1, \ldots, q_n\}} x_q = 0) \wedge \varphi$$
est satisfaisable, alors $q \in Acc$.

Par exemple, la dernière règle implique que si $q_1, \ldots, q_n$ sont accessibles, et s'il existe une règle $f(\varphi) \to q$ dans Δ telle qu'il existe des entiers $N_1, \ldots, N_n$ tels que le vecteur ayant N_1 à la composante correspondant à x_{q_1},..., N_n à la composante correspondant à x_{q_n}, et 0 partout ailleurs satisfait φ, alors q est aussi accessible. Pour voir ceci, il suffit de considérer n termes $t_1, \ldots, t_n$ tels que $t_i \xrightarrow{*}_\Delta q_i$ (ces termes existent puisque les q_i sont accessibles). Alors le terme suivant est accepté par q :

$$t = f(\underbrace{t_1, \ldots, t_1}_{N_1 \text{ fois}}, \ldots, \underbrace{t_n, \ldots, t_n}_{N_n \text{ fois}}) \xrightarrow{*}_\Delta f(\underbrace{q_1, \ldots, q_1}_{N_1 \text{ fois}}, \ldots, \underbrace{q_n, \ldots, q_n}_{N_n \text{ fois}}) \xrightarrow{*}_\Delta q$$

puisque

$$Parikh(\underbrace{q_1 \cdots q_1}_{N_1 \text{ fois}} \cdots \underbrace{q_n \cdots q_n}_{N_n \text{ fois}}) \models \varphi.$$

□

6.1.3 CH-automates pour les termes de processus

Nous représentons les ensembles de termes de processus par des CH-automates. En effet, nous représentons un terme de la forme $t_1 \cdots t_n$ par $\cdot(t_1, \ldots, t_n)$, et un terme de la forme $t_1 || \cdots || t_n$ par $||(t_1, \ldots, t_n)$. Nous gardons cependant la notation infixée pour représenter les termes des membres gauches et droits des règles PRS. De ce fait, l'ensemble $\mathcal{T}$ des termes de processus peut être vu comme l'ensemble des termes de T_Σ

où $\Sigma_0 = \{0\} \cup Var$, $\Sigma'_A = \{.\}$, et $\Sigma'_{AC} = \{||\}$. Plus précisément, comme 0 est neutre par rapport à "$\cdot$" et "||", nous représentons les termes de processus par des *arbres bien formés*, c-à-d., des arbres correspondant à des termes de T_Σ qui ont la forme suivante : 0, $X \in Var$, ou $\cdot(t_1, \ldots, t_n)$ (resp. $||(t_1, \ldots, t_n)$), où les t_i sont des termes bien formés qui sont soit des variables (dans Var), soit des termes ayant "||" comme racine (resp. ayant "$\cdot$" comme racine).

Les termes bien formés peuvent être reconnus par des CH-automates de la forme $\mathcal{A} = (Q, \Sigma, F, \Delta)$, où Q est l'union disjointe $Q = Q' \cup Q_. \cup Q_{||}$ tels que Q' est lui-même l'union disjointe $Q' = Q_0 \cup Q_-$, et Δ a des règles de la forme :

$$\begin{array}{rcl}
X & \rightarrow & q, \text{ où } q \in Q_-, X \in Var; \\
0 & \rightarrow & q, \text{ où } q \in Q_0; \\
q & \rightarrow & q', \text{ où } (q, q') \in Q_0 \times Q_0 \cup Q_- \times Q_- \cup Q_. \times Q_. \cup Q_{||} \times Q_{||}; \\
\cdot(L) & \rightarrow & q, \text{ où } L \subseteq \big(Q \setminus (Q_. \cup Q_0)\big)^*, q \in Q_.; \\
||(\varphi) & \rightarrow & q, \text{ où } q \in Q_{||}, \varphi \text{ est une formule de Presburger} \\
 & & \text{telle que } FV(\varphi) = \{x_q \mid q \in Q \setminus (Q_{||} \cup Q_0)\}
\end{array}$$

Les états de $Q_.$ et $Q_{||}$ reconnaissent les termes dont la racine est "$\cdot$" et "||", respectivement. Les états de Q_- reconnaissent les termes de Var, et les états de Q_0 reconnaissent 0.

6.2 Une construction générique des ensembles des accessibles pour les PRS

Nous proposons dans cette section un algorithme *générique* qui calcule l'ensemble des accessibles des systèmes PRS et qui est paramétré par une procédure Ω d'analyse d'accessibilté des systèmes de réécritures de multiensembles, combinée avec un algorithme d'analyse des systèmes à pile. L'ensemble produit est soit *exactement* égal à l'ensemble des accessibles par le PRS si Ω calcule un ensemble semilinéaire qui représente *exactement* les accessibles par le système de réécritures de multiensembles sous-jacent à $\mathcal{R}$, c-à-d. si ce système de réécriture de multiensembles est effectivement semilinéaire ; soit une *sur-approximation* de l'ensemble des accessibles dans le cas où Ω calcule un ensemble semilinéaire qui représente une sur-approximation des accessibles par ce système de réécritures de multiensembles.

Plus précisément, pour définir la notion de semilinéarité des systèmes de réécriture de multiensembles, nous étendons la notion d'image de Parikh aux ensembles de paral-termes comme suit : Si t est le paral-terme $t = \|(X_1, X_2, \ldots, X_n)$, nous définissons $Parikh(t)$ comme étant l'image de Parikh du mot $X_1 X_2 \cdots X_n$ sur Var, c-à-d.,

$$Parikh(t) = Parikh(X_1 X_2 \cdots X_n)$$

et nous généralisons cette définition aux ensembles de paral-termes de manière standard. Un ensemble de paral-termes $\mathcal{L}$ est semilinéare si $Parikh(\mathcal{L})$ est un ensemble semilinéaire de vecteurs d'entiers.

Nous définissons alors la notion de classes de systèmes de réécriture de multiensembles effectivement semilinéaires comme suit :

Définition 6.2.1 *Une classe $\mathcal{C}$ de systèmes de réécriture de multiensembles est* effectivement semilinéaire *s'il existe un algorithme Ω qui calcule, pour chaque ensemble semilinéaire de paral-termes $\mathcal{L}$ et chaque système $M \in \mathcal{C}$, un ensemble semilinéaire $\Omega(M, \mathcal{L}) = M^*(\mathcal{L})$.*

Dans ce cas, la construction que nous donnons, instanciée avec l'algorithme Ω, produit un CH-automate qui représente l'ensemble des accessibles *exact* pour tout système dans PRS[$\mathcal{C}$][1] et tout ensemble de configurations (termes de processus) donné par un CH-automate. En général, si nous utilisons un algorithme Ω qui construit des sur-approximations semilinéaires $\Omega(M, \mathcal{L})$ des ensemble $M^*(\mathcal{L})$, notre construction instanciée par cet algorithme calcule des sur-approximations des ensembles des accessibles de tout système PRS.

Nous fixons pour le reste de cette section un PRS $\mathcal{R}$ défini sur un ensemble de variables Var, tel que $\mathcal{R} = \mathcal{R}_p \cup \mathcal{R}_s$ où $\mathcal{R}_p$ (resp. $\mathcal{R}_s$) est le système de réécriture de multiensemble (resp. le système de réécriture préfixe) induit par $\mathcal{R}$. Nous supposons que $\mathcal{R}_p$ appartient à une classe de réécriture de multiensembles effectivement semilinéaire $\mathcal{C}$. Soit alors Ω l'algorithme correspondant qui permet de calculer les ensembles semilinéaires $\Omega(M, \mathcal{L}) = M^*(\mathcal{L})$ pour tout $M \in \mathcal{C}$ et tout ensemble semilinéaire de paral-termes $\mathcal{L}$. Pour des raisons techniques (que nous expliquons dans la sous-section suivante), nous supposons également que $\mathcal{C}$ est *fermée* :

Définition 6.2.2 *Une classe $\mathcal{C}$ de systèmes de réécriture de multiensembles est* fermée *si pour chaque système $M \in \mathcal{C}$, et chaque variables de processus X, Y, le système $M \cup \{X \to Y\}$ appartient aussi à $\mathcal{C}$.*

Soit $\mathcal{L}$ un ensemble de termes de process reconnu par un CH-automate $\mathcal{A} = (Q, \Sigma, F, \Delta)$. Dans la suite de cette section, nous définissons un CH-automate $\mathcal{A}^{\Omega}_{\mathcal{R}} = (\widetilde{Q}, \Sigma, \widetilde{F}, \widetilde{\Delta})$ qui reconnaît $Post^*(\mathcal{L})$, où $\widetilde{Q}$ est l'ensemble des états, $\widetilde{F}$ est l'ensemble des états finaux, et $\widetilde{\Delta}$ l'ensemble des règles.

En plus de l'algorithme Ω (qui est un paramètre de notre construction) qui calcule les accessibles des systèmes de $\mathcal{C}$, notre construction utilise un algorithme qui calcule les ensembles des accessibles des systèmes de réécriture préfixe. C-à-d., nous utilisons le fait que pour chaque ensemble de seq-termes définissant un ensemble régulier (de mots) de configurations L, et chaque système de réécriture préfixe P, l'ensemble $P^*(L)$ est aussi un ensemble régulier de mots et peut être construit de manière effective, en utilisant par exemple l'algorithme de [Cau92] [2].

6.2.1 Les états de $\mathcal{A}^{\Omega}_{\mathcal{R}}$

Nous associons à chaque variable de processus X dans Var un état q_X qui l'identifie ($L_{q_X} = \{X\}$). Soit alors $Q_{\mathcal{R}}$ l'ensemble des états $\{q_X \mid X \in Var\}$. Soit q un état de

[1] Rappelons que les systèmes de PRS[$\mathcal{C}$] sont les PRS dont les systèmes de réécritures de multiensembles sous-jacents sont dans $\mathcal{C}$.

[2] Ici, nous identifions le seq-terme $X_1 \cdot \ldots \cdot X_n$ et le mot $X_1 \ldots X_n$.

$Q \cup Q_{\mathcal{R}}$. L'idée est la suivante : Pour déterminer qu'un terme t est un successeur de L_q, l'automate doit l'annoter avec un des états suivants : $(q,-), (q,0), (q,\cdot)$, ou $(q,||)$ selon la nature de la racine de t. Les états $(q,-)$ et $(q,0)$ reconnaissent les successeurs de L_q qui sont respectivement dans Var ou nuls (égaux à 0), et les états $(q,\cdot)$ et $(q,||)$ reconnaissent les successeurs de L_q dont la racine est étiquetée par "$\cdot$" ou "$||$", respectivement. Nous devons distinguer entre les successeurs qui ont des racines différentes pour nous assurer que les termes acceptés par l'automate ont une "bonne" structure, dans le sens que la racine d'un fils d'un nœud étiqueté par un "$||$" (resp. "$\cdot$") est différente de "$||$" (resp. de "$\cdot$"), et qu'un terme t est soit nul, et par conséquent égal à 0, soit il ne contient aucune feuille étiquetée par 0. Ceci peut être assuré, par exemple, en imposant qu'un nœud annoté par un état $(q,||)$ ne peut pas avoir un fils annoté par un état $(q',||)$. De même, le fils d'un nœud annoté par un état $(q,\cdot)$ ne peut pas être annoté par un état $(q',\cdot)$. L'ensemble d'états $\widetilde{Q}$ est donc défini comme suit, où $Q = Q_0 \cup Q_- \cup Q_\cdot \cup Q_{||}$ tel que décrit dans le paragraphe 6.1.3 :

- $\widetilde{Q}_0 = Q_0 \cup \{(q,0) \mid q \in Q \cup Q_{\mathcal{R}}\}$,
- $\widetilde{Q}_- = Q_- \cup Q_{\mathcal{R}} \cup \{(q,-) \mid q \in Q \cup Q_{\mathcal{R}}\}$,
- $\widetilde{Q}_\cdot = Q_\cdot \cup \{(q,\cdot) \mid q \in Q \cup Q_{\mathcal{R}}\}$,
- $\widetilde{Q}_{||} = Q_{||} \cup \{(q,||) \mid q \in Q \cup Q_{\mathcal{R}}\}$.

L'ensemble des états finaux est défini par :

$$\widetilde{F} = \{q, (q,-), (q,0), (q,\cdot), (q,||) \mid q \in F\}.$$

6.2.2 Définition des règles de $\mathcal{A}^{\Omega}_{\mathcal{R}}$

Nous commençons par introduire quelques notions techniques que nous utilisons dans la définition des règles de l'automate.

6.2.2.1 Forme normale transitive

Comme nous allons le voir plus tard, pour définir l'ensemble des règles de $\mathcal{A}^{\Omega}_{\mathcal{R}}$, nous devons caractériser les paral-successeurs (c-à-d., les successeurs qui sont des paral-termes) par $\mathcal{R}$ d'un ensemble de paral-termes, et les seq-successeurs (c-à-d., les successeurs qui sont des seq-termes) par $\mathcal{R}$ d'un ensemble de seq-termes. Soit ${\mathcal{R}'}_p$ le système de réécriture de multiensemble ${\mathcal{R}'}_p = \mathcal{R}_p \cup \{X \to Y \mid Y \in \mathcal{R}^*(X)\}$, et ${\mathcal{R}'}_s$ le système de réécriture préfixe ${\mathcal{R}'}_s = \mathcal{R}_s \cup \{X \to Y \mid Y \in \mathcal{R}^*(X)\}$. Il est facile de voir que pour un seq-terme t donné (resp. un paral-terme t), ${\mathcal{R}'_s}^*(t)$ est l'ensemble de tous les seq-termes qui peuvent être obtenus en appliquant itérativement $\mathcal{R}$ à t (resp. ${\mathcal{R}'_p}^*(t)$ est l'ensemble de tous les paral-termes qui peuvent être obtenus en appliquant itérativement $\mathcal{R}$ à t) :

Lemme 6.2.1 *Soit un seq-terme t (resp. un paral-terme t), ${\mathcal{R}'_s}^*(t)$ est l'ensemble de tous les seq-termes t' t.q. $t' \in \mathcal{R}^*(t)$ (resp. ${\mathcal{R}'_p}^*(t)$ est l'ensemble de tous les paral-termes t' t.q. $t' \in \mathcal{R}^*(t)$).*

Le problème qui se pose alors est de pouvoir calculer ces systèmes ${\mathcal{R}'}_s$ et ${\mathcal{R}'}_p$. Bien sûr, en théorie, nous pouvons le faire pour tout PRS $\mathcal{R}$ en utilisant le résultat de [May98] qui donne une procédure de décision pour le problème d'accessibilité entre

deux termes de processus. D'un point de vue pratique, ceci aurait été très coûteux puisque la technique de [May98] utilise la procédure de décision pour le problème d'accessibilité pour les réseaux de Petri qui est EXPSPACE-difficile. Dans notre cas, nous pouvons éviter cette complexité. En effet, dans notre cas, ces systèmes peuvent être calculés effectivement par une procédure itérative comme suit : $\mathcal{R}'_s = \mathcal{R}^s_n$ et $\mathcal{R}'_p = \mathcal{R}^p_n$, où les sequences $(\mathcal{R}^s_i)_{i\geq 0}$ et $(\mathcal{R}^p_i)_{i\geq 0}$ sont définies par :

– $\mathcal{R}^s_0 = \mathcal{R}_s$, et $\mathcal{R}^p_0 = \mathcal{R}_p$,

– $\mathcal{R}^s_{i+1} = \mathcal{R}^s_i \cup \{X \to Y \mid Y \in \Omega(\mathcal{R}^p_i, X)\}$, et $\mathcal{R}^p_{i+1} = \mathcal{R}^p_i \cup \{X \to Y \mid Y \in \mathcal{R}^{s*}_i(X)\}$,

et n est tel que $\mathcal{R}^s_n = \mathcal{R}^s_{n+1}$ et $\mathcal{R}^p_n = \mathcal{R}^p_{n+1}$.

Notons que les séquences $(\mathcal{R}^s_i)_{i\geq 0}$ et $(\mathcal{R}^p_i)_{i\geq 0}$ sont finies puisqu'il existe un nombre fini de paires (X, Y). Par conséquent, l'indice n existe et est fini.

Notons également que cette procédure est effective puisque $\mathcal{R}_p$ est dans la classe $\mathcal{C}$ qui est fermée, et donc pour chaque i, $\mathcal{R}^p_i$ est dans $\mathcal{C}$, ce qui permet d'appliquer la procédure Ω à $\mathcal{R}^p_i$ dans $\Omega(\mathcal{R}^p_i, X)$. De plus, comme pour tout i, $\mathcal{R}^s_i$ est un système de réécriture préfixe, les algorithmes de [Cau92, BEM97, EHRS00] peuvent alors être utilisés pour calculer les $\mathcal{R}^{s*}_i(X)$. Observons aussi que le système $\mathcal{R}'_p$ obtenu est dans $\mathcal{C}$ puisque cette classe est fermée. Nous pouvons donc appliquer l'algorithme Ω pour calculer les ensembles des accessibles par le système $\mathcal{R}'_p$.

6.2.2.2 Un PRS sur l'alphabet des états

Les règles des CH-automates que nous manipulons sont de la forme $\cdot(L) \to q$ ou $||(\varphi) \to q$, où L et φ expriment des contraintes sur les séquences d'*états*. La construction de l'automate $\mathcal{A}^{\Omega}_{\mathcal{R}}$ rajoute de nouvelles transitions en modifiant les contraintes qui apparaissent dans ces règles en fonction du résultat de l'application des systèmes $\mathcal{R}'_s$ et $\mathcal{R}'_p$ (ceci sera expliqué en détails plus tard). Seulement, puisque les contraintes dans les règles portent sur des séquences sur l'alphabet des états et non sur l'alphabet Var, nous avons besoin de considérer d'autres systèmes de réécriture S_p et S_s qui simulent les règles de $\mathcal{R}'_p$ et $\mathcal{R}'_s$, et qui manipulent des termes où les états q_X remplacent les variables X.

Soit alors α une substitution définie par $\alpha(X) = q_X$, pour tout $X \in Var$. α est étendue aux termes et aux (ensembles de) règles PRS de manière standard. Nous définissons alors les systèmes S_p et S_s comme suit :

$$\begin{aligned} S_p &= \alpha(\mathcal{R}'_p) \cup \{q \to (q, -); q \to (q, \cdot) \mid q \in Q \cup Q_{\mathcal{R}}\} \\ S_s &= \alpha(\mathcal{R}'_s) \cup \{q \to (q, -); q \to (q, ||) \mid q \in Q \cup Q_{\mathcal{R}}\} \end{aligned}$$

Pour comprendre le rôle des règles supplémentaires (qui ne sont pas obtenues par substitution), considérons par exemple que $X \to Y||Z \in \mathcal{R}'_p$. Alors, en partant de q_X le système S_p permet de dériver $||(q_Y, q_Z)$, mais aussi des termes tels que $||\big((q_Y, \cdot), (q_Z, -)\big)$ et $||\big(q_Y, (q_Z, \cdot)\big)$. Avoir le terme $||\big((q_Y, \cdot), (q_Z, -)\big)$ par exemple, exprime que si t_1 est un successeur de Y ayant "$\cdot$" comme racine, et si t_2 est un successeur de Z qui est dans Var, alors $||(t_1, t_2)$ est un successeur de X. De manière similaire, le terme $||\big(q_Y, (q_Z, \cdot)\big)$ exprime que pour tout successeur t de Z ayant "$\cdot$" comme racine, le terme $||(Y, t)$ est un successeur de X.

Observons que dans le cas du système S_s, la stratégie de réécriture préfixe implique que les termes de la forme $(q, -)$ et $(q, ||)$ n'apparaissent pas n'importe où dans les

séquences. En effet, si $X \to Y \cdot Z \in \mathcal{R}'_s$, alors en partant de q_X le système S_s permet de dériver $\cdot\big((q_Y, ||), q_Z\big)$ mais pas toujours $\cdot\big((q_Y, ||), (q_Z, -)\big)$. En effet, dans ce cas, si t_1 et t_2 sont respectivement des successeurs de Y et Z, il n'est pas toujours vrai que $\cdot(t_1, t_2)$ est un successeur de X.

Notons que le système S_p est dans $\mathcal{C}$ puisque $\mathcal{R}'_p$ l'est et que $\mathcal{C}$ est fermée. Par conséquent, nous pouvons utiliser la procédure Ω pour calculer les ensembles semi-linéaires de termes d'états accessibles par S_p.

6.2.2.3 L'ensemble des règles de $\mathcal{A}^{\Omega}_{\mathcal{R}}$

Dans ce qui suit, nous ne faisons pas de distinction entre un ensemble semilinéaire de paral-termes $\mathcal{L}$ et la formule φ qui définit $Parikh(\mathcal{L})$.

Nous donnons maintenant une construction inductive de l'ensemble de règles $\widetilde{\Delta}$. Nous montrons plus tard que cette construction termine. L'intuition exprimée par ces règles est donnée dans le paragraphe suivant.

Nous définissons $\widetilde{\Delta}$ comme le plus petit ensemble de règles contenant Δ, les règles $X \to q_X$, pour $X \in Var$, et tel que :

(β_1) Pour chaque état q :
- (a) $0 \to (q, 0) \in \widetilde{\Delta}$ si $q \in Q_0$,
- (b) $q \to (q, -) \in \widetilde{\Delta}$ si $q \in Q_- \cup Q_{\mathcal{R}}$,
- (c) $q \to (q, ||) \in \widetilde{\Delta}$ si $q \in Q_{||}$, et
- (d) $q \to (q, \cdot) \in \widetilde{\Delta}$ si $q \in Q_{\cdot}$.

(β_2) si $q \to q' \in \Delta$, alors :
- (a) $(q, -) \to (q', -) \in \widetilde{\Delta}$,
- (b) $(q, 0) \to (q', 0) \in \widetilde{\Delta}$,
- (c) $(q, \cdot) \to (q', \cdot) \in \widetilde{\Delta}$, et
- (d) $(q, ||) \to (q', ||) \in \widetilde{\Delta}$.

(β_3)
- (a) si $\cdot(L) \to (q, \cdot) \in \widetilde{\Delta}$, et $\{(q', -), (q', ||)\} \subseteq L$, alors $(q', 0) \to (q, 0)$, $(q', -) \to (q, -)$ et $(q', ||) \to (q, ||)$ sont dans $\widetilde{\Delta}$.
- (b) si $\cdot(L) \to (q, \cdot) \in \widetilde{\Delta}$ et $0 \in L$, alors $0 \to (q, 0)$ est dans $\widetilde{\Delta}$.

(β_4)
- (a) si $||(\varphi) \to (q, ||) \in \widetilde{\Delta}$ et $\vec{u}_p \models \varphi$ (où $\vec{u}_p$ est le vecteur ayant 1 dans la composante qui correspond à x_p et 0 partout ailleurs), et $p \in \{(q', -), (q', \cdot)\}$, alors $(q', 0) \to (q, 0)$, $(q', -) \to (q, -) \in \widetilde{\Delta}$ et $(q', \cdot) \to (q, \cdot) \in \widetilde{\Delta}$.
- (b) si $||(\varphi) \to (q, ||) \in \widetilde{\Delta}$ et $\vec{0} \models \varphi$ (où $\vec{0}$ est le vecteur ayant 0 dans toutes les composantes), alors $0 \to (q, 0) \in \widetilde{\Delta}$.

(β_5)
- (a) Soit X une variable de processus. Si $X \xrightarrow{*}_{\widetilde{\Delta}} (q, -)$, alors $\cdot\big(S^*_s(q_X)\big) \to (q, \cdot) \in \widetilde{\Delta}$, et $||\big(\Omega(S_p, q_X)\big) \to (q, ||) \in \widetilde{\Delta}$.

(b) Si $0 \xrightarrow{*}_{\widetilde{\Delta}} (q,0)$, alors $\cdot\big(S_s^*(0)\big) \to (q,\cdot) \in \widetilde{\Delta}$, et $||\big(\Omega(S_p,0)\big) \to (q,||) \in \widetilde{\Delta}$.

(β_6) si $||(\varphi) \to p$ est une règle de Δ, alors $||\big(\Omega(S'_p,\varphi)\big) \to (p,||) \in \widetilde{\Delta}$, où S'_p est une extension de S_p égale à (où les éléments q sont des états de $Q \cup Q_{\mathcal{R}}$) :

$$S'_p = S_p \cup \{q \to q_X \mid X \xrightarrow{*}_{\widetilde{\Delta}} (q,-)\} \cup \{q \to 0 \mid 0 \xrightarrow{*}_{\widetilde{\Delta}} (q,0)\}$$

(β_7) Soit $\cdot(L) \to p$ une règle de Δ, alors $\cdot\Big(S'^*_s\big(\alpha'(L)\big)\Big) \to (p,\cdot) \in \widetilde{\Delta}$, où α' est la substitution définie par

$$\alpha'(q) = \{q\} \cup \{q_X \mid X \xrightarrow{*}_{\Delta} q\},$$

et S'_s est une extension de S_s égale à (où les éléments q sont des états de $Q \cup Q_{\mathcal{R}}$) :

$$S'_s = S_s \cup \{q \to q_X \mid X \xrightarrow{*}_{\widetilde{\Delta}} (q,-)\} \cup \{q \to 0 \mid 0 \xrightarrow{*}_{\widetilde{\Delta}} (q,0)\}$$

L'ensemble de règles $\widetilde{\Delta}$ peut être construit itérativement comme la limite d'une séquence croissante $\widetilde{\Delta}_1 \subset \widetilde{\Delta}_2 \subset \ldots$. Après la création des règles, nous éliminons toutes les règles qui sont "contenues" dans d'autres règles. Plus précisément, si par exemple $\cdot(L) \to (q,\cdot)$ et $\cdot(L') \to (q,\cdot)$ sont deux règles de $\widetilde{\Delta}$ telles que $L' \subseteq L$, alors nous ne gardons que la règle $\cdot(L) \to (q,\cdot)$.

Nous montrons ci-dessous que cette procédure termine et produit une séquence finie de règles. En effet, les règles (β_3) et (β_4) créent des règles de la forme $(q,\cdots) \to (q',\cdots)$. Puisqu'il y a un nombre fini de paires (q,q'), il y a un nombre fini de telles règles. De la même manière, les règles (β_5), (β_6), et (β_7) ne peuvent être appliquées que s'il existe une nouvelle paire X,q telle que $X \xrightarrow{*}_{\widetilde{\Delta}} (q,-)$. Puisqu'il y a un nombre fini de telles paires, ces règles ne peuvent être appliquées qu'un nombre fini de fois.

Nous obtenons alors le résultat principal de ce chapitre :

Théorème 6.2.1 *Soit $\mathcal{C}$ une classe de règles de réécriture de multiensembles fermée et effectivement semilinéaire, et soit Ω un algorithme pour calculer les accessibles pour des systèmes dans $\mathcal{C}$. Soient $\mathcal{R}$ un système PRS[$\mathcal{C}$], $\mathcal{L}$ un CH-langage, $\mathcal{A} = (Q,\Sigma,F,\Delta)$ un CH-automate qui reconnaît $\mathcal{L}$, et $\mathcal{A}^{\Omega}_{\mathcal{R}}$ l'automate construit ci-dessus. Alors $Post^*(\mathcal{L}) = \mathcal{L}(\mathcal{A}^{\Omega}_{\mathcal{R}})$.*

Avant de donner la preuve de ce théorème qui est présentée dans la section 6.5, nous expliquons informellement l'intuition exprimée par ces règles (β_1)–(β_7).

6.2.2.4 Intuition

Nous expliquons d'abord ces règles de manière informelle.

– **Les règles** (β_1) **:** Ces règles expriment que $L_q \subseteq L_{(q,-)}$ si $q \in Q_- \cup Q_{\mathcal{R}}$, $L_q \subseteq L_{(q,||)}$ si $q \in Q_{||}$, $L_q \subseteq L_{(q,\cdot)}$ si $q \in Q_\cdot$, et $L_q \subseteq L_{(q,0)}$ si $q \in Q_0$.

– **Les règles** (β_2) **:** Ces règles expriment que si $L_q \subseteq L_{q'}$, alors tout successeur t de L_q est aussi un successeur de $L_{q'}$. Il y a quatre règles différentes parce que t peut être soit une variable de processus, soit 0, soit un terme ayant "$\cdot$" (resp. "$||$") comme racine.
– **Les règles** (β_3) **et** (β_4) **:** Ce sont des règles de simplification. Les règles (β_3) expriment que si $\cdot(X)$ est un successeur de L_q alors X l'est aussi, et que si $\cdot\big(||(t_1, \ldots, t_n)\big)$ est un successeur de L_q alors $||(t_1, \ldots, t_n)$ l'est aussi. Et nous avons la même chose si $\cdot(0)$ est un successeur de L_q. De manière similaire, les règles (β_4) expriment que si $||(X)$ est un successeur de L_q alors il en est de même pour X, et que si $||\big(\cdot(t_1, \ldots, t_n)\big)$ est un successeur de L_q, alors $\cdot(t_1, \ldots, t_n)$ l'est aussi. Le même raisonnement est valable si $||(0)$ est un successeur de L_q.
– **Les règles** (β_5) **:** Soit t un successeur d'un terme t' qui appartient à L_q tel que l'on ait les dérivations $t' = u_0 \xrightarrow{\mathcal{R}} u_1 \xrightarrow{\mathcal{R}} \cdots \xrightarrow{\mathcal{R}} u_J = t$. Il y a plusieurs cas en fonction des natures des termes u_i. Ces différents cas sont traités par les règles (β_5), (β_6), et (β_7). Les règles (β_5) considèrent le cas où pendant les dérivations $t' = u_0 \xrightarrow{\mathcal{R}} u_1 \xrightarrow{\mathcal{R}} \cdots \xrightarrow{\mathcal{R}} u_J = t$, il existe deux termes u_{J_1} et u_{J_2} ayant différentes racines. Dans ce cas, nécessairement, puisque les membres gauche et droit d'une même règle de $\mathcal{R}$ sont soit tous les deux des seq-termes, soit tous les deux des paral-termes, il existe un indice I, $J_1 \leq I \leq J_2$ tels que $u_I = 0$, ou $u_I = X$ pour une variable X. Soit I le plus grand indice tel que l'on ait ceci. Considérons d'abord le cas où X est un successeur de t' ($X \xrightarrow{*}_{\widetilde{\Delta}} (q, -)$), et t est un successeur de X (c'est le cas des règles (β_5a)). Il y a deux cas selon la racine de t :
  (a) Si t est de la forme $||(t_1, \ldots, t_n)$, alors il existe nécessairement n variables de processus $X_1, \ldots, X_n$ telles que pour chaque i, t_i est un successeur de X_i et $||(X_1, \ldots, X_n)$ est un successeur de X (nous avons forcément par le lemme 6.2.1 que $||(X_1, \ldots, X_n) \in {\mathcal{R}'_p}^*(X)$). Pour annoter t, l'automate procède d'une manière ascendante, il annote d'abord les fils t_i et ensuite la racine. Pour chaque i, puisque t_i est un successeur de X_i, t_i est soit annoté par q_{X_i} (s'il est égal à X_i), soit par $(q_{X_i}, -)$ ou $(q_{X_i}, \cdot)$ en fonction de la nature de sa racine. Il ne peut pas être annoté par $(q_{X_i}, ||)$ puisque son père est un nœud étiqueté par "$||$". Ensuite, en utilisant ces annotations, l'automate doit annoter la racine de t avec $(q, ||)$, pour exprimer que t est un successeur de L_q ayant une racine étiquetée par "$||$". Pour ce faire, l'automate doit avoir une règle de la forme $||(\varphi) \to (q, ||)$ telle que $Parikh(p_1 \cdots p_n) \models \varphi$, où $p_i \in \{q_{X_i}, (q_{X_i}, \cdot), (q_{X_i}, -)\}$. Plus précisément, la formule φ doit être telle que pour chaque variables de processus $Y_1, \ldots, Y_m$ telles que $||(Y_1, \ldots, Y_m)$ est un successeur de X (c-à-d. par le lemme 6.2.1, que $||(Y_1, \ldots, Y_m) \in {\mathcal{R}'_p}^*(X)$), $Parikh(p_1 \cdots p_m) \models \varphi$, où $p_i \in \{q_{Y_i}, (q_{Y_i}, \cdot), (q_{Y_i}, -)\}$. La formule φ peut alors être obtenue en appliquant itérativement à q_X le système S_p qui simule $\mathcal{R}'_p$. C-à-d., $\varphi = S_p^*(q_X) = \Omega(S_p, q_X)$.
  Dans le cas où 0 est un successeur de t' et t est un successeur de 0 (c'est le cas des règles (β_5b)), nous avons besoin d'avoir la règle $||(\varphi) \to (q, ||)$, où $\varphi = S_p^*(0) = \Omega(S_p, 0)$.
  (b) Si t est de la forme $\cdot(t_1, \ldots, t_{j-1}, X_j, \ldots, X_n)$, alors il existe nécessairement

une variable de processus X_{j-1} telle que $\cdot(X_{j-1}, X_j, \ldots, X_n)$ est un successeur de X (c-à-d. par lemme 6.2.1 que $\cdot(X_{j-1}, X_j, \ldots, X_n) \in {\mathcal{R}'_s}^*(X)$) , t_{j-1} est un successeur de X_{j-1}, et $\cdot(t_1, \ldots, t_{j-2})$ est un successeur du processus nul 0 (dans le cas séquentiel, nous ne pouvons réécrire que les fils les plus à gauche). Alors nécessairement, il existe $j-2$ variables de processus $Y_1, \ldots, Y_{j-2}$ et k indices $0 = i_1 < \cdots < i_k = j-2$ tels que pour chaque l, $1 \leq l \leq k$, $\cdot(Y_{i_l+1}, \ldots, Y_{i_{l+1}})$ est un successeur de 0, t_{i_l+1} est un successeur de Y_{i_l+1}, et $t_h = Y_h$ pour chaque h, $i_l + 1 < h \leq i_{l+1}$ (ceci est dû à la stratégie de réécriture préfixe). En d'autres termes, ceci veut dire que $\cdot(t_{j-1}, X_j, \ldots, X_n)$ est un successeur de X qui peut être successivement réécrit en

$$\cdot(t_{i_{k-1}+1}, Y_{i_{k-1}+2}, \ldots, Y_{i_k}, t_{j-1}, X_j, \ldots, X_n)$$

(puisque $\cdot(t_{i_{k-1}+1}, Y_{i_{k-1}+2}, \ldots, Y_{i_k})$ est un successeur de 0), ensuite en

$$\cdot(t_{i_{k-2}+1}, Y_{i_{k-2}+2}, \ldots, Y_{i_{k-1}}, t_{i_{k-1}+1}, \ldots, Y_{i_k}, t_{j-1}, X_j, \ldots, X_n)$$

et ainsi de suite jusqu'à arriver à

$$\cdot(t_1, Y_2, \ldots, Y_{i_2}, \ldots, t_{i_{k-1}+1}, \ldots, Y_{i_k}, t_{j-1}, X_j, \ldots, X_n).$$

Tous ces termes peuvent bien sûr être obtenus par $\mathcal{R}'_s$ (lemme 6.2.1).

Comme dans le cas précédent, l'automate procède d'une manière ascendante : il annote d'abord X_h, $j \leq h \leq n$ par q_{X_h}, t_{j-1} par $(q_{X_{j-1}}, -)$ ou $(q_{X_{j-1}}, ||)$ selon la nature de sa racine, et t_h, $1 \leq h \leq j-2$ soit par q_{Y_h}, $(q_{Y_h}, -)$, ou $(q_{Y_h}, ||)$. Ensuite, l'automate doit annoter la racine de t avec $(q, \cdot)$, pour exprimer que t est un successeur de L_q ayant une racine étiquetée par "$\cdot$". Pour ce faire, il utilise une règle de la forme $\cdot(L) \to (q, \cdot)$ t.q. $L = S^*_s(q_X)$.

De manière similaire, si nous avions que 0 est un successeur de t' et t est un successeur de 0 (ce cas est traité par les règles (β_5b)), alors l'automate aurait besoin d'une règle de la forme $\cdot(L) \to (q, \cdot)$ t.q. $L = S^*_s(0)$.

– **Les règles** (β_6) **:** Soit t un terme de la forme $||(t_1, \ldots, t_m)$ qui est un successeur d'un terme t' de la forme $||(t'_1, \ldots, t'_n)$ qui appartient à L_q. Supposons que pendant les dérivations $t' = u_0 \xrightarrow{\mathcal{R}} u_1 \xrightarrow{\mathcal{R}} \cdots \xrightarrow{\mathcal{R}} u_J = t$, chaque terme u_i a une racine étiquetée par "$||$". Soient alors $q_1, \ldots, q_n$ des états de $\mathcal{A}$ tels que pour chaque indice i, $t'_i \in L_{q_i}$, et soit ρ une règle $||(\varphi) \to q$ dans Δ telle que $Parikh(q_1 \cdots q_n) \models \varphi$ (cette règle permet à $\mathcal{A}$ d'annoter t' par q). Alors nécessairement, il existe des indices distincts $i_1, \ldots, i_k$, $j_1, \ldots, j_l$ appartenant à $\{1, \ldots, n\}$ tels que $k + l = n$, et des indices distincts $i'_1, \ldots, i'_{k'}$ et $j'_1, \ldots, j'_l$ de $\{1, \ldots, m\}$ tels que $k' + l = m$, et :
  – Pour chaque indice h dans $\{1, \ldots, l\}$, $t_{j'_h}$ est un successeur de t'_{j_h}.
  – Pour chaque indice h dans $\{1, \ldots, k\}$, X_h est un successeur de t'_{i_h},
  – $||(X'_1, \ldots, X'_{k'})$ est un successeur de $||(X_1, \ldots, X_k)$ (par le lemme 6.2.1, il l'est forcément par $\mathcal{R}'_p$),
  – Pour chaque indice h dans $\{1, \ldots, k'\}$, $t_{i'_h}$ est un successeur de X'_h.

Ceci veut dire que d'abord, pour chaque h dans $\{1,\ldots,k\}$, t'_{i_h} a été transformé en X_h, ensuite ces X_h ont interagit ensemble et se sont transformés en des X'_h ($||(X'_1,\ldots,X'_{k'})$ est un successeur de $||(X_1,\ldots,X_k)$), qui se sont alors transformés en $t_{i'_h}$.
Pour déterminer que t est un successeur de L_q, le CH-automate doit annoter les fils t_i comme suit : Il annote $t_{j'_h}$ par $(q_{j_h},-)$, ou $(q_{j_h},\cdot)$ pour h, $1\leq h\leq l$ (puisque $t_{j'_h}$ est un successeur de t'_{j_h}), et $t_{i'_h}$ par $(q_{X'_h},-)$ ou $(q_{X'_h},\cdot)$ pour h, $1\leq h\leq k'$. Ensuite, il doit annoter la racine de t avec $(q,||)$. Pour ce faire, il peut utiliser une nouvelle règle de la forme $||(\varphi')\to(q,||)$, où φ' est la formule définie dans les règles (β_6), obtenue après avoir :

1. substitué dans φ les états q_{i_h} par q_{X_h} pour chaque h t.q. $1\leq h\leq k$, (puisque X_h est un successeur de t'_{i_h}),
2. appliqué ensuite itérativement S_p à la nouvelle formule.

C'est-à-dire, $\varphi=\Omega(S'_p,\varphi)$.

– **Les règles** (β_7) **:** Soit t un terme de la forme $\cdot(t_1,\ldots,t_m)$ qui est un successeur d'un terme t' de la forme $\cdot(t'_1,\ldots,t'_n)$ appartenant à L_q. Supposons que pendant les dérivations $t'=u_0\xrightarrow{\mathcal{R}}u_1\xrightarrow{\mathcal{R}}\cdots\xrightarrow{\mathcal{R}}u_J=t$, chaque terme u_i a une racine étiquetée par "$\cdot$". Soient alors $q_1,\ldots,q_n$ des états de $\mathcal{A}$ tels que pour chaque indice i, $t'_i\in L_{q_i}$, et soit ρ une règle $\cdot(L)\to q$ dans Δ telle que $q_1\cdots q_n\in L$ (cette règle permet à $\mathcal{A}$ d'annoter t' avec q). Alors nécessairement, il existe deux indices j appartenant à $\{1,\ldots,n\}$, et g appartenant à $\{1,\ldots,m\}$ tels que
– 0 est un successeur de $\cdot(t'_1,\ldots,t'_{j-1})$,
– $\cdot(t_1,\ldots,t_g)$ est un successeur de 0, c-à-d., comme expliqué pour le cas des règles β_5, il existe g variables de processus $Y_1,\ldots,Y_g$ et K indices $0=i_1<\cdots<i_K=g-1$ tels que pour chaque H, $1\leq H\leq K$, $\cdot(Y_{i_H+1},\ldots,Y_{i_{H+1}})$ est un successeur de 0, t_{i_H+1} est un successeur de Y_{i_H+1}, et $t_h=Y_h$ pour chaque h, $i_H+1<h\leq i_{H+1}$ (ceci est dû à la stratégie de réécriture préfixe), et

   1. soit t_{g+1} est un successeur de t'_j, $m-(g+1)=n-j$, et pour tout h, $1\leq h\leq(n-j)$, $t'_{j+h}=t_{g+1+h}$;
   2. soit il existe des variables de processus $X_0,\ldots,X_k$ et $X'_1,\ldots,X'_{l'}$ tels que :
      – $n-(k+j)=m-(g+l')$;
      – X_0 est un successeur de t'_j,
      – pour chaque $h\in\{1,\ldots,k\}$, $t'_{h+j}=X_h$,
      – $\cdot(X'_1,\ldots,X'_{l'})$ est un successeur de $\cdot(X_0,\ldots,X_k)$ (par le lemme 6.2.1, il l'est forcément par $\mathcal{R}'_s$),
      – t_{g+1} est un successeur de X'_1,
      – pour chaque h t.q. $1<h\leq l'$, $t_{g+h}=X'_h$, et
      – pour chaque h t.q. $1\leq h\leq m-(g+l')$, $t'_{j+k+h}=t_{g+l'+h}$.

Ceci veut dire que $\cdot(t'_1,\ldots,t'_n)$ s'est transformé en $\cdot(t_1,\ldots,t_m)$ comme suit : D'abord $\cdot(t'_1,\ldots,t'_{j-1})$ est effacé (se transforme en 0), ensuite : (1) soit t'_j est réécrit en t_{g+1}, ou (2) t'_j est réécrit en X_0, qui peut interagir avec $\cdot(X_1,\ldots,X_k)$ (puisque pour chaque $h\in\{1,\ldots,k\}$, $t'_{h+j}=X_h$) pour devenir $\cdot(X'_1,\ldots,X'_{l'})$ ($\cdot(X'_1,\ldots,X'_{l'})$ est un successeur de $\cdot(X_0,\ldots,X_k)$), et ensuite X'_1 est transformé

en t_{g+1}. Enfin, dans les deux cas, après ces réécritures, $\cdot(t_1, \ldots, t_g)$ est rajouté au début de la séquence puisqu'il est un successeur du processus nul 0.
L'automate annote les fils comme suit : pour h, $1 \leq h \leq g$, t_h est soit annoté par q_{Y_h}, ou par $(q_{Y_h}, -)$ ou $(q_{Y_h}, ||)$ en fonction de sa nature. Selon que l'on a le cas 1 ou 2, nous avons :

1. t_{g+1} est annoté par $(q_j, -)$ ou $(q_j, ||)$,
2. ou t_{g+1} est annoté par $(q_{X'_1}, -)$ ou $(q_{X'_1}, ||)$, pour $1 < h \leq l'$, t_{g+h} est annoté avec $q_{X'_h}$, et pour h, $0 \leq h \leq m - (g + l')$, $t_{g+l'+h}$ est annoté avec q_{j+k+h}.

Pour annoter la racine de t avec $(q, \cdot)$, l'automate utilise une nouvelle règle de la forme $\cdot\big({S'_s}^*(L)\big) \rightarrow (q, \cdot)$. En effet, le système S'_s permet de simuler toutes ces étapes sur la séquence d'états $q_1 \cdots q_n$ comme suit : D'abord, $q_1 \cdots q_{j-1}$ est réécrit en 0 puisque 0 est un successeur de $\cdot(t'_1, \ldots, t'_{j-1})$. Cette transformation a lieu en utilisant des règles de la forme (1) $q_i \rightarrow q_X$ si t'_i se transforme en X ; (2) $q_X \rightarrow 0$ ou $q_X \cdot q_Y \rightarrow 0$, et (3) des règles $q_i \rightarrow 0$ qui sont dans S'_s dans le cas où t'_i se réécrit en 0, c-à-d. si $0 \xrightarrow{*}_{\widetilde{\Delta}} (q_i, 0)$.
Ensuite, selon les différents cas, nous avons :

1. q_j est réécrit en $(q_j, -)$ ou $(q_j, ||)$ en utilisant les règles $q_j \rightarrow (q_j, -)$ ou $q_j \rightarrow (q_j, ||)$ de S_s. Ceci est utile puisque t_{g+1} est annoté par l'un de ces deux états.
2. la règle $q_j \rightarrow q_{X_0}$ est appliquée (puisque X_0 est un successeur de t'_j), les états q_{j+h} sont substitués par q_{X_h} (puisque $t'_{h+j} = X_h$), pour $h \in \{1, \ldots, k\}$, et ensuite les règles de $\alpha(\mathcal{R}'_p)$ sont appliquées. Ceci permet de transformer $q_{X_0} \cdots q_{X_k}$ en $q_{X'_1} \cdots q_{X'_{l'}}$. La règle $q_{X'_1} \rightarrow (q_{X'_1}, -)$ ou $q_{X'_1} \rightarrow (q_{X'_1}, ||)$ est ensuite appliquée, puisque t_{g+1} est annoté par $(q_{X'_1}, -)$ ou $(q_{X'_1}, ||)$.

Ensuite, comme $\cdot(Y_1, \ldots, Y_{i_K})$ est un successeur de 0, $q_{Y_1} \cdots q_{Y_K}$ est rajouté au début de la séquence. La séquence d'états ainsi obtenue annote la séquence de termes $t_1, \ldots, t_m$, et donc en utilisant la règle $\cdot\big({S'_s}^*(L)\big) \rightarrow (q, \cdot)$, le terme t est annoté par $(q, \cdot)$.

6.2.3 Complexité de la construction

D'après leur définition, $|\mathcal{R}'_s| = O(|\mathcal{R}_s| + |Var|^2)$ et $|\mathcal{R}'_p| = O(|\mathcal{R}_p| + |Var|^2)$. Donc, $|S_s| = O(|\mathcal{R}_s| + |Var|^2 + |Q|)$ et $|S_p| = O(|\mathcal{R}_p| + |Var|^2 + |Q|)$. Soit L un langage régulier de mots reconnu par un automate fini de mots $\mathcal{A}$, $|L|$ représente le nombre de transitions dans $\mathcal{A}$. Il existe un algorithme qui calcule ${S_s}^*(L)$ avec une complexité polynômiale $O\big(\pi_s(|S_s|, |L|)\big)$ [Cau92].

Soit φ une formule de Presburger et $|\varphi|$ le nombre d'opérateurs dans φ. Soit $\pi_p(|S_p|, |\varphi|)$ la taille de ${S_p}^*(\varphi)$, et $Sat(|\varphi|)$ le temps nécessaire pour décider la satisfaisabilité de φ [3].

Soit $n_s = \max\{|L|, \cdot(L) \rightarrow q \in \Delta\}$, $n_p = \max\{|\varphi|, ||(\varphi) \rightarrow q \in \Delta\}$, et $n_\Delta = \max(n_s, n_p)$; alors dans $\widetilde{\Delta}$ il y a :

[3] D'après le théorème 2.1.8, $Sat(|\varphi|)$ est égale à $2^{2^{2^{O(|\varphi|)}}}$ si φ est une formule quelconque, et est dans NP si φ est de la forme $\exists x_1 \cdots \exists x_n . \phi$, où ϕ est une formule sans quantificateurs.

- $O(|Q| + |Var|)$ règles β_1,
- $O\big((|Q| + |Var|)^2\big)$ règles β_2, β_3, et β_4,
- $O(|Q| \cdot |Var| + |Var|^2)$ règles β_5,
- $O\big(|Var| \cdot (|Q| + |Var|) \cdot |\Delta|\big)$ règles β_6 avant simplification, et $O(|\Delta|)$ règles β_6 après simplification puisqu'il suffit de garder la formule φ' obtenue en considérant pour chaque q, toutes les variables X telles que $X \xrightarrow{*}_{\widetilde{\Delta}} (q, -)$, où $\widetilde{\Delta}$ est l'ensemble de toutes les règles.
- $O\big(|Var| \cdot (|Q| + |Var|) \cdot |\Delta|\big)$ règles β_7 avant simplification, et $O(|\Delta|)$ règles β_7 après simplification.

En plus, les règles ont les tailles suivantes :

- Chaque règle β_1, β_2, β_3, et β_4 a une taille constante.
- Une règle β_5 de la forme $\cdot(L) \to (q, \cdot)$ a une taille en $O(\pi_s(|S_s|, 1))$, et une règle β_5 de la forme $||(\varphi) \to (q, ||)$ a une taille en $O\big(\pi_p(|S_p|, 1)\big)$.
- Une règle β_6 a une taille en $O\Big(\pi_p\big(|S_p| + |Var|(|Q| + |Var|), n_p\big)\Big)$.
- Une règle β_7 a une taille en $O\Big(\pi_s\big(|S_s| + |Var|(|Q| + |Var|), n_s|Var|\big)\Big)$.

En plus, $Sat\Big(\pi_p\big(|S_p| + |Var|(|Q| + |Var|), n_p\big)\Big)$ temps est nécessaire pour calculer une règle β_4.

Dans la plupart des cas, calculer ${S_p}^*(\varphi)$ est au moins exponentiel. En plus, décider si une formule de Presburger est satisfaisable nécessite une triple exponentielle dans le cas général. Par conséquent, la complexité est dominée par ces procédures. Ainsi, pour calculer et stocker l'automate, $O\Big(\pi_p\big(|\mathcal{R}_p| + |Var|(|Q| + |Var|), n_\Delta\big)\Big)$ espace et $Sat\Big(\pi_p\big(|\mathcal{R}_p| + |Var|(|Q| + |Var|), n_\Delta\big)\big)\Big)$ temps sont nécessaires.

6.3 Exemple

Soit $\mathcal{A} = (Q, \Sigma, F, \Delta)$ un CH-automate qui reconnaît le terme t_1 de la figure (6.2), où $Q_\cdot = \{q\}$, $Q_{||} = \emptyset$, $Q_- = \{q_1, q_2, q_3\}$, $F = \{q\}$, et Δ est l'ensemble suivant de règles :

$$\Delta = \{X \to q_1, Y \to q_2, Z \to q_3, \cdot(q_1 q_2 q_3) \to q\}$$

Soit $\mathcal{R}$ le PRS suivant :

- $\mathcal{R}_1 : X \to A||A$,
- $\mathcal{R}_2 : A||A \to D$,
- $\mathcal{R}_3 : D \to F \cdot G$.

Nous montrons comment appliquer notre construction pour pouvoir reconnaître les termes de la figure 6.2 comme successeurs de t_1. Pour ce faire, nous avons besoin de calculer $\mathcal{R}'_s$ et $\mathcal{R}'_p$. Il est facile de voir que $\mathcal{R}'_s = \{X \to D, D \to F \cdot G\}$ et $\mathcal{R}'_p = \{X \to D, X \to A||A, A||A \to D\}$ puisque $D \in \mathcal{R}^*_p(X)$. Soit $Q_\mathcal{R} = \{q_X, q_A, q_D, q_F, q_G\}$. Les systèmes S_p et S_s sont définis par :

$$S_p = \{q_X \to q_D, q_X \to q_A||q_A, q_A||q_A \to q_D\} \cup \{q \to (q, -); (q, \cdot) \mid q \in Q \cup Q_\mathcal{R}\}$$

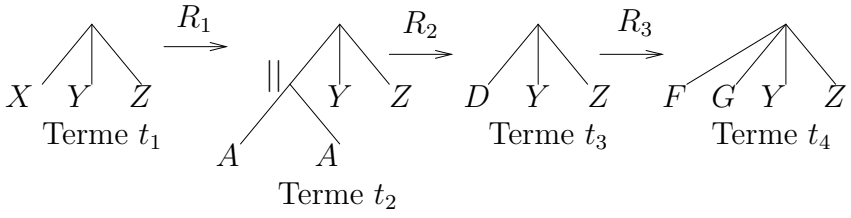


FIG. 6.2 – Un exemple

et

$$S_s = \{q_X \to q_D, q_D \to q_F \cdot q_G\} \cup \{q \to (q, -); (q, ||) \mid q \in Q \cup Q_{\mathcal{R}}\}$$

Nous allons calculer les règles $\widetilde{\Delta}$ (nous ne considérons que les règles dont nous avons besoin pour annoter les termes de la figure (6.2)) : $\widetilde{\Delta}$ contient les règles $X \to q_X, Y \to q_Y$, et $Z \to q_Z$. De plus, elle contient les règles suivantes :

(r_1) Les règles (β_1) impliquent que $q_1 \to (q_1, -)$ et $q_D \to (q_D, -)$ sont dans $\widetilde{\Delta}$,

(r_2) Puisque $X \to_{\widetilde{\Delta}} q_1 \to_{\widetilde{\Delta}} (q_1, -)$, les règles ($\beta_5$) impliquent que $||(\varphi) \to (q_1, ||)$ est dans $\widetilde{\Delta}$, où $\varphi = S_p^*(q_X)$ est la formule

$$\bigvee_{\substack{p \in \{q_X, q_D \\ (q_X,-),(q_D,-) \\ (q_X,\cdot),(q_D,\cdot)\}}} \big((x_p = 1) \wedge \forall p' \mid p' \neq p, x_{p'} = 0\big) \vee$$

$$\bigvee_{p_i \in \{q_A, (q_A,-),(q_A,\cdot)\}} \big((x_{p_1} + x_{p_2} = 2) \wedge \forall p \mid p \neq p_1, p_2, x_p = 0\big)$$

(r_3) Puisque $||(\varphi) \to (q_1, ||)$ est dans $\widetilde{\Delta}$, et $\vec{u}_D \models \varphi$ (où $\vec{u}_D$ est le vecteur ayant 1 dans la composante qui correspond à x_{q_D}, et 0 partout ailleurs), nous obtenons par les règles (β_4) que $(q_D, -) \to (q_1, -)$ est dans $\widetilde{\Delta}$,

(r_4) Puisque $\cdot(q_1 q_2 q_3) \to q$ est dans Δ, $X \to_{\widetilde{\Delta}} q_1 \to_{\widetilde{\Delta}} (q_1, -)$, $Y \to_{\Delta} q_2$, et $Z \to_{\Delta} q_3$, les règles (β_7) impliquent que

$$\cdot\Big(S_s'^*\big((q_1 + q_X)(q_2 + q_Y)(q_3 + q_Z)\big)\Big) \to (q, \cdot)$$

est dans $\widetilde{\Delta}$, où

$$S'_s = S_s \cup \{q_1 \to q_X\}.$$

Ceci veut dire que $\widetilde{\Delta}$ contient la règle $\cdot\Big(\big(q_1 + (q_1, -) + (q_1, ||) + q_X + q_D + (q_D, -) + (q_D, ||) + q_F q_G + (q_F, -) q_G + (q_F, ||) q_G\big)(q_2 + q_Y)(q_3 + q_Z)\Big) \to (q, \cdot)$.

Avec ces règles, tous les termes de la figure (6.2) peuvent être reconnus comme successeurs de t_1 par $\mathcal{R}$. En fait, tous ces termes peuvent être annotés par l'état final $(q, \cdot)$ comme suit :

– t_1 peut être annoté par $(q, \cdot)$ grâce à (r_4),
– Pour t_2, le sous terme $||(A, A)$ est annoté par $(q_1, ||)$ grâce à (r_2), et ensuite t_2 est annoté par $(q, \cdot)$ en utilisant (r_4),
– Concernant t_3, D est annoté par $(q_1, -)$ grâce à (r_1) et (r_3), et ensuite t_3 est annoté par $(q, \cdot)$ grâce à (r_4),
– Finalement, t_4 est annoté par $(q, \cdot)$ grâce à (r_4).

6.4 Applications

6.4.1 Calcul des ensembles des accessibles pour PAD

Corollaire 6.4.1 *Soit $\mathcal{R}$ un PAD et $\mathcal{L}$ un CH-langage reconnu par un CH-automate $\mathcal{A} = (Q, \Sigma, F, \Delta)$. Alors $Post^*(\mathcal{L})$ et $Pre^*(\mathcal{L})$ peuvent être effectivement représentés par des CH-automates $\mathcal{A}' = (Q', \Sigma, F', \Delta')$ où*

$$n_{\Delta'} = O\Big(2^{\left(|\mathcal{R}_p|+|Var|(|Q|+|Var|)\right)} + n_\Delta\Big)$$

Ces automates peuvent être construits en un temps en

$$Sat\Big(2^{\left(|\mathcal{R}_p|+|Var|(|Q|+|Var|)\right)} + n_\Delta\Big)$$

De plus, si les formules qui interviennent dans les règles de Δ sont de la forme $\exists x_1, \ldots, x_n \phi$, où ϕ est sans quantificateurs, alors ces automates peuvent être construits en

$$\mathbf{NTIME}\Big(2^{\left(|\mathcal{R}_p|+|Var|(|Q|+|Var|)\right)} + n_\Delta\Big)$$

Preuve : Ceci se déduit directement du (1) théorème précédent, (2) le fait que si $\mathcal{R}$ est un PAD, alors $\mathcal{R}$ est un PRS[BPP] et $\mathcal{R}^{-1}$ est un PRS[co-BPP] (puisque $Pre^*_{\mathcal{R}}(\mathcal{L}) = Post^*_{\mathcal{R}^{-1}}(\mathcal{L})$), et (3) le fait que BPP et co-BPP sont des classes de règles de réécriture de multiensembles fermées et effectivement semilinéaires. En effet, les BPP sont équivalents aux réseaux de Petri sans communication, et la relation d'accessibilité d'un réseau de Petri sans communication est effectivement semilinéaire [Esp97b]. Nous donnons ci-dessous la construction de $\mathcal{R}^*$ décrite dans [Esp97b] pour pouvoir estimer la complexité de notre procédure. Une preuve détaillée se trouve dans [Esp97b].

Soit $\mathcal{R}$ un BPP sur Var, nous donnons une formule de Presburger $\psi_{\mathcal{R}}$ qui caractérise $\mathcal{R}^*$. Plus précisément, si $Var = \{X_1, \ldots, X_n\}$, alors $\psi_{\mathcal{R}}$ a $x_1, \ldots, x_n, x'_1, \ldots, x'_n$ comme variables libres. La variable x_i correspond au nombre d'occurrences initial de X_i, et x'_i au nombre d'occurrences de X_i après application de $\mathcal{R}^*$. Pour exprimer ceci, nous écrivons $\psi_{\mathcal{R}}(x_1, \ldots, x_n, x'_1, \ldots, x'_n)$. Si $\mathcal{R} = \{r_1, \ldots, r_m\}$, nous avons :

$$\psi_{\mathcal{R}} = \exists y_1, \ldots, y_m; \bigvee_{\substack{Var' \subseteq Var \\ \mathcal{R}' \subseteq \mathcal{R} \\ (Var', \mathcal{R}') \in S}} \Big(\bigwedge_{X_i \in Var'} x_i > 0 \ \wedge \bigwedge_{X_i \notin Var'} x_i = 0 \ \wedge$$

$$\bigwedge_{X_i \in Var} x'_i = x_i + \sum_{r_j \in \mathcal{R}'} \big(W(r_j, X_i) - W(X_i, r_j)\big) y_j \Big)$$

où si $r_i = X \to Z_1 || \cdots || Z_m$, alors $W(X, r_i) = 1, W(Y, r_i) = 0$ si $X \neq Y$, et $W(r_i, Z) = |Z_1 \cdots Z_m|_Z$. De plus, S est l'ensemble des paires $(Var', \mathcal{R}')$ telles que si $Var' = \{Y_1, \ldots, Y_k\}$, alors pour toute variable X qui intervient dans les règles de $\mathcal{R}'$, nous pouvons à partir du terme $||(Y_1, \ldots, Y_k)$ appliquer les règles de $\mathcal{R}'$ et atteindre un terme $||(Y'_1, \ldots, Y'_{k'})$ tel qu'il existe un indice $i \leq k'$; $Y'_i = X$. Notons qu'il est facile de tester cette condition en temps polynômial en construisant un graphe dont les nœuds correspondent aux différentes variables X_i, et où un arc $X \to Y$ exprime qu'il existe une règle de $\mathcal{R}$ de la forme $X \to Y || Y_1 || \cdots || Y_k$.

Ceci implique que si $\mathcal{L}$ est un ensemble de paral-termes dont l'image de Parikh est décrite par une formule $\varphi(x_1, \ldots, x_n)$, alors $\mathcal{R}^*(\mathcal{L})$ est l'ensemble de paral-termes caractérisés par la formule

$$\exists x_1, \ldots, x_n; \psi_{\mathcal{R}}(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \wedge \varphi(x_1, \ldots, x_n)$$

De même, $\mathcal{R}^{-1^*}(\mathcal{L})$ est l'ensemble de paral-termes caractérisés par la formule

$$\exists x'_1, \ldots, x'_n; \psi_{\mathcal{R}}(x_1, \ldots, x_n, x'_1, \ldots, x'_n) \wedge \varphi(x'_1, \ldots, x'_n)$$

Donc, dans le cas où $\mathcal{R}$ est un BPP ou un co-BPP, nous avons :

$$\pi_p(|\mathcal{R}|, |\varphi|) = |\varphi| + O(|Var||\mathcal{R}|2^{|Var|+|\mathcal{R}|})$$

Nous en déduisons alors à partir de la section 6.2.3, que

$$n_{\Delta'} = O\Big(2^{\big(|\mathcal{R}_p|+|Var|(|Q|+|Var|)\big)} + n_\Delta\Big)$$

et que ces automates peuvent être construits en un temps en

$$Sat\Big(2^{\big(|\mathcal{R}_p|+|Var|(|Q|+|Var|)\big)} + n_\Delta\Big)$$

De plus, si les formules qui interviennent dans les règles de Δ sont de la forme $\exists x_1, \ldots, x_n \phi$, où ϕ est sans quantificateurs, alors les formules obtenues par la procédure décrite en haut gardent cette forme. Donc, leur satisfaisabilité peut être testée en un temps NP par rapport à leur taille (théorème 2.1.8), c-à-d. en

$$\mathbf{NTIME}\Big(2^{\big(|\mathcal{R}_p|+|Var|(|Q|+|Var|)\big)} + n_\Delta\Big)$$

□

6.4.2 Model-checking des formules EF pour les systèmes PRS

Nous montrons dans cette section que notre procédure permet le model-checking des formules EF pour tous les PRS[co-$\mathcal{C}$], où $\mathcal{C}$ est une classe fermée et effectivement semilinéaire. En particulier, elle permet le model-checking des formules EF pour les systèmes PAD.

La logique EF est un fragment de la logique temporelle CTL dont les formules sont définies par la grammaire suivante :

$$\phi ::= p \mid \neg\phi \mid \phi \wedge \phi \mid EX\phi \mid EF\phi$$

où p est une proposition atomique. $EX\phi$ exprime qu'il existe un état suivant pour lequel ϕ est vraie, et $EF\phi$ exprime qu'il existe une exécution le long de laquelle ϕ est vraie dans un état futur. Plus précisément, étant donné un PRS $\mathcal{R}$, un ensemble de termes de processus $\mathcal{L}$ satisfait une formule ϕ si $\mathcal{L} \subseteq [\![\phi]\!]$, où $[\![\phi]\!]$ est un ensemble de $\mathcal{T}$ défini inductivement comme suit :

- $[\![p]\!]$ est l'ensemble des termes de processus pour lesquels p est vraie,
- $[\![\neg\phi]\!] = \mathcal{T} \setminus [\![\phi]\!]$,
- $[\![\phi_1 \wedge \phi_2]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$,
- $[\![EX\phi]\!] = Pre([\![\phi]\!])$,
- $[\![EF\phi]\!] = Pre^*([\![\phi]\!])$.

Le problème du model checking d'une formule EF consiste à décider si un ensemble de termes de processus $\mathcal{L}$ satisfait une formule ϕ, c-à-d. si $\mathcal{L} \subseteq [\![\phi]\!]$. Nous obtenons alors :

Proposition 6.4.1 *Soit $\mathcal{C}$ une classe de systèmes de réécriture de multiensembles qui soit fermée et effectivement semilinéaire. Soit $\mathcal{R}$ un PRS[co-$\mathcal{C}$] et ϕ une formule EF. Etant donnés des CH-automates représentant $[\![p]\!]$, un CH-automate reconnaissant $[\![\phi]\!]$ peut être effectivement calculé.*

Preuve : La preuve est par induction structurelle sur ϕ :

- Le cas où $\phi = p$ est direct.
- $\phi = \neg\phi'$, alors $[\![\phi]\!] = \overline{[\![\phi']\!]}$ qui est un CH-langage effectivement calculable puisque $[\![\phi']\!]$ l'est, et les CH-langages sont effectivement fermés par complémentation ;
- $\phi = \phi_1 \wedge \phi_2$, alors $[\![\phi]\!] = [\![\phi_1]\!] \cap [\![\phi_2]\!]$, et le résultat est dû au fait que les CH-langages sont effectivement fermés par intersection ;
- $\phi = EX\phi'$. Dans ce cas $[\![\phi]\!] = Pre([\![\phi']\!])$. Il est facile de voir que pour tout PRS $\mathcal{R}$, et tout ensemble de termes de processus $\mathcal{L}$ reconnu par un CH-automate, $Pre(\mathcal{L})$ est effectivement reconnu par un CH-automate ;
- $\phi = EF\phi'$. Dans ce cas $[\![\phi]\!] = Pre^*([\![\phi']\!])$ qui est effectivement représentable par un CH-automate puisque $\mathcal{R}^{-1}$ est un PRS[$\mathcal{C}$], où $\mathcal{C}$ une classe de systèmes de réécriture de multiensembles qui est fermée par rapport à Var et effectivement semilinéaire (voir le théorème 6.2.1).

□

Ceci constitue une procédure de décision pour le problème de model-checking des formules EF pour les systèmes dans PRS[co-$\mathcal{C}$] où $\mathcal{C}$ est une classe de systèmes de réécriture de multiensembles fermée et effectivement semilinéaire : Il s'agit de construire $[\![\phi]\!]$ et vérifier que $\mathcal{L} \subseteq [\![\phi]\!]$.

Puisqu'un PAD est un PRS[co-BPP], et que la classe co-BPP est fermée et effectivement semilinéaire, nous obtenons comme corollaire immédiat de cette proposition que le model-checking des formules EF est décidable pour PAD :

Corollaire 6.4.2 *Soit $\mathcal{R}$ un PAD et ϕ une formule EF. Etant donnés des CH-automates représentant $[\![p]\!]$, un CH-automate reconnaissant $[\![\phi]\!]$ peut être effectivement calculé.*

Nous obtenons donc une procédure pour le model-checking des formules EF pour les systèmes PAD. Pour calculer $[\![\phi]\!]$, chaque étape de calcul d'intersection et de Pre est polynômiale, alors que la complémentation et le calcul du Pre^* augmentent exponentiellement les tailles des automates. Si n est le nombre de négations et d'opérations Pre^* dans la formule ϕ, la complexité totale de l'algorithme de calcul de $[\![\phi]\!]$ est donc en $O\big(tour(n)\big)$, où $tour(0) = 0$ et $tour(i+1) = 2^{tour(i)}$.

Mayr [May98] a déjà montré que le problème de model-checking des formules EF est décidable et PSPACE-difficile pour PAD. La procédure de décision qu'il considère est aussi non élémentaire

6.4.3 Calcul de sur-approximations des ensembles des accessibles

Pour toute la classe de PRS, la construction de l'automate $\mathcal{A}^{\Omega}_{\mathcal{R}}$ peut être utilisée pour calculer des sur-approximations de l'ensemble des accessibles. En effet, dans cette construction Ω peut être instanciée par une procédure qui calcule des sur-approximations semilinéaires de l'ensemble des accessibles pour des systèmes de réécriture de multiensembles. C-à-d., telles que pour chaque ensemble semilinéaire L et chaque système de réécriture de multiensembles M, $\Omega(M, L)$ est semilinéaire et $Post^*_M(L) \subseteq \Omega(M, L)$.

Comme exemples de telles procédures, nous pouvons citer la procédure de Karp et Miller qui calcule l'ensemble de couverture de l'ensemble des accessibles, ou des procédures qui construisent des invariants linéaires basés sur des contraintes de flot [EM96], etc. Nous pouvons aussi utiliser des procédures (exactes ou approchées) d'analyse d'accessibilité des automates à compteurs. Nous pouvons citer par exemple les procédures basées sur la représentation des contraintes arithmétiques par des polyhèdres, et l'utilisation d'opérateurs d'élargissement pour forcer la terminaison du calcul [CH78]. Il y a également des procédures basées sur la représentation des contraintes arithmétiques par des automates régulier de mots, et l'utilisation de méta-transitions pour accélérer la convergence du calcul [BW94, BW98], etc [DBR01, FL02].

Nous pouvons aussi combiner toutes ces différentes procédures. En effet, puisque les approximations semilinéaires de l'ensemble des accessibles caculées par les différentes procédures sont en général différentes, leur intersection constitue une approximation plus précise.

6.5 Preuve du théorème 6.2.1

Dans cette section, puisqu'il n'y un pas de confusion possible concernant Ω, nous écrivons simplement $\mathcal{R}^*_p(L)$ à la place de $\Omega(\mathcal{R}_p, L)$. Pour montrer le théorème 6.2.1, il suffit de montrer les lemmes suivants :

Lemme 6.5.1 *Pour chaque terme v, et chaque $q \in Q \cup Q_{\mathcal{R}}$ nous avons :*
- *$v \xrightarrow{*}_{\widetilde{\Delta}} (q, 0) \Rightarrow v \in Post^*(L_q)$, et $v = 0$,*

– $v \xrightarrow{*}_{\widetilde{\Delta}} (q,-) \Rightarrow v \in Post^*(L_q)$, *et* $v \in Var$,
– $v \xrightarrow{*}_{\widetilde{\Delta}} (q,||) \Rightarrow v \in Post^*(L_q)$, *et* $racine(v) = ||$,
– $v \xrightarrow{*}_{\widetilde{\Delta}} (q,\cdot) \Rightarrow v \in Post^*(L_q)$, *et* $racine(v) = \cdot$.

Preuve :

Nous montrons par induction sur i que pour chaque terme v, et chaque $q \in Q \cup Q_{\mathcal{R}}$ nous avons :

(a) $[v \xrightarrow{*}_{\widetilde{\Delta}_i} (q,-) \Rightarrow v \in Post^*(L_q)$, et $v \in Var]$,
$[v \xrightarrow{*}_{\widetilde{\Delta}_i} (q,0) \Rightarrow v \in Post^*(L_q)$, et $v = 0]$,
(b) $v \xrightarrow{*}_{\widetilde{\Delta}_i} (q,||) \Rightarrow v \in Post^*(L_q)$, et $racine(v) = ||$,
(c) $v \xrightarrow{*}_{\widetilde{\Delta}_i} (q,\cdot) \Rightarrow v \in Post^*(L_q)$, et $racine(v) = \cdot$.

De plus, nous montrons par induction sur i que pour chaque $q \in Q \cup Q_{\mathcal{R}}$ nous avons :

(d) si $\cdot(L) \to (q,\cdot) \in \widetilde{\Delta}_i$, alors pour chaque $q_1, \ldots, q_n$ tels que $q_1 \cdots q_n \in L$, pour chaque $t_1, \ldots, t_n$ tels que $t_j \in Post^*(L_{p_j})$ si $q_j \in \{(p_j,-),(p_j,||)\}$, et $t_j \in L_{q_j}$ si $q_j \in Q \cup Q_{\mathcal{R}}$, $racine(t_j) \neq \cdot$, et $t_j \neq 0$, nous avons $\cdot(t_1, \ldots, t_n) \in Post^*(L_q)$,
(e) si $||(\varphi) \to (q,||) \in \widetilde{\Delta}_i$, alors pour chaque $q_1, \ldots, q_n$ tels que $Parikh(q_1 \cdots q_n) \models \varphi$, pour chaque $t_1, \ldots, t_n$ tel que $t_j \in Post^*(L_{p_j})$ si $q_j = \{(p_j,-),(p_j,\cdot)\}$, et $t_j \in L_{q_j}$ si $q_j \in Q \cup Q_{\mathcal{R}}$, $racine(t_j) \neq ||$, et $t_j \neq 0$, nous avons $||(t_1, \ldots, t_n) \in Post^*(L_q)$,

Ces points doivent être montrés simultanément :

• $i = 0$. $\widetilde{\Delta}_0$ contient les règles Δ, en plus des règles (β_1) et (β_2). Il est alors facile de voir que les propriétés sont satisfaites.

• $i > 0$. Nous montrons d'abord les propriétés (d) et (e). Pour des raisons de simplicité, nous ne mentionnons pas la nature des termes t_i (leurs racines et les informations indiquant s'ils sont nuls ou pas) considérées dans les différents points. Cette propriété concernant les racines peut être vérifiée facilement.

⋆ Supposons que $\cdot(L) \to (q,\cdot) \in \widetilde{\Delta}_i \setminus \widetilde{\Delta}_{i-1}$. Il y a plusieurs cas en fonction des règles d'inférence par lesquelles cette règle a été créée :

1. $\cdot(L) \to (q,\cdot)$ est la conséquence d'une β_5-règle. Soit alors X une variable de processus telle que $L = S_s^*(q_X)$ et $X \xrightarrow{*}_{\widetilde{\Delta}_{i-1}} (q,-)$. Nous obtenons par induction que $X \in Post^*(L_q)$. Soit $p_1 \cdots p_n \in S_s^*(q_X)$. Soient alors $X_1, \ldots, X_n$ tels que $p_j = q_{X_j}$, $p_j = (q_{X_j},-)$, ou $p_j = (q_{X_j},||)$ (ceci est dû à la définition de S_s). Nous déduisons par la définition de S_s que $\cdot(X_1, \ldots, X_n) \in {\mathcal{R}'_s}^*(X)$, c-à-d. $\cdot(X_1, \ldots, X_n) \in Post^*(X)$. Nous allons montrer la propriété par induction sur le nombre l d'états de le forme $p_j = (q_{X_j},-)$ ou $p_j = (q_{X_j},||)$ dans $p_1 \cdots p_n$:
   – $l = 0$. La propriété est vraie puisque $L_{q_{X_j}} = \{X_j\}$,
   – $l > 0$. Il y a deux cas :
      (a) $p_1 = (q_{X_1},-)$ ou $p_1 = (q_{X_1},||)$. Nous avons donc nécessairement $q_{X_1} \cdot p_2 \cdots p_n \in S_s^*(q_X)$ et $p_1 \cdots p_n \in S_s(q_{X_1} \cdot p_2 \cdots p_n)$. Comme le nombre d'états de la forme $p_j = (q_{X_j},-)$ ou $p_j = (q_{X_j},||)$ dans $q_{X_1} \cdot p_2 \cdots p_n$ est $l-1$, nous déduisons par induction que pour chaque $t_2, \ldots, t_n$ tels que $t_j \in Post^*(L_{q_{X_j}})$ si $p_j \in \{(q_{X_j},-),(q_{X_j},||)\}$, et $t_j \in L_{q_{X_j}} = \{X_j\}$ si

$p_j = q_{X_j}$; nous avons $\cdot(X_1, t_2, \ldots, t_n) \in Post^*(X)$. Par conséquent, pour chaque terme $t_1 \in Post^*(L_{q_{X_1}})$ ($L_{q_{X_1}} = \{X_1\}$),

$$\cdot(t_1, t_2, \ldots, t_n) \in Post^*\big(\cdot\,(X_1, t_2, \ldots, t_n)\big) \subseteq Post^*(X) \subseteq Post^*(L_q).$$

(b) Il existe un indice $m > 1$ tel que $p_1 \cdots p_{m-1} \in Q^*_{\mathcal{R}}$, et

$$p_m \in \{(q_{X_m}, -), (q_{X_m}, ||)\}$$

Alors par la définition de S_s nous obtenons que $p_1 \cdots p_{m-1} = q_{X_1} \cdots q_{X_{m-1}} \in S^*_s(0)$, et que $p_1 \cdots p_n \in S^*_s(p_m \cdots p_n) \subseteq S^*_s(q_X)$. Et nous déduisons du cas précédent que pour chaque $t_m, \ldots, t_n$ tels que $t_j \in Post^*(L_{q_{X_j}})$ si $p_j \in \{(q_{X_j}, -), (q_{X_j}, ||)\}$, et $t_j \in L_{q_{X_j}} = \{X_j\}$ si $p_j = q_{X_j}$, nous avons $\cdot(t_m, \ldots, t_n) \in Post^*(X)$. Par conséquent, comme $X_1 \cdots X_{m-1} \in S^*_s(0)$, nous avons

$$\cdot(X_1, \ldots, X_{m-1}, t_m, \ldots, t_n) \in Post^*(X) \subseteq Post^*(L_q).$$

Ceci montre que la propriété est satisfaite puisque pour $j < m$, $L_{p_j} = L_{q_{X_j}} = \{X_j\}$.

2. $\cdot(L) \to (q, \cdot)$ est la conséquence d'une β_7-règle. Soient $q_1 \cdots q_n \in L$, nous allons montrer que pour chaque $t_1, \ldots, t_n$ tels que $t_j \in Post^*(L_{s_j})$ si $q_j \in \{(s_j, -), (s_j, P)\}$, et $t_j \in L_{q_j}$ si $q_j \in Q \cup Q_{\mathcal{R}}$, nous avons $\cdot(t_1, \ldots, t_n) \in Post^*(L_q)$.
Soit alors $\cdot(L') \to q$ une règle de Δ telle que $L = {S'_{s_{i-1}}}^*\big(\alpha'(L')\big)$, où α' est la substitution définie par

$$\alpha'(s) = \{s\} \cup \{q_X \mid X \xrightarrow{*}_{\Delta} s\},$$

et $S'_{s_{i-1}}$ est l'ensemble suivant de seq-règles : $S'_{s_{i-1}} = S_s \cup \{s \to q_X \mid X \xrightarrow{*}_{\widetilde{\Delta}_{i-1}} (s, -)\} \cup \{s \to 0 \mid 0 \xrightarrow{*}_{\widetilde{\Delta}_{i-1}} (s, 0)\}$.
Puisque $\cdot(L') \to q$ est une règle de Δ, il s'en suit que pour chaque $u_1, \ldots, u_g$ tels que $u_j \in L_{p_j}$ où $p_1 \cdots p_g \in \alpha'(L')$, nous avons $\cdot(u_1, \ldots, u_g) \in L_q$ (dû à la définition de α').
Supposons $q_1 \cdots q_n \in {S'_{s_{i-1}}}^k\big(\alpha'(L')\big)$. Nous procédons par induction sur k :
– $k = 0$. Alors $q_1 \cdots q_n \in \big(\alpha'(L')\big)$. Nous déduisons de la remarque précédente que pour chaque $u_1, \ldots, u_n$ t.q. $u_j \in L_{q_j}$, nous avons $\cdot(u_1, \ldots, u_n) \in L_q \subseteq Post^*(L_q)$, et la propriété est satisfaite.
– $k > 0$. Alors $q_1 \cdots q_n \in {S'_{s_{i-1}}}^k\big(\alpha'(L')\big)$. Soient $p_1, \ldots, p_h$ tels que $q_1 \cdots q_n \in S'_{s_{i-1}}(p_1 \cdots p_h)$ et $p_1 \cdots p_h \in {S'_{s_{i-1}}}^{k-1}\big(\alpha'(L')\big)$ Alors par induction sur k, nous déduisons que pour chaque $v_1, \ldots, v_h$ tels que $v_j \in Post^*(L_{s_j})$ si

$$p_j \in \{(s_j, -), (s_j, ||)\}$$

et $v_j \in L_{p_j}$ si $p_j \in Q \cup Q_{\mathcal{R}}$, nous avons

$$\cdot(v_1, \ldots, v_h) \in Post^*(L_q). \tag{6.1}$$

Considérons de tels termes v_j . Puisque $q_1 \cdots q_n \in S_s(p_1 \cdots p_h)$, La définition des règles de réécriture préfixe $S'_{s_{i-1}}$ implique qu'il y a plusieurs cas selon la règle de $S'_{s_{i-1}}$ qui a été appliquée :

(a) Il existe un état $s \in Q \cup Q_{\mathcal{R}}$ tel que $p_1 = s$ et $q_1 \cdots q_n$ est $(s, -)p_2 \cdots p_h$ ou $(s, ||)p_2 \cdots p_h$. Ceci veut dire que $n = h$, et $q_j = p_j$ pour $j > 1$ (une des règle $p_1 \to (p_1, -)$ ou $p_1 \to (p_1, ||)$ a été appliquée).

(b) Il existe un état $s \in Q \cup Q_{\mathcal{R}}$ et une variable X tels que $p_1 = s$ et $q_1 \cdots q_n$ est $q_X p_2 \cdots p_h$. Ceci veut dire que $X \xrightarrow{*}_{\widetilde{\Delta}_{i-1}} (s, -)$, $n = h$, et $q_j = p_j$ pour $j > 1$ (la règle $p_1 \to q_X$ a été appliquée).

(c) Il existe un état $s \in Q \cup Q_{\mathcal{R}}$ et une variable X tels que $p_1 = s$ et $q_1 \cdots q_n$ est $p_2 \cdots p_h$. Ceci veut dire que $0 \xrightarrow{*}_{\widetilde{\Delta}_{i-1}} (s, 0)$, $n = h - 1$, et $q_j = p_{j+1}$ pour $j \geq 1$ (la règle $p_1 \to 0$ a été appliquée).

(d) $p_1 \cdots p_m = q_{X_1} \cdots q_{X_m}$ pour des variables $X_1, \ldots, X_m$, et $q_1 \cdots q_n$ est $q_{Y_1} \cdots q_{Y_l} \cdot p_{m+1} \cdots p_h$ (c-à-d. $n = h - m + l$, $q_j = q_{Y_j}$, et $q_{l+j} = p_{m+j}$ pour $j \leq m$. Ce cas correspond à l'application d'une règle $q_{X_1} \cdots q_{X_m} \to q_{Y_1} \cdots q_{Y_l}.$),

Considérons par exemple les cas (a), (b), et (d), l'autre cas étant similaire :

– Le cas (a) : $q_1 \cdots q_n$ est $(p_1, -)p_2 \cdots p_h$ ou $(p_1, ||)p_2 \cdots p_h$, et $p_1 \in Q \cup Q_{\mathcal{R}}$. Soit alors $v'_1 \in Post^*(L_{p_1})$, et $v_1 \in L_{p_1}$ tels que $v'_1 \in Post^*(v_1)$. Alors de (6.1), nous obtenons que $\cdot(v_1, v_2 \ldots, v_h) \in Post^*(L_q)$, et donc

$$\cdot(v'_1, v_2 \ldots, v_h) \in Post^*(L_q).$$

– Le cas (b) : $q_1 \cdots q_n$ est $q_X p_2 \cdots p_h$ tel que $X \xrightarrow{*}_{\widetilde{\Delta}_{i-1}} (p_1, -)$, et $p_1 \in Q \cup Q_{\mathcal{R}}$. Ceci veut dire par induction sur i que $X \in Post^*(L_{p_1})$. Soit alors $v_1 \in L_{p_1}$ tel que $X \in Post^*(v_1)$. Alors à partir de (6.1), nous obtenons que $\cdot(v_1, v_2 \ldots, v_h) \in Post^*(L_q)$, et donc

$$\cdot(X, v_2 \ldots, v_h) \in Post^*(L_q).$$

– Le cas (d) : $p_1 \cdots p_m = q_{X_1} \cdots q_{X_m}$, il s'en suit que $v_1 = X_1, \ldots, v_m = X_m$, c-à-d. nous avons à partir de (6.1) que

$$\cdot(X_1, \ldots, X_m, v_{m+1}, \ldots, v_h) \in Post^*(L_q).$$

Puisque $q_{X_1} \cdots q_{X_m} \to q_{Y_1} \cdots q_{Y_l} \in S_s$, nous déduisons que $X_1 \cdots X_m \to Y_1 \cdots Y_l \in \mathcal{R}'_s$, et donc que $Y_1 \cdots Y_l \in Post^*(X_1 \cdots X_m)$. Par conséquent, $\cdot(Y_1, \ldots, Y_l, v_{m+1}, \ldots, v_h) \in Post^*\big(\cdot(X_1, \ldots, X_m, v_{m+1}, \ldots, v_h)\big) \subseteq Post^*(L_q)$. Ceci montre que la propriété est satisfaite.

$\star$ Pour la propriété (e), supposons que $||(\varphi) \to (q, ||) \in \widetilde{\Delta}_i \setminus \widetilde{\Delta}_{i-1}$. La preuve est similaire à la preuve précédente, et est même plus simple puisque c'est pas la peine d'imposer que les réécritures ont lieu en tête de pile. Les règles β_5 ont le même rôle que dans le cas du "$\cdot$", et les règles β_6 sont analogues aux règles β_7.

Nous montrons dans ce qui suit les propriétés (a), (b), et (c). La preuve est fastidieuse et il y a plusieurs cas à considérer. Dans ce qui suit, nous ne traitons que les

cas les plus intéressants. Soit q un état de $\widetilde{Q}$, et soit $v \xrightarrow{*}_{\widetilde{\Delta}_i} q$ une exécution de $\widetilde{\Delta}_i$ sur le terme v. Soit n le nombre de fois qu'une transition dans $\widetilde{\Delta}_i \setminus \widetilde{\Delta}_{i-1}$ est appliquée pendant cette exécution. Dans un tel cas, nous dénotons l'exécution précédente par $v \xrightarrow{*}{}^{n}_{\widetilde{\Delta}_i} q$. Nous procédons par induction sur n :

• $n = 0$, alors seules les règles de $\widetilde{\Delta}_{i-1}$ sont appliquées, et nous obtenons le résultat par induction sur i,

• $n > 0$. Il y a trois cas :

1. il existe un contexte C et un terme v' tels que $v = C[v']$, et

$$v = C[v'] \xrightarrow{*}{}^{n-1}_{\widetilde{\Delta}_i} C[s'] \rightarrow_{\widetilde{\Delta}_i} C[s] \xrightarrow{k}_{\widetilde{\Delta}_{i-1}} q \tag{6.2}$$

Nous montrons la propriété par induction sur k :

– $k = 0$. Alors C est le contexte trivial, $v = v'$, et $q = s$. Il y a différents cas selon la nature des états s et s'. Par exemple, s'il y a deux états p et p' dans $Q \cup Q_{\mathcal{R}}$ tels que $s' = (p', -)$, et $s = (p, -)$, alors nous obtenons par induction sur n que

$$v' \in Post^*(L_{p'}) \text{ et } v' \in Var \tag{6.3}$$

Alors le fait que $C[(p', -)] \xrightarrow{1}_{\widetilde{\Delta}_i} C[(p, -)]$ implique que $(p', -) \rightarrow_{\widetilde{\Delta}_i} (p, -)$ est une nouvelle règle, qui peut être soit une (β_3)-, ou une (β_4)-règle. Dans les deux cas, il est facile de montrer que $v' \in Post^*(L_p)$. Par exemple, supposons que $(p', -) \rightarrow_{\widetilde{\Delta}_i} (p, -)$ est une (β_3)-règle. Dans un tel cas, il y a dans $\widetilde{\Delta}_{i-1}$ une règle de la forme $\cdot(L) \rightarrow (p, \cdot)$ telle que $\{(p', -), (p', ||)\} \subseteq L$. Nous déduisons de la propriété (d) que pour chaque $t \in Post^*(L_{p'}) \cap Var$, $\cdot(t) \in Post^*(L_p)$, ce qui veut dire que $t \in Post^*(L_p)$. Ceci implique que $Post^*(L_{p'}) \cap Var \subseteq Post^*(L_p)$, et nous déduisons que $v \in Post^*(L_p)$. La propriété est donc satisfaite.

– $k > 0$. Il y a trois cas :

(a) $v \xrightarrow{*}_{\widetilde{\Delta}_i} C[s] \xrightarrow{k-1}_{\widetilde{\Delta}_{i-1}} q' \xrightarrow{1}_{\widetilde{\Delta}_{i-1}} q$. Supposons par exemple que $q' = (q'_0, ||)$ et $q = (q_0, ||)$, les autres cas sont similaires. Alors par induction sur k nous obtenons que $v \in Post^*(L_{q'_0})$, et $racine(v) = ||$. Donc, comme dans le cas précédent, puisque $(q'_0, ||) \xrightarrow{1}_{\widetilde{\Delta}_{i-1}} (q_0, ||)$ nous obtenons que $v \in Post^*(L_{q_0})$.

(b) $q = (q_0, \cdot)$, et il existe l termes $v_1, \ldots, v_l$ tels que $v = \cdot(v_1, \ldots, v_l)$. Puisque $s \notin Q \cup Q_{\mathcal{R}}$ $(n > 0)$, supposons que $s = (p, -)$ ou $s = (p, ||)$, pour un $p \in Q$. Soient alors un contexte C' tel que $C[x] = \cdot(v_1, \ldots, v_{m-1}, C'[x], v_{m+1}, \ldots, v_l)$, et l états $q_1, \ldots, q_l \in \widetilde{Q}$ tels que

$$v = \cdot(v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}_i} C[s] = \cdot(v_1, \ldots, v_{m-1}, C'[s], v_{m+1}, \ldots, v_l) \xrightarrow{k-1}_{\widetilde{\Delta}_{i-1}}$$

$$\cdot(q_1, \ldots, q_l) \xrightarrow{1}_{\widetilde{\Delta}_{i-1}} (q_0, \cdot).$$

Soient $p_1, \ldots, p_l \in Q \cup Q_{\mathcal{R}}$ tels que $q_j = p_j$ ou $q_j \in \{(p_j, -), (p_j, ||)\}$. Alors par induction sur k nous obtenons que $v_m \in Post^*(L_{p_m})$, et $racine(v_m) \neq \cdot$. En plus, par induction sur i, nous déduisons que pour $j \neq m$, $v_j \in L_{p_j}$

si $p_j \in Q_{||} \cup Q_-$, et $v_j \in Post^*(L_{p_j})$ sinon, et $racine(v_j) \neq \cdot$. Alors la propriété est satisfaite grâce à la propriété (d), puisque nous obtenons que $v \in Post^*(L_{q_0})$.

(c) $q = (q_0, ||)$, et il existe l termes $v_1, \ldots, v_l$ tels que $v = ||(v_1, \ldots, v_l)$, un contexte C' tel que

$$C[x] = ||(v_1, \ldots, v_{m-1}, C'[x], v_{m+1}, \ldots, v_l),$$

l états $q_1, \ldots, q_l$ dans Q' tels que

$$v = ||(v_1, \ldots, v_l) \stackrel{*}{\rightarrow}_{\widetilde{\Delta}_i} C[s] = ||(v_1, \ldots, v_{m-1}, C'[s], v_{m+1}, \ldots, v_l) \stackrel{k-1}{\rightarrow}_{\widetilde{\Delta}_{i-1}}$$

$$||(q_1, \ldots, q_l) \stackrel{1}{\rightarrow}_{\widetilde{\Delta}_{i-1}} (q_0, ||)$$

Ce cas est similaire au cas précédent. Nous utilisons la propriété (e) au lieu de la propriété (d).

2. Il existe un contexte C et un opérateur $f \in \{\cdot, ||\}$ tels que

$$v = C[f(v_1, \ldots, v_l)] \stackrel{*}{\rightarrow}^{n-1}_{\widetilde{\Delta}_i} C[f(p_1, \ldots, p_l)] \rightarrow_{\widetilde{\Delta}_i} C[s] \stackrel{k}{\rightarrow}_{\widetilde{\Delta}_{i-1}} q$$

tels qu'il existe $n_1 < n, \ldots, n_l < n$ tels que $\sum_{1 \leq i \leq l} n_i = n - 1$, et

$$v_j \stackrel{*}{\rightarrow}^{n_j}_{\widetilde{\Delta}_i} p_j$$

Soit $p_j = (q_j, \sharp)$, où $\sharp \in \{-, \cdot, ||\}$. Alors nécessairement par induction sur n, il s'en suit que pour chaque j, $1 \leq j \leq l$,

$$v_j \in Post^*(L_{q_j}), \tag{6.4}$$

Soit $s = (p, \sharp)$. Selon la nature de l'opérateur f, nous pouvons utiliser les propriétés (d) ou (e) pour montrer que $f(v_1, \ldots, v_m) \in Post^*(L_{q_0})$, où $q \in \{(q_0, -), (q_0, ||), (q_0, \cdot)\}$. Alors nous pouvons montrer comme précédemment par induction sur k que $v \in Post^*(L_q)$.

3. Il existe un contexte C tel que $v = C[v_1, \ldots, v_m]$, et

$$v = C[v_1, \ldots, v_m] \stackrel{*}{\rightarrow}^{n}_{\widetilde{\Delta}_i} C[p_1, \ldots, p_m] \stackrel{k}{\rightarrow}_{\widetilde{\Delta}_{i-1}} q$$

tel qu'il existe $n_1 < n, \ldots, n_m < n$ tels que $\sum_{1 \leq i \leq m} n_i = n$, et

$$v_j \stackrel{*}{\rightarrow}^{n_j}_{\widetilde{\Delta}_i} p_j$$

Soit $p_j = (q_j, \sharp)$. Alors nécessairement, nous déduisons par induction sur n que pour chaque j, $1 \leq j \leq m$,

$$v_j \in Post^*(L_{q_j}), \tag{6.5}$$

Alors nous pouvons montrer comme précédemment par induction sur k que $v \in Post^*(L_{q_0})$, où $q \in \{(q_0, -), (q_0, ||), (q_0, \cdot)\}$.

□

Lemme 6.5.2 *Pour chaque $q \in Q \cup Q_{\mathcal{R}}$, et chaque $v \in T_p$, nous avons :*
- *$v \in Post^*(L_q)$, et $racine(v) = ||$, alors $v \in L_{(q,||)}$,*
- *$v \in Post^*(L_q)$, et $racine(v) = \cdot$, alors $v \in L_{(q,\cdot)}$,*
- *$v \in Post^*(L_q) \cap Var$, alors $v \in L_{(q,-)}$,*
- *$v \in Post^*(L_q)$, et $v = 0$, alors $v \in L_{(q,0)}$.*

Preuve :

Nous allons montrer par induction sur k que :
- $v \in Post^k(L_q)$, et $racine(v) = ||$, alors $v \xrightarrow{*}_{\widetilde{\Delta}} (q, ||)$,
- $v \in Post^k(L_q)$, et $racine(v) = \cdot$, alors $v \xrightarrow{*}_{\widetilde{\Delta}} (q, \cdot)$,
- $v \in Post^k(L_q) \cap Var$, alors $v \xrightarrow{*}_{\widetilde{\Delta}} (q, -)$,
- $v \in Post^k(L_q)$, et $v = 0$, alors $v \xrightarrow{*}_{\widetilde{\Delta}} (q, 0)$.

Nous commençons l'induction :
- $k = 0$. Alors la propriété est satisfaite grâce aux règles (β_1).
- $k > 0$. Soit alors $v \in Post^k(L_q)$, et $v' \in Post^{k-1}(L_q)$ tels que $v \in Post(v')$. Supposons que $racine(v) = ||$, les autres cas peuvent être traités de la même manière. Alors il y a trois cas :

  1. $v' \in Var$, et v est obtenu à partir de v' en appliquant une par-règle. Soit alors X une variable telle que $v' = X$. Alors nous obtenons par induction que $X \in L_{(q,-)}$. Les règles $(\beta_5 a)$ impliquent que $||(\varphi) \rightarrow (q, ||)$ est une règle de $\widetilde{\Delta}$, où $\varphi = S_p^*(q_X)$. Puisque v est obtenu à partir de X en appliquant une règle de $\mathcal{R}_p$, nous déduisons que $v = ||(X_1, \ldots, X_n)$ où $Parikh(q_{X_1} \cdots q_{X_n}) \models S_p(q_X)$, c-à-d. $Parikh(q_{X_1} \cdots q_{X_n}) \models \varphi$. Par conséquent, nous avons :
  $$v = ||(X_1, \ldots, X_n) \xrightarrow{*}_{\widetilde{\Delta}} ||(q_{X_1}, \ldots, q_{X_n}) \rightarrow_{\widetilde{\Delta}} (q, ||)$$
  2. $v' = 0$, et v est obtenu à partir de v' en appliquant une par-règle. Alors par induction nous obtenons que $0 \in L_{(q,0)}$. Les règles $(\beta_5 b)$ impliquent que $||(\varphi) \rightarrow (q, ||)$, où $\varphi = S_p^*(0)$. Puisque v est obtenu à patir de 0 en appliquant une règle de $\mathcal{R}_p$, nous déduisons que $v = ||(X_1, \ldots, X_n)$ où $Parikh(q_{X_1} \cdots q_{X_n}) \models S_p(0)$, c-à-d. $Parikh(q_{X_1} \cdots q_{X_n}) \models \varphi$. Par conséquent, nous obtenons comme précédemment que $v \in L_{(q,||)}$.
  3. $racine(v') = ||$. Alors par induction, nous déduisons que $v' \in L_{(q,||)}$. Nous voulons montrer que v est aussi dans $L_{(q,||)}$. Il y a plusieurs cas selon le type de la règle qui a été appliquée :
     - Une par-règle de la forme
     $$X_1 || \cdots || X_n \rightarrow X'_1 || \cdots || X'_m$$
     a été appliquée, et a réécrit $X_1, \ldots, X_n$, en $X'_1, \ldots, X'_m$. Alors, il existe un contexte C tel que
     $$v' = C[||(X_1, \ldots, X_n, v_1, \ldots, v_l)]$$

et

$$v = C[||(X'_1, \ldots, X'_m, v_1, \ldots, v_l)]$$

Il y a deux cas, qui dépendent des valeurs de m et l :

(a) $m > 1$ ou $l > 1$. Soit alors $p_1, \ldots, p_n$, $p'_1, \ldots, p'_l \in \widetilde{Q}$, et $p \in Q \cup Q_{\mathcal{R}}$ tels que

$$||(X_1, \ldots, X_n, v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}} ||(p_1, \ldots, p_n, p'_1, \ldots, p'_l) \rightarrow_{\widetilde{\Delta}} p$$

ou

$$||(X_1, \ldots, X_n, v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}} ||(p_1, \ldots, p_n, p'_1, \ldots, p'_l) \rightarrow_{\widetilde{\Delta}} (p, ||)$$

• Supposons que nous avons le second cas. Ceci veut dire que

$$v' = C[||(X_1, \ldots, X_n, v_1, \ldots, v_l)] \xrightarrow{*}_{\widetilde{\Delta}} C[(p, ||)] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||)$$

Soit alors φ une formule telle que $||(\varphi) \rightarrow (p, ||)$ est une règle de $\widetilde{\Delta}$, et

$$Parikh(p_1 \cdots p_n p'_1 \cdots p'_l) \models \varphi.$$

Nous déduisons du lemme 6.5.3 que $Parikh(q_{X'_1} \cdots q_{X'_m} p'_1 \cdots p'_l) \models \varphi$, et donc

$$||(X'_1, \ldots, X'_m, v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}} ||(q_{X'_1}, \ldots, q_{X'_m}, p'_1, \ldots, p'_l) \rightarrow_{\widetilde{\Delta}} (p, ||)$$

ce qui veut dire que

$$v = C[||(X'_1, \ldots, X'_m, v_1, \ldots, v_l)] \xrightarrow{*}_{\widetilde{\Delta}} C[(p, ||)] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||).$$

• Supposons que

$$||(X_1, \ldots, X_n, v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}} ||(p_1, \ldots, p_n, p'_1, \ldots, p'_l) \rightarrow_{\widetilde{\Delta}} p$$

Alors nous obtenons aussi de la même manière que

$$v = C[||(X'_1, \ldots, X'_m, v_1, \ldots, v_l)] \xrightarrow{*}_{\widetilde{\Delta}} C[(p, ||)]$$

et nous pouvons alors montrer par induction structurelle sur C que

$$C[(p, ||)] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||).$$

(b) $m = 1$ et $l = 0$, ou $m = 0$ et $l = 1$, ce qui veut dire que la paral-règle qui a été appliquée est de le forme $X_1|| \cdots ||X_n \rightarrow X$ (ou $X_1|| \cdots ||X_n \rightarrow 0$). Considérons par exemple le premier cas. Soit alors C' le contexte tel que

$$v' = C'[\cdot(||(X_1, \ldots, X_n), u_1, \ldots, u_j)]$$

et

$$v = C'[\cdot(X, u_1, \ldots, u_j)]$$

Soit $v' = C'[\cdot(|||(X_1, \ldots, X_n), u_1, \ldots, u_j)] \xrightarrow{*}_{\widetilde{\Delta}} C'[\cdot(\tilde{p}, q_1, \ldots, q_j))] \xrightarrow{*}_{\widetilde{\Delta}} C'[\tilde{q'}] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||)$, où $\tilde{p} \in \{p, (p, ||)\}$ pour un $p \in Q \cup Q_{\mathcal{R}}$, et $\tilde{q'} \in \{q', (q', \cdot)\}$ pour un $q' \in Q \cup Q_{\mathcal{R}}$.
$\star$ Supposons que $\tilde{q'} = (q', \cdot)$. Comme précédemment, considérons la règle de $\widetilde{\Delta}$, $||(\varphi) \to (p, ||)$ telle que

$$Parikh(q_{X_1} \cdots q_{X_n}) \models \varphi.$$

Puisque $q_X \in S_p^*(|||(q_{X_1}, \ldots, q_{X_n}))$, nous déduisons du lemme 6.5.3 que le vecteur $\vec{u}_{q_X}$ satisfait φ, et donc, nous déduisons à partir des règles (β_4) que $(q_X, -) \to (p, -)$ est dans $\widetilde{\Delta}$, et donc que

$$X \to_{\widetilde{\Delta}} q_X \to_{\widetilde{\Delta}} (q_X, -) \to_{\widetilde{\Delta}} (p, -)$$

En plus, soit $\cdot(L) \to (q', \cdot)$ une règle de $\widetilde{\Delta}$ telle que $\tilde{p}q_1 \cdots q_j \in L$. Alors forcément, nous avons que $(p, -)q_1 \cdots q_j \in L$, ce qui veut dire que $v = C'[\cdot(X, u_1, \ldots, u_j)] \xrightarrow{*}_{\widetilde{\Delta}} C'[\cdot((p, -), q_1, \ldots, q_j)] \to_{\widetilde{\Delta}} C'[(q', \cdot)] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||)$
$\star$ Si $\tilde{q'} = q'$, nous montrons comme précedemment que

$$v = C'[\cdot(X, u_1, \ldots, u_j)] \xrightarrow{*}_{\widetilde{\Delta}} C'[(q', \cdot)]$$

et ensuite, nous montrons par induction structurelle sur C' que

$$C'[(q', \cdot)] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||)$$

– Une seq-règle de la forme

$$X \to X_1 \cdots X_m$$

a été appliquée à un terme de la forme

$$v' = C[|||(X, v_1, \ldots, v_l)]$$

et a réécrit X en $\cdot(X_1, \ldots, X_m)$ pour produire le terme

$$v = C[|||\big(\cdot(X_1, \ldots, X_m), v_1, \ldots, v_l\big)].$$

Supposons comme précédemment qu'il existe des états $p_1, p'_1, \ldots, p'_l$, et un état $p \in Q \cup Q_{\mathcal{R}}$ tels que

$$||(X, v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}} ||(p_1, p'_1, \ldots, p'_l) \to_{\widetilde{\Delta}} (p, ||)^4$$

où $p_1 = (p', -)$ ou $p_1 = p'$ pour un état p' dans $Q \cup Q_{\mathcal{R}}$.

[4]Comme expiqué précédemment, le cas où $||(X, v_1, \ldots, v_l) \xrightarrow{*}_{\widetilde{\Delta}} p$ peut être résolu de la même manière en considérant en plus une induction structurelle sur C.

Alors puisque $X \xrightarrow{*}_{\widetilde{\Delta}} (p', -)$, nous déduisons à partir des règles (β_5) que

$$\cdot(q_{X_1}, \ldots, q_{X_m}) \xrightarrow{*}_{\widetilde{\Delta}} (p', \cdot)$$

puisque $q_{X_1} \cdots q_{X_n} \in S_s^*(q_X)$.
En plus, soit $||(\varphi) \to (p, ||)$ une règle de $\widetilde{\Delta}$ telle que

$$Parikh(p_1 p'_1 \cdots p'_l) \models \varphi.$$

Alors les règles (β_6) impliquent que $Parikh((p', \cdot)p'_1 \cdots p'_l) \models \varphi$, ce qui veut dire que
$v = C[||\big(\cdot(X_1, \ldots, X_m), v_1, \ldots, v_l\big)] \xrightarrow{*}_{\widetilde{\Delta}} C[||\big(\cdot(q_{X_1}, \ldots, q_{X_m}), p'_1, \ldots, p'_l\big)] \xrightarrow{*}_{\widetilde{\Delta}}$
$C[||((p', \cdot), p'_1, \ldots, p'_l)] \to_{\widetilde{\Delta}} C[(p, ||)] \xrightarrow{*}_{\widetilde{\Delta}} (q, ||)$.

– Les autres cas où

$$v = C[\cdot(X_1, \ldots, X_n, v_1, \ldots, v_l)]$$

et

$$v' = C[\cdot(X'_1, \ldots, X'_m, v_1, \ldots, v_l)]$$

ou

$$v = C[\cdot(X, v_1, \ldots, v_l)]$$

et

$$v' = C[\cdot\big(||(X_1, \ldots, X_n), v_1, \ldots, v_l\big)]$$

peuvent être traités de manière similaire puisque les règles d'inférence pour l'opérateur séquentiel sont analogues aux règles pour l'opérateur parallèle. Dans ce cas, nous utiliserons la première partie du lemme 6.5.3 à la place de la deuxième partie utilisée jusqu'à présent.

□

Lemme 6.5.3
• *Soit* $\cdot(X_1, \ldots, X_m, u_1, \ldots, u_n) \xrightarrow{*}_{\widetilde{\Delta}} \cdot(p_1, \ldots, p_m, p'_1, \ldots, p'_n)$, *où*

$$p_1 \cdots p_m p'_1 \cdots p'_n \in L$$

pour un L *tel que* $\cdot(L) \to (q, \cdot)$ *soit un règle de* $\widetilde{\Delta}$. *Si* $X_1 \cdots X_m \to Y_1 \cdots Y_k \in \mathcal{R}_s$, *alors* $q_{Y_1} \cdots q_{Y_k} p'_1 \cdots p'_n \in L'$.
• *Soit* $||(X_1, \ldots, X_m, u_1, \ldots, u_n) \xrightarrow{*}_{\widetilde{\Delta}} ||(p_1, \ldots, p_m, p'_1, \ldots, p'_n)$, *où*

$$Parikh(p_1 \cdots p_m p'_1 \cdots p'_n) \models \varphi$$

pour un φ *tel que* $||(\varphi) \to (q, ||)$ *soit une règle de* $\widetilde{\Delta}$. *Si* $X_1 || \cdots || X_m \to Y_1 || \cdots || Y_k \in \mathcal{R}_p$, *alors* $Parikh(q_{Y_1} \cdots q_{Y_k} p'_1 \cdots p'_n) \models \varphi$.

Preuve :

Nous ne montrons que le premier point, le second peut se montrer de la même manière.

Il y a deux cas qui dépendent de comment la règle $\cdot(L) \to (q, \cdot)$ a été créée. Considérons par exemple le cas où cette règle est le résultat d'une (β_5)-règle. Soit alors $A \in Var$ tel que $A \xrightarrow{*}_{\widetilde{\Delta}} (q, -)$, et $L = S_s^*(q_A)$. Nous montrons que $q_{Y_1} \cdots q_{Y_k} p'_1 \cdots p'_n \in L$. Nous procédons par induction sur le nombre K d'indices i dans $\{1, \ldots, m\}$ tels que $p_i \neq q_{X_i}$:

1. $K = 0$, alors $p_i = q_{X_i}$ pour chaque i, $1 \leq i \leq m$. Alors puisque $q_{Y_1} \cdots q_{Y_k} \in S_s(q_{X_1} \cdots q_{X_m})$, il est clair à partir de la définition de L que $q_{Y_1} \cdots q_{Y_k} p'_1 \cdots p'_n \in L$.
2. $K > 0$. Soit alors i un indice tel que $p_i = (q_{F_i}, -)$ pour une variable F_i, et pour chaque j, $1 \leq j < i$, $p_j = q_{X_j}$. Alors nécessairement, dû aux règles de réécriture préfixe S_s, nous déduisons que $(q_{F_i}, -) p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L$, et $q_{X_1} \cdots q_{X_{i-1}} \in S_s^*(0)$. Par conséquent, nous obtenons à partir de la définition de S_s que $q_{F_i} p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L$. En plus, puisque $X_i \xrightarrow{*}_{\widetilde{\Delta}} (q_{F_i}, -)$, le lemme (6.5.1) implique que $X_i \in Post^*(F_i)$, c-à-d. par la définition de $\mathcal{R}'_s$ et $\mathcal{R}'_p$ que $X_i \in \mathcal{R}'^*_s(F_i)$ (et $X_i \in \mathcal{R}'^*_p(F_i)$). Il s'en suit que $q_{X_i} \in S_s{}^*(q_{F_i})$ (notons qu'il est important ici d'avoir $\mathcal{R}'_s$ dans la définition de S_s, $\mathcal{R}_s$ ne suffit pas puisque X_i peut être obtenu à partir de F_i après l'application de différentes règles de $\mathcal{R}_s$ et $\mathcal{R}_p$), et par conséquent que $q_{X_i} p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L$. Puisque $q_{X_1} \cdots q_{X_{i-1}} \in S_s^*(0)$, nous obtenons que

$$q_{X_1} \cdots q_{X_{i-1}} q_{X_i} p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L,$$

Puisque le nombre des états $p_j \neq q_{X_j}$ (de la forme $(s, -)$) dans

$$q_{X_1} \cdots q_{X_{i-1}} q_{X_i} p_{i+1} \cdots p_m$$

est inférieur à K, nous déduisons par induction que $q_{Y_1} \cdots q_{Y_k} p'_1 \cdots p'_n \in L$.

Considérons le cas où la règle $\cdot(L) \to (q, \cdot)$ est le résultat d'une (β_7)-règle. Nous procédons exactement comme précédemment. La seule différence est que maintenant, dans l'étape d'induction, nous devons consiérer aussi le cas où il y a un indice i tel que $p_i = (p, -)$ pour un p dans Q, et pour chaque j, $1 \leq j < i$, $p_j = q_{X_j}$. Alors nécessairement, dû aux règles de réécriture préfixe S'_s, nous déduisons que $(p, -) p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L$, et $q_{X_1} \cdots q_{X_{i-1}} \in S_s^*(0)$. Par conséquent, à partir de la définition de S'_s, nous obtenons que $p p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L$. En plus, puisque $X_i \xrightarrow{*}_{\widetilde{\Delta}} (p, -)$, la définition de S'_s implique que $q_{X_i} \in S'^*_s(p)$, et par conséquent, que $q_{X_i} p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L$. Puisque $q_{X_1} \cdots q_{X_{i-1}} \in S_s^*(0)$, nous obtenons que

$$q_{X_1} \cdots q_{X_{i-1}} q_{X_i} p_{i+1} \cdots p_m p'_1 \cdots p'_n \in L,$$

Puisque le nombre des états $p_j \neq q_{X_j}$ (de la forme $(s, -)$) dans $q_{X_1} \cdots q_{X_{i-1}} q_{X_i} p_{i+1} \cdots p_m$ est inférieur à K, nous déduisons par induction que $q_{Y_1} \cdots q_{Y_k} p'_1 \cdots p'_n \in L$. □

6.6 Conclusion

Dans ce chapitre, nous avons représenté les termes de processsus par des arbres à largeurs non bornés. Nous avons défini une classe d'automates d'arbres pour représenter des ensembles d'arbres à arités non bornées qui sont fermés par permutations des fils de certains nœuds (ceux correspondant à l'opérateur "||"). Nous avons montré que cette classe d'automates est effectivement fermée par toutes les opérations booléennes et que son problème du vide est décidable. D'autres classes d'automates similaires aux nôtres ont été indépendamment considérées par d'autres auteurs [Col02, Lug03]. Les "multiset automata" de [Col02] définissent les mêmes ensembles que les nôtres, alors que les "multitree automata" de [Lug03] définissent une classe plus générale qui permet de compter le nombre de fils égaux.

Nous avons utilisé cette classe d'automates pour calculer les ensembles des accessibles des systèmes PRS. Notre approche constitue un cadre générique où toutes les techniques d'analyse symbolique des réseaux de Petri et des automates à compteurs utilisant des contraintes linéaires peuvent être intégrées, en combinaison avec les techniques d'analyse d'accessibilité symboliques des systèmes à pile. En plus, notre algorithme permet de dériver des constructions effectives pour des sous-classes significatives de PRS. En particulier, il permet de calculer les ensembles des accessibles pour la classe PAD, et de décider la satisfaisabilité des formules EF pour cette classe. Comme déjà mentionné auparavant, ce résultat étend toutes les techniques d'analyse des systèmes à pile [Cau92, BEM97, FWW97] et des systèmes PA [LS98, EP00] au cadre plus général des systèmes PAD. Cette classe est importante puisqu'elle permet de modéliser les programmes récursifs parallèles où les procédures peuvent communiquer.

L'algorithme présenté dans ce chapitre a plusieurs avantages par rapport à la technique du chapitre précédent qui a aussi comme conséquence de pouvoir résoudre les problèmes d'accessibilité (modulo toutes les équivalences) pour les systèmes PAD :

- cette technique permet *toujours* de résoudre le problème d'accessibilité pour les PAD, alors que l'approche précédente ne peut être appliquée que si le langage $\mathcal{L}'$ est $\sim$-compatible ;
- elle permet le model-checking des formules EF pour les systèmes PAD, alors que les algorithmes du chapitre précédent ne le permettent pas ;
- cette technique peut s'appliquer à des classes plus générales que les systèmes PAD, et permet d'unifier, d'intégrer, et d'exploiter tous les travaux existants d'analyse des réseaux de Petri effectivement semilinéaires et des systèmes à pile, pour l'analyse exacte de différentes sous-classes de PRS. Elle offre également un cadre uniforme qui permet de combiner toutes les techniques existantes d'analyse des réseaux de Petri et des automates à compteurs pour calculer des sur-approximations des ensembles des accessibles pour toute la classe PRS [EM96, CH78, DBR01, ABS01, FL02].

Cependant, l'avantage majeur de l'autre approche est qu'elle est polynômiale au meilleur des cas, et qu'au pire des cas elle risque d'être 1-exponentielle ; alors que l'algorithme présenté dans ce chapitre est, pour les PAD, au meilleur des cas 1-exponentiel si les formules des règles de l'automate initial sont de la forme $\exists^* \phi$ où ϕ est sans quantificateurs ; et dans le pire des cas, elle peut être 4-exponentielle. C'est la complexité

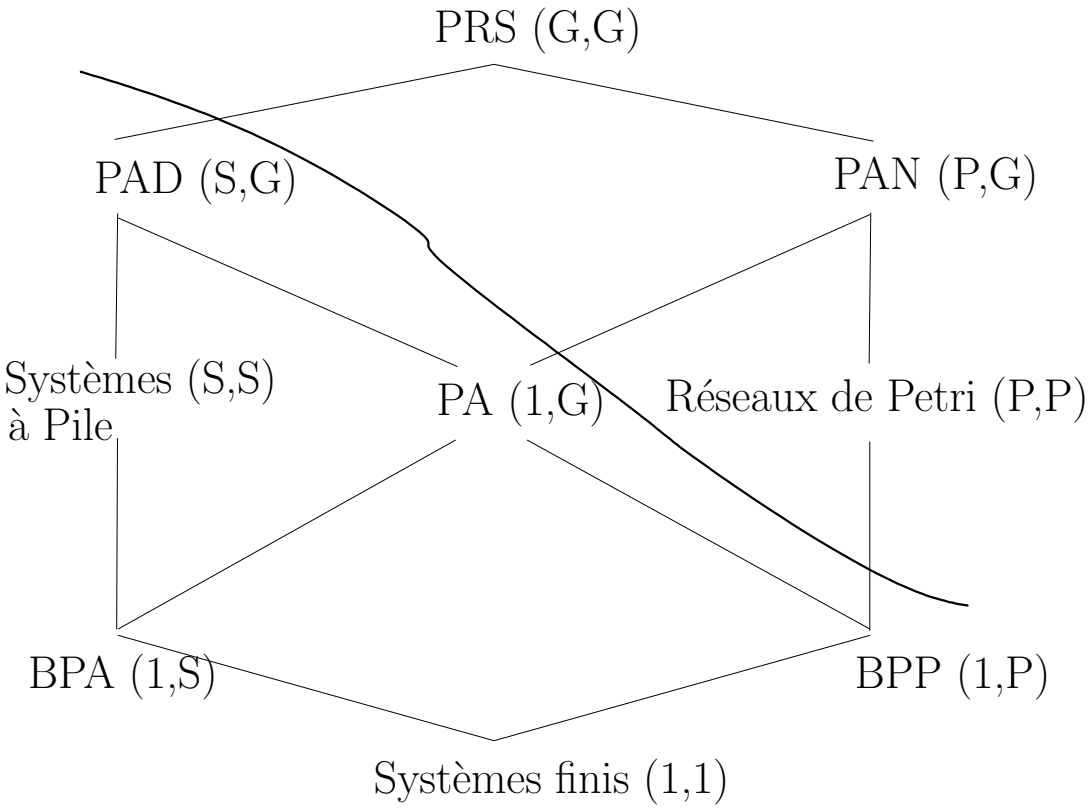


FIG. 6.3 – Limites de la décidabilité du model-checking de EF

théorique de notre procédure ; mais dans la pratique, nous pouvons utiliser les techniques d'analyses symboliques des automates à compteurs et des réseaux de Petri qui se comportent bien.

Finalement, notons que Mayr [May98] a montré que PAD est la classe la plus générale de la hiéarchie de PRS pour laquelle le model-checking des formules EF est décidable. La figure 6.3 montre les limites de décidabilité de EF pour cette hiérarchie [May98]. Le model-checking de EF est décidable pour toutes les classes qui sont en dessous de la ligne, et indécidable pour toutes celles en dessus. Nous en déduisons qu'il n'est pas possible, pour les classes en dessus de la ligne, de représenter les ensembles des prédécesseurs par des CH-automates. Notre construction pour PAD est donc "optimale" pour cette hiérarchie.

Troisième partie

Analyse des programmes récursifs parallèles par calcul d'abstractions des chemins d'exécutions

Introduction

Dans la partie précédente, nous avons modélisé les programmes parallèles récursifs par des PRS. Ce formalisme est certes très puissant mais ne permet de représenter la synchronisation de manière précise que dans certains cas. L'avantage de ce modèle est qu'il est possible de l'analyser de manière exacte dans plusieurs cas intéressants. Dans cette partie, nous allons utiliser des modèles plus puissants pour représenter de manière plus précise les programmes. Ces formalismes permettent de modéliser récursion et synchronisation de manière exacte. Plus précisément, nous considérons dans cette partie deux classes de programmes : (1) les programmes contenant un nombre fixe de processus séquentiels concurrents. Nous modélisons ces programmes par des systèmes à pile communicants ; et (2) les programmes où dynamisme et récursion sont permis. Nous modélisons ces programmes par des systèmes de réécriture qui étendent PA "à-la CCS" par des opérateurs de synchronisation et de restriction.

Comme l'analyse des programmes devient indécidable dès que l'on considère la synchronisation et les appels récursifs de procédures [Ram00], les deux modèles que nous considérons sont indécidables. Nous ne pourrons donc pas résoudre le problème d'accessibilité de manière exacte dans ces cas. Pour contourner ce problème, nous proposons une approche générale qui permet de résoudre ce problème d'accessibilité de manière abstraite.

Rappelons que le problème auquel nous nous intéressons est de savoir si

$$Post^*(\mathcal{L}) \cap \mathcal{L}' = \emptyset$$

ou de manière équivalente si

$$Pre^*(\mathcal{L}') \cap \mathcal{L} = \emptyset$$

où $\mathcal{L}$ et $\mathcal{L}'$ sont des ensembles infinis de configurations. Ce problème consiste à déterminer s'il existe une configuration de $\mathcal{L}$ qui peut atteindre une configuration de $\mathcal{L}'$. Pour le résoudre, nous pouvons soit (1) calculer, si possible, l'ensemble des accessibles $Post^*(\mathcal{L})$ et vérifier que son intersection avec $\mathcal{L}'$ est vide (ou de manière équivalente, calculer $Pre^*(\mathcal{L}')$ et vérifier qu'il n'intersecte pas $\mathcal{L}$). C'est l'approche que nous avons considéré jusqu'à présent. Soit (2) calculer l'ensemble des séquences d'actions $Paths(\mathcal{L}, \mathcal{L}')$ qui permettent d'accéder à une configuration de $\mathcal{L}'$ à partir d'une configuration de $\mathcal{L}$, et vérifier s'il est vide :

$$Paths(\mathcal{L}, \mathcal{L}') = \emptyset$$

Comme les modèles considérés sont indécidables, nous ne pouvons ni calculer l'ensemble des accessibles, ni celui des chemins d'exécutions de manière exacte. Nous

adoptons alors une approche basée sur le calcul d'une abstraction $\alpha\big(Paths(\mathcal{L}, \mathcal{L}')\big)$ qui constitue une sur-approximation de l'ensemble des chemins d'exécutions qui mènent de $\mathcal{L}$ à $\mathcal{L}'$. Si cette sur-approximation est vide, nous sommes sûrs que $Paths(\mathcal{L}, \mathcal{L}')$ l'est aussi.

Nous proposons dans cette partie un cadre algébrique générique, basé sur les algèbres de Kleene, pour le calcul de ces abstractions (sur-approximations). Notre méthode est basée sur la combinaison des techniques d'analyse d'accessibilité symboliques basées sur les automates avec des algorithmes de résolution de systèmes de contraintes polynômiales dans des domaines abstraits. Ce cadre peut être instancié par différentes classes d'abstraction. Ceci permet d'avoir différents algorithmes d'analyse ayant différents coûts en fonction de la précision de l'abstraction considérée. Nous considérons dans ce travail deux classes d'abstractions : (1) les abstractions à chaînes finies, qui sont des abstractions dont le domaine ne contient pas de chaînes infinies. Dans ce cas, le système de contraintes peut être résolu par un calcul itératif de point fixe. (2) Les abstractions commutatives qui sont des abstractions qui "oublient" l'ordre entre les différentes actions exécutées. Ces abstractions correspondent aux classes de langages qui contiennent un mot ssi ils contiennent toutes ses permutations. Dans ce cas, nous résolvons le système de contraintes en utilisant la procédure de [HK99]. Nous donnons plusieurs exemples d'abstractions utiles qui permettent de faire des analyses à différents coûts et différentes précisions.

Chapitre 7

Programmes récursifs communicants

Nous considérons dans ce chapitre les programmes ayant un nombre fixe de processus séquentiels concurrents qui tournent en parallèle et qui peuvent se synchroniser. Chaque processus séquentiel pouvant avoir des variables globales, et chaque procédure pouvant aussi avoir des variables locales. Nous modélisons ces programmes par des *automates à pile communicants*. Contrairement à PRS, ce modèle permet de représenter de manière précise récursion *et* synchronisation ; mais ne permet pas de considérer le dynamisme. Comme expliqué dans l'introducion de cette partie, puisque ces modèles sont indécidables, nous résolvons leur problème d'accessibilité de manière abstraite en calculant des abstractions des chemins d'exécution.

Ce chapitre est organisé comme suit. La section suivante explique plus en détails le principe de l'approche considérée dans ce chapitre. La section 7.2 présente le modèle que nous considérons qui est basé sur les automates à pile communicants. La section 7.3 donne des exemples d'abstractions et définit les classes d'abstractions que nous considérons. La section 7.4 montre comment réduire le problème de calculer des langages de chemins abstraits au calcul de certains ensembles de prédecesseurs/successeurs. La section 7.5 présente un algorithme générique pour le calcul des prédécesseurs et donne sa complexité. Un algorithme similaire qui calcule les successeurs est décrit à la section 7.6.

Le contenu de ce chapitre a été publié dans [BET03a, BET03b]

7.1 Le principe de l'approche

Considérons un programme séquentiel pouvant avoir des procédures récursives, et soit c une configuration de ce programme. Comme mentionné à la section 4.3.2, et comme nous allons le décrire avec plus de détails dans la section suivante, un tel programme séquentiel peut être modélisé par un automate à pile. Une configuration du programme peut donc être représentée par une paire comprenant un point de contrôle et une pile contenant l'information sur les procédures qui ont été appelées mais dont

l'exécution n'est pas encore terminée. Si nous considérons un chemin d'exécution possible du programme comme une séquence d'instructions, nous pouvons identifier l'ensemble de tous les chemins du programme qui mènent à c à partir d'une configuration initiale c_0 par un langage hors-contexte $Paths(c_0, c)$ (et chaque langage hors-contexte peut être obtenu de cette manière), puisque les langages définis par les automates à pile définissent exactement les langages hors-contextes [HU79]. Par exemple, prenons un programme qui soit termine immédiatement soit exécute une instruction a, s'appelle récursivement, exécute une instruction b, et termine. Soit c une configuration correspondant à la terminaison. Nous avons $Paths(c_0, c) = \{a^n b^n \mid n \geq 0\}$.

Considérons maintenant un système concurrent ayant deux programmes séquentiels communiquant par rendez-vous comme décrit à la section 4.1. Nous supposons sans perte de généralité que : **(A)** l'ensemble des signaux Sig est l'union disjointe de deux ensembles de signaux : les signaux pouvant être envoyés du processus un au processus deux, et les signaux pouvant être envoyés du processus deux au processus un. Nous pouvons toujours nous ramener à ce cas de figure par un renommage des signaux. Soit (c_1, c_2) une configuration du système, où c_1 (resp. c_2) est une configuration du processus 1 (resp. du processus 2). Le problème est de décider si (c_1, c_2) est accessible à partir de la configuration initiale (c_{01}, c_{02}) en tenant compte de la synchronisation. Une première condition nécessaire est que c_i soit accessible à partir de c_{0i} quand les synchronisations ne sont pas considérées. De plus, si nous interprétons les chemins d'exécutions qui mènent vers c_1 et c_2 comme des séquences d'instructions de *synchronisation* comme suit : Les actions de synchronissation $a!$ et $a?$ sont représentées par a, et toutes les instructions internes qui peuvent être exécutées par l'un des deux programmes indépendamment de l'autre sont cachées, c-à-d., elles sont toutes représentées par une action invisible "τ". Nous obtenons de cette manière deux langages hors-contexte $Paths(c_{01}, c_1)$ et $Paths(c_{02}, c_2)$. Avec cette représentation, (c_1, c_2) est accessible ssi $Paths(c_{01}, c_1) \cap Paths(c_{02}, c_2)$ est non vide, c-à-d., s'il existe des chemins des deux composantes dont la séquence de communication est la même pour les deux partenaires. Observons que ceci est vrai puisque si "a" apparaît dans les chemins d'exécution des deux composantes, alors la condition **(A)** implique que ce signal a forcément été envoyé par l'un et reçu par l'autre. Malheureusement, puisque le problème du vide de l'intersection des langages hors-contexte est indécidable, il n'est pas possible de décider l'accessibilité de (c_1, c_2) à partir de (c_{01}, c_{02}).

En général, étant donnés deux ensembles $C_1 \times C_2$ et $C'_1 \times C'_2$ de configurations du système concurrent, il est indécidable de déterminer si une configuration de $C_1 \times C_2$ est accessible à partir d'une configuration de $C'_1 \times C'_2$; c-à-d., il n'est pas possible de déterminer si $Paths(C'_1, C_1) \cap Paths(C'_2, C_2)$ est vide. Comme expliqué dans l'introduction de cette partie, pour attaquer ce problème, nos calculons des abstractions $A(C'_i, C_i)$ satisfaisant $A(C'_i, C_i) \supseteq Paths(C'_i, C_i)$. Il est clair que si $A(C'_1, C_1) \cap A(C'_2, C_2)$ est vide alors $Paths(C'_1, C_1) \cap Paths(C'_2, C_2)$ l'est aussi. Donc nous vérifions si $A(C'_1, C_1) \cap A(C'_2, C_2) = \emptyset$, et si la réponse est positive, nous déduisons que $C_1 \times C_2$ n'est pas accessible à partir de $C'_1 \times C'_2$.

Le problème est de trouver des abstractions des langages de chemins qui soient calculables et pour lesquelles le problème du vide d'une intersection soit décidable. Dans ce chapitre, nous donnons un algorithme générique pour le calcul de telles approximations. Cet algorithme est basé sur (1) les constructions de [BEM97, EHRS00]

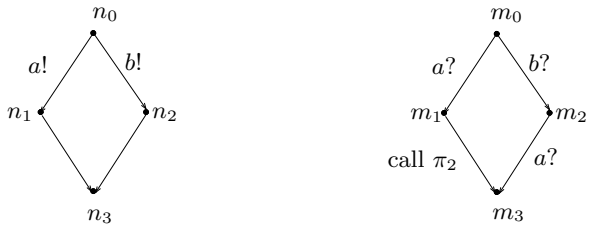


FIG. 7.1 – Un exemple

qui calculent, étant donné un ensemble arbitraire régulier C de configurations d'un programme séquentiel, les ensembles $pre^*(C)$ et $post^*(C)$ des prédeceseurs et successeurs de C. (2) La résolution d'inégalités polynômiales dans des domaines abstraits. Nous considérons deux classes d'abstractions : les abstractions à chaînes finies et les abstractions commutatives. Nous donnons plusieurs exemples de telles abstractions qui permettent différentes analyses en fonction du coût et de la précision désirés.

7.2 Modélisation

Les programmes que nous considérons correspondent à un nombre fixe n de processus séquentiels concurrents. Ils peuvent être représentés par des PFG tels que la procédure `main` crée n processus séquentiels en parallèle, et tels que les CFG de ces processus ne contiennent pas d'appels parallèles. Pour simplifier la présentation, nous supposons que de tels programmes sont donnés sous la forme d'un n-uplet de *systèmes de flow graphs*, un pour chaque composante séquentielle, où un *système de flow graphs* est un PFG qui modélise les programmes séquentiels, c-à-d., un PFG ne comprenant pas d'instructions correspondant à des appels parallèles de la forme $call(p_1||\cdots||p_k)$. Nous supposons que les noms des procédures des programmes séquentiels différents sont disjoints. Nous supposons également que les ensembles S_{ij} des signaux envoyés du processus i vers le processus j sont disjoints (nous pouvons toujours se ramener à ce cas par un renommage des signaux).

La figure 7.1 représente une paire de flow graphs qui correspond à un programme où deux processus séquentiels sont mis en parallèle. Ces processus communiquent par les signaux a et b. Sur la figure, les points n_0, n_3, m_0, et m_3 correpondent respectivement aux points d'entrée et de sortie des procédures π_1 et π_2 : $e_{\pi_1}, r_{\pi_1}, e_{\pi_2}$, et r_{π_2}.

7.2.1 Modèle formel : Systèmes à pile

Un *Système à pile* (PDS pour PushDown System) est un quintuplet

$$\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$$

où P est un ensemble fini d'*états de contrôle*, *Act* est un ensemble fini d'*actions*, Γ est l'*alphabet de la pile*, et $\Delta \subseteq (P \times \Gamma) \times Act \times (P \times \Gamma^*)$ est un ensemble fini de *règles de transition*. Si $((p,\gamma),a,(p',w)) \in \Delta$, nous écrivons $\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',w\rangle$. Une *configuration* de $\mathcal{P}$ est une paire $\langle p,w\rangle$ où $p \in P$ est un point de contrôle et $w \in \Gamma^*$ est le *contenu de la pile*. Nous supposons sans perte de généralité que toutes les règles de Δ sont de la forme $\langle p,\gamma\rangle \overset{a}{\hookrightarrow} \langle p',w\rangle$ tel que $|w| \leq 2$ (en fait, comme nous le verrons plus tard, les PDS obtenus à partir des programmes satisfont ces contraintes). c_0 est la *configuration initiale* de $\mathcal{P}$. L'ensemble de toutes les configurations est dénoté par $\mathcal{C}$. Un ensemble de configurations C est *régulier* si pour chaque point de contrôle $p \in P$ le langage $\{w \in \Gamma^* \mid \langle p,w\rangle \in C\}$ est régulier.

Pour chaque action a, nous définissons la relation $\overset{a}{\longrightarrow} \subseteq \mathcal{C} \times \mathcal{C}$ comme suit : si $\langle q,\gamma\rangle \overset{a}{\hookrightarrow} \langle q',w\rangle$, alors $\langle q,\gamma v\rangle \overset{a}{\longrightarrow} \langle q',wv\rangle$ pour chaque $v \in \Gamma^*$; $\langle q,\gamma v\rangle$ est un *prédécesseur immédiat* de $\langle q',wv\rangle$, et $\langle q',wv\rangle$ in *successeur immédiat* de $\langle q,\gamma v\rangle$.

La fonction des prédécesseurs $pre^*\colon 2^{\mathcal{C}} \to 2^{\mathcal{C}}$ de $\mathcal{P}$ est définie comme suit : c appartient à $pre^*(C)$ si un successeur de c appartient à C. Nous définissons $post^*(C)$ de manière similaire.

Un *Système à pile communicant* (CPDS pour Communicating PushDown System) est un n-uplet $CP = (\mathcal{P}_1,\ldots,\mathcal{P}_n)$ de systèmes à pile sur le même ensemble d'actions *Act*. Pour modéliser la communication, nous supposons que *Act* contient une action spéciale τ qui représente les actions internes, et que chaque action dans $Lab = Act \setminus \{\tau\}$ est une action de synchronisation.

Comme expliqué dans la première section, l'idée est de représenter toutes les instructions internes des processus par l'action invisible τ, et les instructions de synchronisation $a!$ et $a?$ par l'étiquette a. Nous ne distinguons par entre $a!$ et $a?$ car nous supposons que les ensembles des signaux S_{ij} sont disjoints. Donc, si de par notre modélisation, deux automates à pile (modélisant deux processus séquentiels concurrents) "exécutent" une action a, alors nous sommes sûrs que l'un des deux processus a émis le signal a et l'autre l'a reçu. Nous utilisons cette modélisation pour pouvoir parler d'une même "séquence de synchronisation" de deux automates à pile, et donc réduire le problème d'accessibilité à un problème d'intersection de langages hors-contextes comme expliqué dans la première section de ce chapitre.

Une *configuration globale* d'un CPDS est un n-uplet $g = (c_1,\ldots,c_n)$ de configurations de $\mathcal{P}_1,\ldots\mathcal{P}_n$. Nous étendons les relations $\overset{a}{\longrightarrow}$ aux paires de configurations globales comme suit. Soit $g = (c_1,\ldots,c_n)$ et $g' = (c'_1,\ldots,c'_n)$ des configurations globales :

- $g \overset{\tau}{\longrightarrow} g'$ s'il existe $1 \leq i \leq n$ tels que $c_i \overset{\tau}{\longrightarrow} c'_i$ et $c'_j = c_j$ pour chaque $j \neq i$;
- $g \overset{a}{\longrightarrow} g'$ s'il existe des indices $i \neq j$ tels que $c_i \overset{a}{\longrightarrow} c'_i$, $c_j \overset{a}{\longrightarrow} c'_j$, et $c'_k = c_k$ pour chaque $i \neq k \neq j$.

Comme expliqué ci-dessus, dans ce dernier cas, nous ne pouvons avoir $g \overset{a}{\longrightarrow} g'$ où a est un signal que si une des composantes (i ou j) émet le signal a et l'autre le reçoit.

Etant donné un ensemble G de configurations globales, nous définissons $pre^*(G)$ et $post^*(G)$ de manière standard.

7.2.2 Passer d'un n-uplet de systèmes de flow graphs à un CPDS

Etant donné un n-uplet de systèmes de flow graphs, nous définissons un CPDS correspondant comme suit. Nous associons à chacun des n systèmes de flow graphs un système à pile. Le CPDS est juste le n-uplet de ces n systèmes à pile.

En adoptant la modélisation donnée dans [ES01], nous associons à un système de flow graphs G_i ayant un ensemble de nœuds N_i et un ensemble d'arêtes E_i, le PDS $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ défini comme suit. P est l'ensemble de toutes les valuations possibles des variables globales, ou un singleton si le programme n'a pas de variables globales ou si toutes les variables globales ont été abstraites. Act contient l'action τ et une action a pour chaque signal a. Γ est l'ensemble de toutes les paires (n, v), où n est un nœud du flow graph, et v est une valuation des variables locales. La configuration initiale c_0 est définie par $c_0 = \langle glob_0, (e_{\texttt{main}_i}, loc_0)\rangle$, où $glob_0$ et loc_0 sont les valeurs initiales des variables globales et locales, et $e_{\texttt{main}_i}$ est le nœud initial de la procédure principale $\texttt{main}_i$ du processus i. Δ contient l'ensemble des règles :

Instruction vide : $\langle glob, (n_1, loc)\rangle \overset{\tau}{\hookrightarrow} \langle glob, (n_2, loc)\rangle$, si $n_1 \rightarrow n_2 \in E_i$;

Affectation : $\langle glob, (n_1, loc)\rangle \overset{\tau}{\hookrightarrow} \langle glob', (n_2, loc')\rangle$, si $n_1 \overset{a}{\rightarrow} n_2 \in E_i$ et a est une affectation, où $glob$ et $glob'$ (loc et loc') sont les valeurs des variables globales (locales) avant et après l'affectation ;

Conditionnelle "if-then-else" : $\langle glob, (n_1, loc)\rangle \overset{\tau}{\hookrightarrow} \langle glob, (n_2, loc)\rangle$, si $n_1 \overset{a}{\rightarrow} n_2 \in E_i$ et a est une conditionnelle "if-then-else", où $glob$ et loc sont tels que la condition de l'instruction est satisfaite ;

Appel récursif : si $n_1 \xrightarrow{call(p)} n_2 \in E_i$, alors considérer la règle

$$\langle glob, (n_1, loc)\rangle \overset{\tau}{\hookrightarrow} \langle glob, (e_p, loc_p) \cdot (n_2, loc)\rangle,$$

où e_p est le point d'entrée de la procédure p, loc_p les valeurs des arguments passées par la procédure appelante à la procédure p, et loc mémorise les variables locales de la procédure appelante ;

Terminaison :[1] $\langle glob, (r_p, loc)\rangle \overset{\tau}{\hookrightarrow} \langle glob', \epsilon\rangle$, où r_p est le point de sortie de la procédure p et $glob'$ mémorise la valeur retournée par la procédure après sa terminaison.
Synchronisation : Si $n_1 \overset{a!}{\rightarrow} n_1' \in E_i$ ou $n_1 \overset{a?}{\rightarrow} n_1' \in E_i$ alors considérer la règle $\langle glob_1, (n_1, loc_1)\rangle \overset{a}{\rightarrow} \langle glob_1, (n_1', loc_1)\rangle$.

La figure 7.1 représente un CPDS à deux PDSs correspondant au programme séquentiel du côté droit de la figure 7.2.

[1] Observons qu'ici, les résultats de retour des procédures appelées sont stockés dans les variables globales. Ceci est différent de la modélisation que nous avons considéré pour modéliser nos programmes par des PRS.

Notons que pour les programmes séquentiels, cette traduction vers les automates à pile peut aussi être vue comme une traduction vers PRS comme expliqué dans la section 4.3.2.

$\mathcal{P} = (\{p\}, \{a, b, \tau\}, \{m_0, m_1, m_2, m_3\}, \langle p, m_0\rangle, \{r_1, \ldots r_5\})$ où

$$r_1 : \langle p, m_0\rangle \overset{a}{\hookrightarrow} \langle p, m_1\rangle \qquad r_4 : \langle p, m_2\rangle \overset{a}{\hookrightarrow} \langle p, m_3\rangle$$
$$r_2 : \langle p, m_0\rangle \overset{b}{\hookrightarrow} \langle p, m_2\rangle \qquad r_5 : \langle p, m_3\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$$
$$r_3 : \langle p, m_1\rangle \overset{\tau}{\hookrightarrow} \langle p, m_0 m_3\rangle$$

FIG. 7.2 – Un système à pile

7.3 Abstraction des langages de chemin

7.3.1 Analyse d'accessibilité et langages de chemin

Soit $(\mathcal{P}_1, \ldots, \mathcal{P}_n)$ un CPDS. Considérons le problème de vérifier si un ensemble de configurations $C_1 \times \cdots \times C_n$ est accessible à partir de $C'_1 \times \cdots \times C'_n$. Ce problème peut être réduit au probléme de vérifier le vide de l'intersection de langages hors-contexte. En effet, l'ensemble des actions visibles Lab est une union disjointe d'ensembles $Lab_{i,j}$ correspondant aux actions de synchronisation entre chaque paire de systèmes $\mathcal{P}_i$ et $\mathcal{P}_j$ (puisque les S_{ij} sont disjoints). Soit $L_i = Paths(C'_i, C_i)$ l'ensemble des chemins menant dans $\mathcal{P}_i$ d'une configuration de C'_i à une configuration de C_i, et soit

$$\widehat{L_i} = L_i \sqcup\!\sqcup (\bigcup_{j,k \neq i} Lab_{j,k})^*$$

où $\sqcup\!\sqcup$ est l'opérateur de shuffle défini comme suit : si u et v deux mots de Lab, alors

$$u \sqcup\!\sqcup v = \{u_1 v_1 u_2 v_2 \cdots u_n v_n \mid u = u_1 \cdots u_n, v = v_1 \cdots v_n\}$$

Ceci exprime que nous insérons partout dans tous les chemins de $\mathcal{P}_i$ des étiquettes correspondant aux actions de synchronisation entre des paires d'autres processus. Autrement dit, nous étendons chaque système à pile $\mathcal{P}_i$ par des boucles (self loops) sur chaque état de contrôle étiquetées par des actions de synchronisation entre des paires d'autres processus. Notons que cette extension n'est faite que lorsque $n \geq 3$ (pour $n = 2$, nous avons $\widehat{L_i} = L_i$).

Il est facile de voir que $C_1 \times \cdots \times C_n$ est accessible à partir de $C'_1 \times \cdots \times C'_n$ si et seulement si

$$\widehat{L_1} \cap \ldots \cap \widehat{L_n} \neq \emptyset$$

Comme mentionné dans la première section, notre approche pour attaquer ce problème (qui est indécidable) est basée sur le calcul d'abstractions $A(C', C)$ des langages de chemins $Paths(C', C)$ des systèmes à pile, pour des ensembles donnés de configurations C' et C. Une fois des abstractions $A(C'_i, C_i)$ des langages de chemins $\widehat{L_i}$ ont été calculées, nous testons si leur intersection est vide, auquel cas, nous concluons que $C_1 \times \cdots \times C_n$ n'est pas accessible à partir de $C'_1 \times \cdots \times C'_n$.

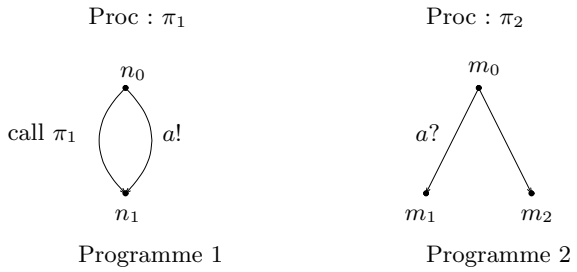


FIG. 7.3 – Un autre exemple

7.3.2 Exemples d'abstractions de langages de chemins

Nous considérons dans ce qui suit quatre exemples d'abstractions. Notons que pour appliquer notre approche, nous avons besoin d'une représentation finie de $A(C', C)$. Donc nous devons calculer un *objet abstrait* représentant $A(C', C)$.

Nous décrivons ci-dessous les quatre abstractions que nous considérons de manière informelle. Nous donnons les définitions formelles plus tard.

7.3.2.1 Les ensembles d'actions interdites et nécessaires

L'objet abstrait est une paire $[F, R]$, où $F, R \subseteq Lab$. F (pour Forbidden), est l'ensemble des actions *interdites*. Il contient les lettres a qui n'apparaissent dans aucune séquence de $Paths(C', C)$. R (pour Required), est l'ensemble des actions *nécessaires*. Il contient les lettres a qui apparaissent dans toutes les séquences de $Paths(C', C)$. $[F, R]$ représente le langage de toutes les séquences qui ne contiennent aucune lettre de F et au moins une occurrence de chaque lettre de R. Il est facile de voir que les langages représentés par $[F_1, R_1]$ et $[F_2, R_2]$ ont une intersection vide si et seulement si $F_1 \cap R_2 \neq \emptyset$ ou $R_1 \cap F_2 \neq \emptyset$.

Dans la figure 7.3 nous pouvons utiliser cette abstraction pour déduire qu'aucune configuration ayant n_1 et m_2 comme points de contrôle n'est accessible : L'action a est nécessaire pour accéder à n_1, mais interdite si nous voulons arriver à m_2.

7.3.2.2 Label bitvectors

Dans cette deuxième abstraction, l'objet abstrait représentant $A(C', C)$ est un ensemble B de vecteurs de bits $Lab \to \mathbb{B}$. Un vecteur de bits b appartient à B s'il existe une séquence dans $Paths(C', C)$ telle que $b(a) = 1$ si a apparaît dans la séquence et $b(a) = 0$ sinon.

Dans la figure 7.1 nous pouvons utiliser cette abstraction pour déduire qu'aucune configuration ayant n_3 et m_3 comme points de contrôle n'est accessible. Puisque les deux programmes séquentiels n'ont pas de variables, leurs PDSs correspondant ont chacun un état de contrôle p_{01} et p_{02}. Les alphabets de pile sont $\Gamma_1 = \{n_0, \ldots, n_3\}$, $\Gamma_2 = \{m_0, \ldots m_3\}$. Les configurations initiales sont $c_{01} = (p_{01}, n_0)$ et $c_{02} = (p_{02}, m_0)$. Nous dénotons par $B(n_3)$ (resp. $B(m_3)$) les ensembles des vecteurs de bits correspondants à l'ensemble $Paths(c_{01}, \{\langle p_{01}, n_3 w\rangle \mid w \in \Gamma_1^*\})$ des séquences qui mènent de c_{01}

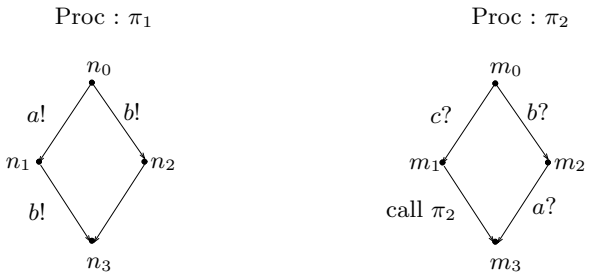


FIG. 7.4 – Exemple 3

à une configuration où le programme 1 est au point de contrôle n_3 (resp. l'ensemble $Paths(c_{02}, \{\langle p_{02}, m_3 w\rangle \mid w \in \Gamma_2^*\})$ des séquences qui mènent de c_{02} à une configuration où le programme 1 est au point de contrôle m_3). Nous avons $B(n_3) = \{10, 01\}$, alors que $B(m_3) = \{11\}$ (les composantes des vecteurs de bits correspondent aux lettres a, b). Comme $B(n_3) \cap B(m_3) = \emptyset$, nous déduisons qu'il n'est pas possible d'atteindre une configuration où n_3 et m_3 sont tous les deux actifs. Observons que l'abstraction précédente ne permet pas d'arriver à cette conclusion puisque pour arriver à n_3 ou à m_3 les actions a et b sont toutes les deux permises.

7.3.2.3 Ordre des premières occurrences

L'objet abstrait représentant $A(C', C)$ est un ensemble W de mots w tels que pour chaque $a \in Lab$, $|w|_a \leq 1$ ($|w|_a$ dénote le nombre d'occurrences de la lettre a dans w). Un mot $w = a_1 \cdots a_n$ appartient à W s'il existe un chemin dans $Paths(C', C)$ tel que l'ensemble des lettres qui apparaissent dans w est précisément égal à $\{a_1, \ldots, a_n\}$, et en plus, les premières occurrences de ces lettres dans le chemin apparaissent dans l'ordre défini par w (c-à-d., pour chaque $i < j$, a_i apparait pour la première fois avant a_j). Tester le vide de l'intersection des langages représentés par W_1 et W_2 est équivalent à tester le vide de leur intersection en tant qu'ensembles.

Nous pouvons utiliser cette abstraction pour déduire qu'aucune configuration du programme de la figure 7.4 ayant n_3 et m_3 comme points de contrôle n'est accessible à partir d'une configuration initiale. Comme précédemment, puisque les deux programmes séquentiels n'ont pas de variables, leurs PDSs correspondant ont chacun un état de contrôle p_{01} et p_{02}. Les alphabets de pile sont $\Gamma_1 = \{n_0, \ldots, n_3\}$, $\Gamma_2 = \{m_0, \ldots m_3\}$. Les configurations initiales sont $c_{01} = \{(p_{01}, n_0)$, $c_{02} = (p_{02}, m_0)$. Nous dénotons par $W(n_3)$ et $W(m_3)$ les ensembles des mots correspondant à $Paths(c_{01}, \{\langle p_{01}, n_3 w\rangle \mid w \in \Gamma_1^*\})$ et $Paths(c_{02}, \{\langle p_{02}, m_3 w\rangle \mid w \in \Gamma_2^*\})$. Nous avons $W(n_3) = \{ab, b\}$, et $W(m_3) = \{cba, ba\}$. Puisque $W(n_3) \cap W(m_3) = \emptyset$, il n'y a pas de séquence de communication commune aux deux programmes séquentiels permettant d'accéder aux points de contrôle n_3 et m_3.

Observons que l'abstraction précédente ne permet pas de conclure qu'aucune configuration où les programmes 1 et 2 sont simultanément aux points de contrôles n_3 et

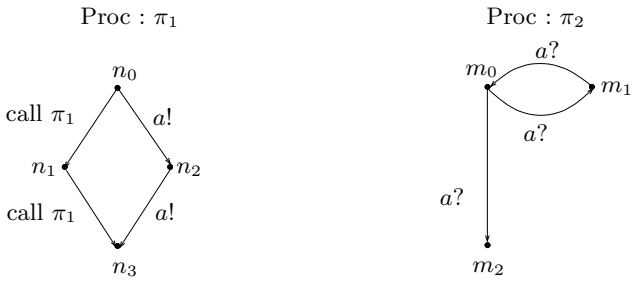


FIG. 7.5 – Un dernier exemple

m_3 n'est accessible. En effet, cette abstraction mémorise l'information que pour arriver à n_3 ou à m_3, il faut exécuter une séquence d'actions où a et b sont tous les deux présents, sans tenir compte de l'ordre dans lequel ces actions sont exécutées, alors que dans ce cas, l'ordre est important.

7.3.2.4 Les images de Parikh

C'est l'abstraction la plus précise que nous considérons. L'objet abstrait représentant $A(C', C)$ est l'image de Parikh de $Paths(C', C)$.

Dans la figure 7.5 nous pouvons conclure qu'aucune configuration ayant n_3 et m_2 comme points de contrôle n'est accessible : Pour arriver à n_3, a doit s'exécuter un nombre paire de fois ; alors que pour atteindre m_3, elle doit s'exécuter un nombre impair de fois.

Il y a plusieurs autres abstractions possibles. Par exemple, nous pouvons combiner les label bitvectors et les images de Parikh : Pour certaines actions, nous nous intéressons à savoir si elles apparaissent dans la séquence ou non, tandis que pour d'autres, nous comptons le nombre de leur apparition. De plus, il est parfois utile de compter modulo un nombre (comme dans le cas de la figure 7.5).

7.3.3 Un cadre formel

7.3.3.1 Préliminaires

Nous rappelons dans ce qui suit quelques définitions standards qui vont nous être utiles pour définir notre cadre formel d'abstraction.

Treillis complets et points fixes :

Soit E un ensemble donné. Un *ordre* $\leq$ sur E est une relation dans $E \times E$ qui est :
- réflexive : $\forall x \in E, x \leq x$;
- antisymétrique : si $x \leq y$ et $y \leq x$, alors $x = y$;
- transitive : si $x \leq y$ et $y \leq z$, alors $x \leq z$.

Définition 7.3.1 (Treillis complets) *Soit X une partie de E, y est un* majorant *(resp.* minorant*) de X si $\forall x \in X$, $x \leq y$ (resp. $y \leq x$). Soit $Maj(X)$ (resp. $Min(X)$) l'ensemble des majorants (resp. minorants) de X. y est un* plus grand élément *de X (resp.* plus petit élément *de X) si $y \in X$ et y est un majorant (resp. minorant) de X. X admet une* borne supérieure $\sqcup X$ *(resp. une* borne inférieure $\sqcap X$*) ssi $Maj(X)$ admet $\sqcup X$ comme plus petit élément (resp. $Min(X)$ admet $\sqcap X$ comme plus grand élément). $(E, \leq, \sqcup, \sqcap, \bot, \top)$ est un* treillis complet *ssi toute partie X de E admet une borne inférieure $\sqcap X$ et une borne supérieure $\sqcup X$. En particulier, $\bot = \sqcap E = \sqcup \emptyset$ et $\top = \sqcup E = \sqcap \emptyset$.*

Définition 7.3.2 (Fonctions monotones et fonctions continues) *Soit f une fonction de E dans E, f est monotone ssi*

$$x \leq y \Rightarrow f(x) \leq f(y).$$

f est $\sqcup$-continue ssi

$$\forall X \subseteq E, f(\sqcup X) = \sqcup_{x \in X} f(x)$$

$x \in E$ est un point fixe (resp. pré-point fixe) d'une fonction f ssi $x = f(x)$ (resp. $f(x) \leq x$).

Théorème 7.3.1 (Tarski [Tar55]) *Si $(E, \leq, \sqcup, \sqcap, \bot, \top)$ est un treillis complet, et si f est une fonction monotone de $E \to E$, alors f admet un plus petit point fixe qui est aussi son plus petit pré-point fixe.*

Théorème 7.3.2 (Kleene) *Si $(E, \leq, \sqcup, \sqcap, \bot, \top)$ est un treillis complet, et si f est une fonction continue de $E \to E$, alors le plus petit point fixe de f est égal à*

$$\sqcup_n f^n(\bot)$$

avec $f^0(x) = x$, et $f^{n+1}(x) = f(f^n(x))$.

Abstractions et connexions de Galois :

Notons par $\mathcal{L}$ le treillis complet des langages sur Lab, c-à-d.,

$$\mathcal{L} = (2^{Lab^*}, \subseteq, \cup, \cap, \emptyset, Lab^*).$$

Formellement, une abstraction de $\mathcal{L}$ [CC77] consiste en un *treillis abstrait*

$$\mathcal{D} = (D, \leq, \sqcup, \sqcap, \bot, \top)$$

où D est un domaine appelé le *domaine abstrait*, et une *connexion de Galois* (α, γ) entre $\mathcal{L}$ et $\mathcal{D}$, c-à-d., une paire de fonctions $\alpha : 2^{Lab^*} \to D$ dite *la fonction d'abstraction* et $\gamma : D \to 2^{Lab^*}$ dite *la fonction de concrétisation* telles que

$$\forall x \in 2^{Lab^*},\ \forall y \in D.\ \alpha(x) \leq y \Longleftrightarrow x \subseteq \gamma(y)\ .$$

Une connexion de Galois assure que pour tout langage L,

$$L \subseteq \gamma\big(\alpha(L)\big)$$

ce qui exprime que la concrétisation de l'abstraction d'un langage L est une sur-approximation de L. C'est ce qui assure que l'abstraction est correcte.

7.3.3.2 Notre cadre formel d'abstraction

Notre but est de définir des abstractions de $\mathcal{L}$ telles que (i) le langage de chemins abstrait $\alpha(Paths(C'_i, C_i))$ peut être calculé quand C_i et C'_i sont réguliers, et (ii) le vide d'une intersection soit décidable dans le domaine D. Pour ceci, nous considérons une classe d'abstractions, que nous appelons *abstractions de Kleene*, pour lesquelles le treillis $\mathcal{D}$ a une bonne structure algébrique. Dans le reste de cette section nous définissons ces abstractions, et nous montrons dans la section suivante que les quatre exemples de la section 7.3.2 sont en fait des abstractions de Kleene.

Algèbres de Kleene :

Un *semi-anneau idempotent* est un quintuplet $\mathcal{K} = (K, \oplus, \odot, \bar{0}, \bar{1})$, où $\oplus$ est une opération associative, commutative, et idempotente ($a \oplus a = a$), et $\odot$ est une opération associative. $\bar{0}$ et $\bar{1}$ sont les éléments neutres pour $\oplus$ et $\odot$, respectivement. $\bar{0}$ est un élément absorbant pour $\odot$ ($a \odot \bar{0} = \bar{0} \odot a = \bar{0}$), et $\odot$ est distributif par rapport à $\oplus$. $\mathcal{K}$ est un *Lab-semi-anneau* si K est généré par $\bar{0}$, $\bar{1}$, et un élément v_a pour chaque $a \in Lab$. Un semi-anneau est *fermé* si $\oplus$ peut être étendu en un opérateur sur les ensembles infinis énumérables (c-à-d., les sommes infinies énumérables sont permises), et cet opérateur a les mêmes propriétés que $\oplus$ (il est associatif, commutatif, idempotent, et $\odot$ est distributif par rapport à cet opérateur). Dans les semi-anneaux fermés, nous pouvons définir $a^0 = \bar{1}$, $a^{n+1} = a \odot a^n$, et $a^\star = \bigoplus_{n \geq 0} a^n$. Rajouter l'opération "$\star$" à $\mathcal{K}$ le transforme en une *algèbre de Kleene*.

Convention 7.3.1 *Dans la suite, nous réservons le terme algèbre de Kleene pour les algèbres de Kleene induites par des Lab-semi-anneaux fermés et idempotents.*

Soit $\mathcal{K}$ une algèbre de Kleene avec un domaine K. Pour des raisons techniques que nous verrons plus tard, nous avons besoin d'introduire la notion de "miroir" x^R (R pour Reverse) d'un élément x de K. Cette notion est définie inductivement comme suit :

- $v_a^R = v_a$,
- $(x \oplus y)^R = x^R \oplus y^R$,
- $(x \odot y)^R = y^R \odot x^R$, et
- $(x^\star)^R = (x^R)^\star$.

Il s'en suit que pour chaque élément x de K, nous avons :

$$(x^R)^R = x. \tag{7.1}$$

Abstractions de Kleene :

Un treillis abstrait $\mathcal{D} = (D, \leq, \sqcup, \sqcap, \bot, \top)$ est *compatible* avec une algèbre de Kleene $\mathcal{K} = (K, \oplus, \odot, \star, \bar{0}, \bar{1})$ si $K = D$, et l'ordre, la borne inférieure, et l'opération $\sqcup$ du treillis abstrait sont donnés par $x \leq y$ si $x \oplus y = y$, $\bot = \bar{0}$, et $\sqcup = \oplus$, respectivement.

Une *abstraction de Kleene* est une abstraction dans laquelle le treillis abstrait $\mathcal{D}$ est compatible avec l'algèbre de Kleene $\mathcal{K} = (K, \oplus, \odot, \bar{0}, \bar{1})$, et la connexion de Galois

est donnée par $\alpha : 2^{Lab^*} \to K$ et $\gamma : K \to 2^{Lab^*}$ telles que

$$\begin{aligned} \alpha(L) &= \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n} \\ \gamma(x) &= \{a_1 \cdots a_n \in 2^{Lab^*} \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x\} \end{aligned}$$

Il est facile de vérifier que (α, γ) est une connexion de Galois.

Intuitivement, une abstraction de Kleene est telle que les opérations abstraites $\oplus$, $\odot$, et $\star$ de $\mathcal{K}$ correspondent à l'union, la concaténation, et la clôture de Kleene des langages du treillis $\mathcal{L}$. $\bar{0}$ et $\bar{1}$ sont les objets abstraits correspondant au langage vide et à $\{\epsilon\}$, respectivement. L'élément v_a est l'objet abstrait correspondant au langage $\{a\}$. La borne supérieure $\top \in K$ est l'objet abstrait correspondant au langage Lab^*, et l'opération $\sqcap$ correspond à l'intersection des langages dans le treillis $\mathcal{L}$. Si nous souhaitons calculer $\alpha(L)$ pour un langage L, nous "abstrayons" chaque mot $a_1 \ldots a_n$ de L par l'objet $v_{a_1} \odot \cdots \odot v_{a_n}$, et nous considérons l'union de ces objets.

Propriétés des abstractions de Kleene :

Nous obtenons directement de la définition de α donnée ci-dessus que pour chaque $L \in 2^{Lab^*}$,

$$\alpha\big(miroir(L)\big) = \big(\alpha(L)\big)^R. \tag{7.2}$$

De plus, dans une abstraction de Kleene nous avons $\alpha(\emptyset) = \bot$ et $\gamma(\bot) = \emptyset$, ce qui implique que

Proposition 7.3.1

$$\forall L_1, \ldots, L_n.\ \alpha(L_1) \sqcap \cdots \sqcap \alpha(L_n) = \bot \ \Rightarrow\ L_1 \cap \cdots \cap L_n = \emptyset$$

Preuve : Nous raisonnons par l'absurde. Supposons que $\alpha(L_1) \sqcap \ldots \sqcap \alpha(L_n) = \bot$ et que $L_1 \cap \ldots \cap L_n \neq \emptyset$. Soit alors $x \in L_1 \cap \ldots \cap L_n$. Donc $\alpha(x) = \bot$. Puisque (α, γ) est une connexion de Galois, et que $\alpha(x) \leq \alpha(x)$, il s'en suit que $x \subseteq \gamma(\alpha(x)) = \gamma(\bot) = \emptyset$, ce qui constitue une contradiction. □

Cette propriété est nécessaire pour notre approche : Pour tester le vide de l'intersection de deux langages hors-contexte, il suffit de tester le vide de l' intersection de leurs abstractions.

Abstractions de Kleene particulières :

Dans ce travail, nous considérons deux familles d'abstractions de Kleene :

- Les *abstractions à chaînes finies* qui sont les abstractions telles que K ne contient pas de chaîne ascendante infinie. Les *abstractions finies* où le domaine abstrait K est fini sont des cas particuliers d'abstractions à chaînes finies.
- Les *abstractions commutatives* qui sont les abstractions où $\odot$ est commutatif. Intuitivement, ceci veut dire que $\odot$ "oublie l'ordre" entre les lettres. Nous obtenons ainsi que $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ est une *algèbre de Kleene commutative*.

Nous montrons que notre problème peut être résolu dans le cadre de ces deux abstractions.

7.3.4 Instances des abstractions

Nous montrons que les quatre exemples de la section 7.3.2 sont soit des abstractions à chaînes finies soit des abstractions de Kleene commutatives. Nous définissons les algèbres de Kleene correspondantes en donnant le domaine, les définitions de $\oplus$, $\odot$, et les éléments $\overline{1}$ et $\overline{0}$. Notons que par la convention 7.3.1, l'opération $\star$ est déterminée par $\oplus$ et $\odot$.

7.3.4.1 Les ensembles d'actions interdites et nécessaires

L'algèbre de Kleene $\mathcal{K}$ est définie comme suit :
- $K = \{\overline{0}\} \cup \{[F, R] \in 2^{Lab} \times 2^{Lab} \mid F \cap R = \emptyset\}$,
- $[F_1, R_1] \oplus [F_2, R_2] = [F_1 \cap F_2, R_1 \cap R_2]$,
- $[F_1, R_1] \odot [F_2, R_2] = [F_1 \cap F_2, R_1 \cup R_2]$,
- $\overline{0}$ est par définition neutre pour $\oplus$ et absorbant pour $\odot$,
- $\overline{1} = [Lab, \emptyset]$.

Dans ce cas, K est engendré par les éléments v_a de chaque $a \in Lab$, où $v_a = [Lab \setminus \{a\}, \{a\}]$. Le treillis abstrait est obtenu en prenant $\top = [\emptyset, \emptyset]$, et définissant $\sqcap$ par :
- $[F_1, R_1] \sqcap [F_2, R_2] = [F_1 \cup F_2, R_1 \cup R_2]$ si $(F_1 \cap R_2) \cup (F_2 \cap R_1) = \emptyset$,
- $[F_1, R_1] \sqcap [F_2, R_2] = \overline{0}$ sinon,
- $\overline{0} \sqcap x = x \sqcap \overline{0} = \overline{0}$.

Observons que cette abstraction est en même temps une abstraction à chaînes finies et une abstraction commutative.

7.3.4.2 Label bitvectors

Nous considérons l'algèbre de Kleene $\mathcal{K}$ donnée par :

1. $K = 2^{[Lab \to \mathbb{B}]}$, où $[Lab \to \mathbb{B}]$ est l'ensemble des vecteurs de bits indexés par Lab,
2. $\oplus = \cup$,
3. $\odot = \vee$ (l'opérateur booléen $\vee$ est étendu aux vecteurs de bits de la manière suivante : $b_1 \vee b_2 = b$ t.q. $\forall a \in Lab$, $b(a) = b_1(a) \vee b_2(a)$),
4. $\overline{0} = \emptyset$, et
5. $\overline{1} = \{(0, \ldots, 0)\}$.

Il est facile de voir que c'est une algèbre de Kleene engendrée par les éléments v_a pour chaque a dans Lab, où v_a est le singleton $\{b_a\}$ donné par $b_a(a) = 1$, et $b_a(a') = 0$ pour tout $a' \neq a$, Le treillis abstrait est obtenu en prenant $\top = 2^{[Lab \to \mathbb{B}]}$, et $\sqcap = \cap$. Comme dans le cas précédent, cette abstraction est à la fois une abstraction à chaînes finies et une abstraction commutative.

7.3.4.3 Ordre des premières occurrences

Soit $W = \{w \in Lab^* \mid \forall a \in Lab, |w|_a \leq 1\}$, c-à-d., l'ensemble des mots où chaque lettre apparaît au plus une fois. Nous considérons l'algèbre de Kleene $\mathcal{K}$ donnée par :

1. $K = 2^W$;
2. $\oplus = \cup$,

3. $U \odot V$ est l'ensemble des mots $u_1 \cdot v_2'$ tels que $u_1 \in U$ et il existe $v_2 \in V$ tel que v_2' est la projection de v_2 sur l'ensemble des lettres qui n'apparaissent pas dans u_1,
4. $\overline{0} = \emptyset$, et
5. $\overline{1} = \{\epsilon\}$. Il est facile de voir que cette algèbre est en effet une algèbre de Kleene engendrée par les éléments $v_a = \{a\}$ pour chaque a dans Lab.

Le treillis abstrait compatible avec $\mathcal{K}$ est obtenu en prenant $\top = W$, et $\sqcap = \cap$. Observons que cette abstraction est une abstraction à chaînes finies puisque K est fini.

7.3.4.4 Images de Parikh

Les éléments abstraits sont des ensembles semilinéaires de vecteurs d'entiers dans $[Lab \to \mathbb{N}]$. Nous considérons l'algèbre commutative $\mathcal{K}$ donnée par :

1. K est l'ensemble de tous les ensembles semilinéaires,
2. $\oplus = \cup$,
3. $S_1 \odot S_2 = \{\vec{u}_1 + \vec{u}_2 \mid \vec{u}_1 \in S_1, \vec{u}_2 \in S_2\}$,
4. $\bar{0} = \emptyset$, et
5. $\bar{1}$ est un singleton contenant le vecteur qui associe 0 à toutes les lettres de Lab.

Il est facile de voir que $\mathcal{K}$ est une algèbre de Kleene engendrée par les éléments $v_a = \{\vec{u}_a\}$ pour chaque $a \in Lab$, où $\vec{u}_a(a) = 1$ et $\vec{u}_a(b) = 0$ pour chaque $b \neq a$.

Le treillis abstrait est obtenu en prenant l'ensemble de tous les vecteurs de $[Lab \to \mathbb{N}]$ comme l'élément $\top$, et en définissant $\sqcap$ comme l'intersection des ensembles semilinéaires. Nous utilisons ici le fait que les ensembles semilinéaires sont effectivement fermés par intersection (par le théorème 2.1.6). De plus, si nous représentons un ensemble semilinéaire comme un ensemble de paires $(\vec{b}, \{\vec{p}_1, \ldots, \vec{p}_n\})$, il est facile de calculer la représentation de $S_1 \oplus S_2$, $S_1 \cdot S_2$, et $S_1^\star$ à partir des représentations de S_1 et S_2.

7.4 Calculer les langages de chemins abstraits

Etant donnée une abstraction de Kleene , le langage de chemins abstrait $\alpha(Paths(C', C))$ est un élément particulier de l'algèbre de Kleene correspondante. Nous donnons une procédure générique pour calculer cet élément en fonction des éléments de base v_a qui engendrent l'algèbre. Tout au long de cette section, nous illustrons les différentes étapes de notre construction sur le PDS de la figure 7.2.

7.4.1 K-prédécesseurs et K-successeurs

Nous introduisons dans ce qui suit la notion de K-configuration et K-relation de transition pour un système à pile.

Nous fixons un automate à pile $\mathcal{P} = (P, Act, \Gamma, c_0, \Delta)$ et une algèbre de Kleene $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ correspondant à une abstraction de Kleene. L'ensemble des expressions de chemins Π_K est le sous ensemble de K défini inductivement comme étant le plus petit ensemble tel que :

– $\bar{1} \in \Pi_K$,

– si $\pi \in \Pi_K$, alors pour chaque $a \in Lab$, $v_a \odot \pi \in \Pi_K$.

Etant donnée une expression de chemin π, nous dénotons par $|\pi|$ (la *longueur* de π) le nombre d'occurrences des éléments de base v_a dans π.

Une K-configuration de $\mathcal{P}$ est une paire (c, π), où c est une configuration de $\mathcal{P}$ et π est une expression de chemins sur K. Nous étendons la relation de transition de l'automate à pile $\xrightarrow{a}$ aux K-configurations comme suit : si $c \xrightarrow{a} c'$ pour une lettre $a \in Lab$, alors $(c, \pi) \rightarrow_K (c', v_a \odot \pi)$ et $(c', \pi) \leftarrow_K (c, v_a \odot \pi)$ pour chaque $\pi \in \Pi_K$; en plus, si $c \xrightarrow{\tau} c'$, alors $(c, \pi) \rightarrow_K (c', \pi)$ et $(c, \pi) \leftarrow_K (c', \pi)$ pour chaque $\pi \in \Pi_K$. $(c', v_a \odot \pi)$ (respectivement (c', π)) est un K-successeur immediat de (c, π) et $(c, v_a \odot \pi)$ (respectivement (c, π)) est un K-prédécesseur immediat de (c', π). La *relation d'accessibilité en avant (en arrière) sur* K $\Rightarrow_K$ ($\Leftarrow_K$) est la clôture réflexive-transitive de $\rightarrow_K$ ($\leftarrow_K$).

Etant donné un ensemble de configurations C, nous définissons $pre^*_K(C)$ ($post^*_K(C)$) comme l'ensemble des K-configurations (c, π) telles que $(c', \bar{1}) \Leftarrow_K (c, \pi)$ ($(c', \bar{1}) \Rightarrow_K (c, \pi)$) pour un $c' \in C$. Intuitivement, un K-prédécesseur (c', π) d'une configuration c mémorise dans π l'information concernant la séquence de lettres qui mènent de c à c' (si la séquence est $a_1 \dots a_n$, alors l'information stockée est $v_{a_1} \odot \dots \odot v_{a_n}$), et un K-successeur (c', π) d'une configuration c est tel que $\pi = v_{a_n} \odot \dots \odot v_{a_1}$ si la séquence $a_1 \dots a_n$ mène de c à c'.

En d'autres termes, nous avons

$$\begin{aligned} pre^*_K(c) &= \{(c', \pi) \mid c' \in pre^*(c),\ \pi \in \alpha(Paths(c', c))\} \\ post^*_K(c) &= \{(c', \pi) \mid c' \in post^*(c),\ \pi \in \alpha\Big(miroir\big(Paths(c, c')\big)\Big)\} \end{aligned}$$

7.4.2 K-automates

L'approche algorithmique que nous proposons pour calculer $\alpha(Paths(C', C))$ est basée sur le calcul d'automates représentant les images par pre^*_K et $post^*_K$ de langages réguliers de configurations.

Dans [BEM97, EHRS00], des automates sont utilisés pour représenter des ensembles (qui peuvent être infinis) de configurations. Nous étendons cette notion aux K-automates pour pouvoir manipuler des ensembles réguliers de K-configurations.

Définition 7.4.1 *Un K-automate pour le PDS $\mathcal{P}$ est un quintuplet $\mathcal{A} = (\Gamma, Q, \delta, P, F)$ où Q est un ensemble fini d'états, $\delta \subseteq Q \times \Gamma \times K \times Q$ est un ensemble de transitions, $P \subseteq Q$ est un ensemble d'états initiaux, et $F \subseteq Q$ est l'ensemble des états finaux.*

Nous définissons la relation de transition $\longrightarrow \subseteq Q \times \Gamma^* \times K \times Q$ *comme la plus petite relation telle que :*

– *si $(q, \gamma, e, q') \in \delta$ alors $q \xrightarrow{(\gamma, e)} q'$,*

– $q \xrightarrow{(\epsilon, \bar{1})} q$ *pour chaque $q \in Q$, et*

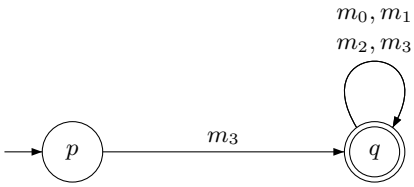


FIG. 7.6 – Un automate acceptant $\langle p, m_3\Gamma^*\rangle = \{\langle p, m_3 w\rangle \mid w \in (m_0 + \ldots + m_3)^*\}$

– *si* $q \xrightarrow{(w_1,e_1)} q''$ *et* $q'' \xrightarrow{(w_2,e_2)} q'$ *alors* $q \xrightarrow{(w_1 w_2, e_1 \odot e_2)} q'$.

Un chemin de $\mathcal{A}$ *est une séquence* $\rho = q_0 \xrightarrow{(\gamma_1,e_1)} q_1 \cdots q_{n-1} \xrightarrow{(\gamma_n,e_n)} q_n$. *Ce chemin est aussi dénoté par* $\rho = q_0 \xrightarrow{(w,e)} q$, *où* $w = \gamma_1 \odot \cdots \odot \gamma_n$ *et* $e = e_1 \odot \cdots \odot e_n$. *Nous dénotons l'expression e par* $\lambda\big((q_0, w, q)_\rho\big)$. *Si* $w = \gamma \in \Gamma$, *c-à-d., si* ρ *correspond à une transition, nous écrivons simplement* $e = \lambda\big((q_0, \gamma, q)\big)$. *Ce chemin correspond à une exécution de* $\mathcal{A}$ *si* $q_0 \in P$. *Cette exécution est dite* acceptante *si* $q_n \in F$. *Dans ce cas,* $\mathcal{A}$ *accepte la paire* $\big(\langle q_0, w\rangle, e\big)$.

Les automates finis ordinaires peuvent être vus comme des K-automates dont toutes les transitions sont étiquetées par $\bar{1}$. De manière équivalente, ils peuvent être obtenus en éliminant toutes les références à K dans la définition des K-automates. De ce fait, un automate reconnaît l'ensemble des configurations

$$\{\langle p, w\rangle \mid p \xrightarrow{w} q \text{ pour } p \in P \text{ et } q \in F\}.$$

La figure 7.6 montre un automate pour le PDS de la figure 7.2 qui accepte l'ensemble des configurations $\langle p, m_3\Gamma^*\rangle = \{\langle p, m_3 w\rangle \mid w \in (m_0 + \ldots + m_3)^*\}$.

Etant donné un automate $\mathcal{A}$, nous dénotons par $Conf(\mathcal{A})$ l'ensemble des configurations reconnues par $\mathcal{A}$. Dans le reste de ce chapitre, nous utilisons les symboles p, p', p'', p_i, etc. pour représenter les états initiaux des (K-)automates. Les états arbitraires sont dénotés par q, q', q'', q_i, etc.

7.4.3 Réduction au calcul des images par pre^*_K et $post^*_K$

Soient C et C' deux ensembles réguliers de configurations représentés par des automates finis, et considérons le problème de calculer $\alpha(Paths(C', C))$. Supposons que nous sommes capables de calculer un K-automate $\mathcal{A}_{pre^*_K}$ qui reconnaît l'ensemble $pre^*_K(C)$. Alors, nous pouvons construire un K-automate $\mathcal{A}^{C'}_{pre^*_K}$ sur $\Gamma \times K$ qui est la restriction de $\mathcal{A}_{pre^*_K}$ aux configurations de C'. Cet automate peut être facilement construit par un produit entre $\mathcal{A}_{pre^*_K}$ et l'automate reconnaissant C'. [2] Alors, $\alpha(Paths(C', C))$

[2]Nous considérons le plus petit ensemble de transitions tel que si $q_1 \xrightarrow{(a,e)} q'_1$ est dans $\mathcal{A}_{pre^*_K}$ et $q_2 \xrightarrow{a} q'_2$ est dans $\mathcal{A}_{C'}$ alors $(q_1, q_2) \xrightarrow{(a,e)} (q'_1, q'_2)$ est dans $\mathcal{A}^{C'}_{pre^*_K}$.

est le K-*langage* de $\mathcal{A}^{C'}_{pre^*_K}$, défini comme l'élément de K donné par :

$$\bigoplus\{e_1 \odot \cdots \odot e_n \mid \mathcal{A}^{C'}_{pre^*_K} \text{ a une exécution acceptante étiquetée par } (w, e_1 \odot \cdots \odot e_n)\}.$$

Une approche symétrique peut être adoptée en calculant un automate $\mathcal{A}_{post^*_K}$ reconnaissant $post^*_K(C')$ et se restreindre ensuite à l'ensemble des configurations de C, obtenant ainsi un K-automate $\mathcal{A}^{C}_{post^*_K}$. Le K-*langage* de $\mathcal{A}^{C}_{post^*_K}$ est égal à

$$\alpha\Big(miroir\big(Paths(C', C)\big)\Big)$$

Ensuite, $\alpha(Paths(C', C))$ est obtenu en prenant le miroir de ce K-langage puisque nous avons à partir des équations (7.1) et (7.2) que

$$\alpha(Paths(C', C)) = \left(\alpha\Big(miroir\big(Paths(C', C)\big)\Big)\right)^R.$$

Par conséquent, nous avons réduit notre problème de calculer $\alpha(Paths(C', C))$ à (i) construire $\mathcal{A}_{pre^*_K}$ (ou $\mathcal{A}_{post^*_K}$), et (ii) construire une représentation finie du K-langage de $\mathcal{A}_{pre^*_K}$ (ou $\mathcal{A}_{post^*_K}$).

Le problème (ii) peut être résolu en utilisant les techniques standard pour résoudre des équations linéaires dans des algèbres de Kleene (par ex., en utilisant l'algorithme de Floyd-Warshall, ou l'élimination de Gauss). En effet, ce problème est une généralisation directe du problème du calcul d'expressions régulières associées aux automates finis.

Donc, le problème essentiel à résoudre est comment calculer des K-automates reconnaissant les images par $post^*_K$ et pre^*_K. Nous montrons que nous pouvons réduire ce problème à celui de résoudre des (in)équations polynômiales.

7.5 Calcul des K-prédécesseurs

Etant donné un ensemble de configurations C reconnu par un automate $\mathcal{A}$, nous construisons un K-automate $\mathcal{A}_{pre^*_K}$ qui reconnaît $pre^*_K(C)$. Nous supposons sans perte de généralité que $\mathcal{A}$ n'a pas de transitions menant à un état initial.

7.5.1 Une procédure générique

Nous procédons en deux étapes. D'abord, nous construisons un automate $\mathcal{A}_{pre^*}$ qui reconnaît l'ensemble $pre^*(C)$. Ensuite, nous étiquetons les transitions de $\mathcal{A}_{pre^*}$ par des éléments adéquats de K.

Pour la première étape nous utilisons la procédure de [BEM97, EHRS00], qui consiste à rajouter des nouvelles transitions à $\mathcal{A}$ en appliquant la *règle de saturation* suivante :

> Si $\langle p, \gamma\rangle \xhookrightarrow{a} \langle p', w\rangle$ et $p' \xrightarrow{w} q$ est une exécution de l'automate courant, rajouter une transition (p, γ, q).

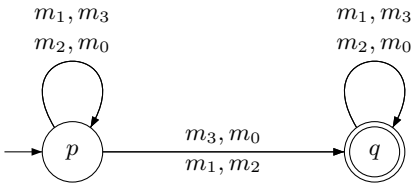


FIG. 7.7 – L'automate représentant $pre^*(\langle p, m_3\Gamma^*\rangle)$

Par exemple, la règle de saturation nous permet de rajouter la règle de transition $p \xrightarrow{m_3} p$ à l'automate de la figure 7.6, puisque $\langle p, m_3\rangle \overset{\tau}{\hookrightarrow} \langle p, \epsilon\rangle$ (règle r_5) et $p \xrightarrow{\epsilon} p$. Ensuite, nous pouvons rajouter $p \xrightarrow{m_2} p$ (règle r_4), $p \xrightarrow{m_0} p$ (règle r_2), $p \xrightarrow{m_1} q$ (règle r_3, et $p \xrightarrow{m_0 m_3} q$) etc. Le résultat final est l'automate représenté à la figure 7.7.

La seconde étape consiste à étiqueter les transitions t de $\mathcal{A}_{pre^*}$ avec des éléments $l(t) \in K$. Pour ce faire, nous associons à chaque transition t une variable x_t et nous définissons un ensemble de contraintes sur ces variables dont la plus petite solution (par rapport à $\leq$) correspond aux étiquettes $l(t)$. Dans ces contraintes, $v_\tau = \bar{1}$:

(α_1) si $t = (q, \gamma, q')$ est déjà une transition de $\mathcal{A}$, alors

$$\bar{1} \leq x_t,$$

(α_2) pour chaque règle $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma'\rangle$ et chaque $q \in Q$,

$$v_a \odot x_{(p',\gamma',q)} \leq x_{(p,\gamma,q)}$$

(α_3) pour chaque règle $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \epsilon\rangle$,

$$v_a \leq x_{(p,\gamma,p')}$$

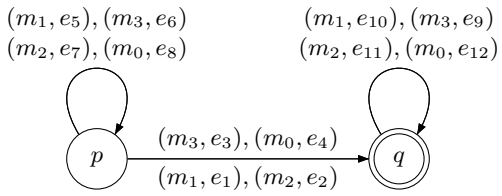
(α_4) pour chaque règle $\langle p, \gamma\rangle \overset{a}{\hookrightarrow} \langle p', \gamma_1\gamma_2\rangle$ et chaque $q \in Q$,

$$\bigoplus_{q' \in Q} (v_a \odot x_{(p',\gamma_1,q')} \odot x_{(q',\gamma_2,q)}) \leq x_{(p,\gamma,q)}$$

Formellement, si $\mathcal{A}_{pre^*} = (\Gamma, Q, \delta, P, F)$ est l'automate qui reconnaît $pre^*\big(Conf(\mathcal{A})\big)$ construit par la procédure de [BEM97, EHRS00], alors $\mathcal{A}_{pre^*_K} = (\Gamma, Q, \delta', P, F)$ où

$$\delta' = \{(q, \gamma, e, q') \mid (q, \gamma, q') \in \delta, e = l\big((q, \gamma, q')\big)\}.$$

Dans le reste de cette section nous prouvons que $\mathcal{A}_{pre^*_K}$ reconnaît $pre^*_K\big(Conf(\mathcal{A})\big)$. Dans la section suivante nous montrons comment calculer les étiquettes $l(p, \gamma, q)$ (notons que la définition que nous avons données n'est pas effective). Avant de présenter la preuve formelle, nous expliquons brièvement et intuitivement ces règles (α_1)–(α_4), en se basant sur notre exemple.



$$
\begin{aligned}
(x_4 \odot x_9) \oplus (x_8 \odot x_3) &\leq x_1 \\
v_a \odot x_3 &\leq x_2 \\
\bar{1} &\leq x_3 \\
(v_a \odot x_1) \oplus (v_b \odot x_2) &\leq x_4 \\
x_8 \odot x_6 &\leq x_5 \\
\bar{1} &\leq x_6 \\
v_a \odot x_6 &\leq x_7 \\
(v_b \odot x_7) \oplus (v_a \odot x_5) &\leq x_8 \\
\bar{1} &\leq x_9 \\
\bar{1} &\leq x_{10} \\
\bar{1} &\leq x_{11} \\
\bar{1} &\leq x_{12}
\end{aligned}
$$

FIG. 7.8 – Système d'inégalités pour le K-automate représentant $pre^*_K(\langle p, m_3\Gamma^*\rangle)$

- La règle (α_1) exprime que si c est une configuration de $Conf(\mathcal{A})$, alors $(c, \bar{1})$ doit être accepté par $\mathcal{A}_{pre^*_K}$,
- Les règles (α_2), (α_3), et (α_4) expriment que pour chaque règle $r = (p, \gamma) \overset{a}{\hookrightarrow} (p', u)$, si $(\langle p', uw\rangle, \pi)$ est un K-prédécesseur de $Conf(\mathcal{A})$, alors $(\langle p, \gamma w\rangle, v_a \odot \pi)$ l'est aussi.

La figure 7.8 montre l'étiquetage obtenu pour l'automate de la figure 7.7, où nous avons numéroté les transitions de 1 à 12, et où les variables x_i correspondent à x_{t_i}. Nous expliquons brièvement quelques unes de ces inégalités. L'inégalité $v_a \odot x_3 \leq x_2$ exprime que si π est une expression de chemin de K telle que $(\langle p, m_3\rangle, \pi)$ est un K-prédécesseur de $(\langle p, m_3\Gamma^*\rangle, \bar{1})$, alors $(\langle p, m_2\rangle, v_a \odot \pi)$ l'est aussi (règle r_4). Puisque les transitions $p \xrightarrow{m_3} q$ et $p \xrightarrow{m_2} q$ sont respectivement étiquetées par e_3 et e_2, il s'en suit que

$$\pi \leq e_3 \Rightarrow v_a \odot \pi \leq e_2.$$

Par conséquent, $v_a \odot e_3 \leq e_2$.

De manière similaire, l'inégalité $(x_4 \odot x_9) \oplus (x_8 \odot x_3) \leq x_1$ exprime que si π est une expression de chemin de K telle que $(\langle p, m_0m_3\rangle, \pi)$ est un K-prédécesseur de $(\langle p, m_3\Gamma^*\rangle, \bar{1})$, alors $(\langle p, m_1\rangle, \pi)$ l'est aussi (règle r_3). Puisqu'il y a deux chemins menant

de p à q par $m_0 m_3$ (qui sont respectivement étiquetés par $e_4 \odot e_9$ et $e_8 \odot e_3$), il s'en suit que si $\pi \leq e_8 \odot e_3$ ou $\pi \leq e_4 \odot e_9$, alors $\pi \leq e_1$ (puisque e_1 est l'étiquette du chemin $p \xrightarrow{m_1} q$), ce qui veut dire que $e_8 \odot e_3 \leq e_1$ et $e_4 \odot e_9 \leq e_1$, c-à-d., $(e_4 \odot e_9) \oplus (e_8 \odot e_3) \leq e_1$.

Nous sommes maintenant prêts pour énoncer et démontrer notre théorème principal :

Théorème 7.5.1 *Etant donnés un PDS $\mathcal{P}$ et un ensemble régulier de configurations reconnu par un automate $\mathcal{A}$, le K-automate $\mathcal{A}_{pre^*_K}$ décrit ci-dessus reconnaît $pre^*_K\big(Conf(\mathcal{A})\big)$.*

Ce théorème est une conséquence immédiate des lemmes 7.5.1 et 7.5.2 que nous montrons ci-dessous.

Lemme 7.5.1 *Pour chaque configuration $\langle p, v\rangle \in Conf(\mathcal{A})$, si $\big(\langle p, v\rangle, \bar{1}\big) \Leftarrow_K \big(\langle p', w\rangle, \pi\big)$ pour une expression de chemins $\pi \in \Pi_K$, alors il existe $e \in K$ tel que $\pi \leq e$ et $p' \xrightarrow{(w,e)} q$ pour un état final q de $\mathcal{A}_{pre^*_K}$.*

Preuve :

Pour montrer ce lemme, il suffit de montrer que si $\big(\langle p, v\rangle, \bar{1}\big) \Leftarrow_K \big(\langle p', w\rangle, \pi\big)$ pour une expression de chemins $\pi \in \Pi_K$, alors il existe un état final q tel que $\rho = p' \xrightarrow{w} q$ est un chemin de $\mathcal{A}_{pre^*}$ et $\pi \leq \lambda\big((p', w, q)_\rho\big)$.

Nous écrivons $\big(\langle p, v\rangle, \bar{1}\big) \overset{k}{\Leftarrow}_K \big(\langle p', w\rangle, \pi\big)$ pour exprimer que $\big(\langle p', w\rangle, \pi\big)$ est un K-prédécesseur de $\big(\langle p, v\rangle, \bar{1}\big)$ obtenu après k étapes. Nous procédons par induction sur k :

- $k = 0$. Alors $p' = p$, $w = v$, et $\pi = \bar{1}$. Puisque $\langle p, v\rangle \in Conf(\mathcal{A})$, il existe un état final q tel que $\rho = p \xrightarrow{v} q$ est un chemin de $\mathcal{A}$ (et donc de $\mathcal{A}_{pre^*}$). Par conséquent, la règle (α_1) implique que $\bar{1} \leq \lambda\big((p, v, q)_\rho\big)$.
- $k > 0$. Soit alors $p'' \in P, u \in \Gamma^*, \pi' \in \Pi_K$ tel que

$$\big(\langle p, v\rangle, \bar{1}\big) \overset{k-1}{\Leftarrow}_K \big(\langle p'', u\rangle, \pi'\big) \overset{1}{\Leftarrow}_K \big(\langle p', w\rangle, \pi\big)$$

  Par hypothèse d'induction, nous déduisons qu'il existe un état final q tel que $\rho = p'' \xrightarrow{u} q$ est une exécution de $\mathcal{A}_{pre^*}$ et $\pi' \leq \lambda\big((p'', u, q)_\rho\big)$.

  Puisque $\big(\langle p'', u\rangle, \pi'\big) \overset{1}{\Leftarrow}_K \big(\langle p', w\rangle, \pi\big)$, il existe $\gamma \in \Gamma, w_1, u_1 \in \Gamma^*$ et une règle $r = (p', \gamma) \overset{a}{\hookrightarrow} (p'', u_1)$ de Δ tels que $w = \gamma w_1$, $u = u_1 w_1$, et $\pi = v_a \odot \pi'$. Soit alors $q' \in Q$ tel que $\rho = p'' \xrightarrow{u_1} q' \xrightarrow{w_1} q$. Soit $\rho_1 = p'' \xrightarrow{u_1} q'$ la première partie de ρ, et $\rho_2 = q' \xrightarrow{w_1} q$ sa seconde partie. Nous obtenons que $\pi' \leq \lambda\big((p'', u_1, q')_{\rho_1}\big) \odot \lambda\big((q', w_1, q)_{\rho_2}\big)$. D'après la règle de saturation, il s'en suit qu'il existe une transition $p' \xrightarrow{\gamma} q'$ dans $\mathcal{A}_{pre^*}$, ce qui veut dire que $\rho' = p' \xrightarrow{\gamma} q' \xrightarrow{w_1} q$ est un chemin dans $\mathcal{A}_{pre^*}$. A partir des règles $(\alpha_2), (\alpha_3)$, et (α_4), nous déduisons que

$$v_a \odot \lambda\big((p'', u_1, q')_{\rho_1}\big) \leq \lambda\big((p', \gamma, q')\big)$$

Par conséquent, nous avons que $\rho' = p' \xrightarrow{w} q$ est un chemin de $\mathcal{A}_{pre^*}$ (puisque $\gamma w_1 = w$), où q est un état final, et

$$\begin{aligned}\pi &= v_a \odot \pi' \\ &\leq v_a \odot \lambda\big((p'', u_1, q')_{\rho_1}\big) \odot \lambda\big((q', w_1, q)_{\rho_2}\big) \\ &\leq \lambda\big((p', \gamma, q')\big) \odot \lambda\big((q', w_1, q)_{\rho_2}\big) \\ &= \lambda\big((p', w, q)_{\rho'}\big)\end{aligned}$$

□

Lemme 7.5.2 *Soit $p \xrightarrow{(w,e)} q$ un chemin de $\mathcal{A}_{pre^*_K}$, alors pour chaque expression de chemins $\pi \leq e$, il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin dans $\mathcal{A}$, et $\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p, w \rangle, \pi\big)$.*

Preuve : Nous montrons par induction sur $|\pi|$ que si $\pi \leq \lambda\big((p, w, q)_\rho\big)$ pour un chemin $\rho = p \xrightarrow{w} q$ de $\mathcal{A}_{pre^*}$, alors il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin de $\mathcal{A}$ et $\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p, w \rangle, \pi\big)$.

Soit $\gamma \in \Gamma$, $w_1 \in \Gamma^*$, et $q' \in Q$ tels que $w = \gamma w_1$ et

$$\rho = p \xrightarrow{\gamma} q' \xrightarrow{w_1} q.$$

Soit π une expression de chemin telle que

$$\pi \leq \lambda\big((p, w, q)_\rho\big).$$

- $|\pi| = 0$, c-à-d., $\pi = \bar{1}$. Alors nécessairement, selon les inégalités (α_1), nous avons déjà $p \xrightarrow{w} q$ dans $\mathcal{A}$. Par conséquent, la propriété est vraie avec $p' = p$ et $w' = w$.
- $|\pi| > 0$. Alors π satisfait $\pi \leq \lambda\big((p, \gamma, q')\big) \odot \lambda\big((q', w_1, q)_{\rho_1}\big)$, où $\rho_1 = q' \xrightarrow{w_1} q$ est la dernière partie de ρ. Puisque $\pi \neq \bar{1}$, $\lambda\big((p, \gamma, q')\big) \neq \bar{1}$ (puisque la procédure de saturation rajoute des règles de transition qui partent des états initiaux, et l'automate initial ne contient pas d'arêtes menant vers des états initiaux), et il existe une règle $r = (p, \gamma) \overset{a}{\hookrightarrow} (p_1, u)$ de Δ et deux expressions de chemin π_1 et π_2 telles que $\rho' = p_1 \xrightarrow{u} q' \xrightarrow{w_1} q$ est un chemin de $\mathcal{A}_{pre^*}$, $\pi = v_a \odot \pi_1 \odot \pi_2$, $v_a \odot \pi_1 \leq \lambda\big((p, \gamma, q')\big)$, $\pi_1 \leq \lambda\big((p_1, u, q')\big)$, et $\pi_2 \leq \lambda\big((q', w_1, q)_{\rho_1}\big)$.
  Il s'en suit que

$$\begin{aligned}\pi_1 \odot \pi_2 &\leq \lambda\big((p_1, u, q')\big) \odot \lambda\big((q', w_1, q)_{\rho_1}\big) \\ &= \lambda\big((p_1, uw_1, q)_{\rho'}\big)\end{aligned}$$

  Puisque $|\pi_1 \odot \pi_2| < |\pi|$, l'hypothèse d'induction implique qu'il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin dans $\mathcal{A}$, et $\big(\langle p', w' \rangle, \bar{1}\big) \Leftarrow_K \big(\langle p_1, uw_1 \rangle, \pi_1 \odot \pi_2\big)$. En plus, en appliquant r nous obtenons que $(\langle p_1, uw_1 \rangle, \pi_1 \odot$

$\pi_2\big) \Leftarrow_K \big(\langle p, \gamma w_1\rangle, v_a \odot \pi_1 \odot \pi_2\big)$. Par conséquent, puisque $w = \gamma w_1$ et $\pi = v_a \odot \pi_1 \odot \pi_2$, nous obtenons

$$\big(\langle p', w'\rangle, \bar{1}\big) \Leftarrow_K \big(\langle p, w\rangle, \pi\big)$$

où $\langle p', w'\rangle$ est tel que $p' \xrightarrow{w'} q$ est un chemin dans $\mathcal{A}$.

□

7.5.2 Résolution du système d'inégalités

Dans cette section nous considérons le problème de calculer l'étiquette $l(t)$ pour chaque transition t, c-à-d., de résoudre le système d'inégalités donné par (α_1)–(α_4).

Soient $t_1, t_2, \ldots, t_m$ des numéros arbitraires des transitions de $\mathcal{A}_{pre^*}$, et soient $x_1, \ldots, x_m$ leurs variables associées. Alors, $l(t_1), \ldots, l(t_m)$ est la plus petite solution d'un système d'inégalités de la forme

$$f_i(x_1, \ldots, x_m) \leq x_i, \quad 1 \leq i \leq m \tag{7.3}$$

où les f_i sont des monômes dans $K[x_1, \ldots, x_m]$, l'algèbre de Kleene des polynômes sur K dont les indéterminés sont $x_1, \ldots, x_m$. En effet, il suffit d'observer que deux différentes inégalités de la forme $e_1 \leq x_{(p,\gamma,q)}$ et $e_2 \leq x_{(p,\gamma,q)}$ peuvent être remplacées par l'inégalité $e_1 \oplus e_2 \leq x_{(p,\gamma,q)}$.

Nous montrons comment résoudre ce système pour les deux classes d'abstractions que nous considérons.

7.5.2.1 Abstractions à chaînes finies

Soient $\mathbf{X} = (x_1, \ldots, x_m)$ et F la fonction

$$F(\mathbf{X}) = \big(f_1(x_1, \ldots, x_m), \ldots, f_m(x_1, \ldots, x_m)\big).$$

La plus petite solution de (8.3) est le plus petit pré-point fixe de F. Il est facile de montrer que F est monotone et $\oplus$-continue. Par conséquent, le théorème de Tarski implique que le plus petit pré-point fixe de F existe et est égal à son plus petit point fixe, et par le théorème de Kleene, ce point fixe est égal à :

$$\bigoplus_{i \geq 0} F^i(\bar{\mathbf{0}})$$

Donc, puisque K ne contient pas de chaîne ascendante infinie, un calcul itératif du plus petit point fixe termine toujours. Notons que la longueur des chaînes ascendantes n'est pas nécessairement bornée. Quand une longueur maximale existe, par ex., dans le cas des abstractions finies, elle donne une borne supérieure sur le nombre d'itérations de l'algorithme.

7.5.2.2 Abstractions commutatives

Dans ce cas $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ est une algèbre de Kleene commutative. Le système d'inégalités (8.3) peut être résolu en utilisant la procédure élégante donnée par Hopkins et Kozen dans [HK99]. Pour estimer la complexité de notre algorithme, nous rappelons la procédure de [HK99] brièvement. Pour ce faire, nous avons besoin de quelques notions préliminaires.

Soit $\mathbf{x} = (x_1, \ldots, x_n)$ un vecteur de n inconnus, soit $\mathbf{a} = (a_1, \ldots, a_n)$ un vecteur d'éléments de K, et soit $\mathbf{f} = f_1, \ldots, f_l$ un vecteur de polynômes à inconnus $\mathbf{x}$ et à coefficients dans K. Nous écrivons $\mathbf{f}(\mathbf{a})$ pour la valeur de $\mathbf{f}$ évaluée en $\mathbf{a}$. Le *jacobien* de $\mathbf{f}$, $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{x})$, est la matrice $l \times n$ dont le $(i,j)^{éme}$ élément est $\frac{\partial f_i}{\partial x_j}(\mathbf{x})$, où $\frac{\partial}{\partial x_i}$ est *l'opérateur différentiel* défini par

- $\frac{\partial x_i}{\partial x_i} = \bar{1}$, $\frac{\partial x_j}{\partial x_i} = \bar{0}$ pour $i \neq j$, et $\frac{\partial a}{\partial x_i} = \bar{0}$ pour $a \in K$.
- $\frac{\partial}{\partial x_i}(f \oplus g) = \frac{\partial f}{\partial x_i} \oplus \frac{\partial g}{\partial x_i}$
- $\frac{\partial}{\partial x_i}(f \odot g) = (f \odot \frac{\partial g}{\partial x_i}) \oplus (\frac{\partial f}{\partial x_i} \odot g)$
- $\frac{\partial}{\partial x_i}(f^\star) = f^\star \odot \frac{\partial f}{\partial x_i}$

Alors, la plus petite solution de (8.3) est le point fixe de la chaîne $\mathbf{a_0} \leq \mathbf{a_1} \leq \mathbf{a_2} \ldots$[3] définie par

$$\begin{aligned} \mathbf{a_0} &= \mathbf{f}(\bar{\mathbf{0}}) \\ \mathbf{a_{k+1}} &= \frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{a_k})^\star \odot \mathbf{a_k}, \end{aligned}$$

Donc, pour calculer $\mathbf{a_{k+1}}$ nous évaluons le jacobien au vecteur $\mathbf{a_k}$, nous obtenons ainsi une matrice sur K. Ensuite, nous calculons la clôture de Kleene de cette matrice[4] en utilisant par exemple l'algorithme de Floyd-Warshall. Finalement, nous calculons le produit de la matrice résultante et du vecteur $\mathbf{a_k}$.

7.5.3 Exemple

Nous donnons la solution du système d'inégalités de la figure 7.8 dans le cas où $(K, \oplus, \odot, \star, \bar{0}, \bar{1})$ est une algèbre de Kleene commutative. Nous appliquons l'algorithme donné dans [HK99], où, par exemple, $f_1(\mathbf{x})$ est le monôme $(x_4 \odot x_9) \oplus (x_8 \odot x_3)$, et $f_4(\mathbf{x})$ est le monôme $(v_a \odot x_1) \oplus (v_b \odot x_2)$. Le jacobien $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}$ est une matrice 12×12 dont les éléments sont $\bar{0}$, v_a, v_b, ou x_i. Par exemple, le $(1,3)^{éme}$ élément de cette matrice est x_8 (puisque $\frac{\partial f_1}{\partial x_3} = x_8$), et le $(2,3)^{éme}$ élément est v_a (puisque $\frac{\partial f_2}{\partial x_3} = v_a$). La plus petite solution, donnée par $e_1 = e_5 = e_8 = v_a^\star \odot v_a \odot v_b$, $e_3 = e_6 = e_9 = e_{10} = e_{11} = e_{12} = \bar{1}$, $e_2 = e_7 = v_a$, et $e_4 = (v_a \odot v_a^\star \odot v_a \odot v_b) \oplus (v_a \odot v_b)$.

Il s'en suit que la configuration $\langle p, m_0 \rangle$ est étiquetée par e_4, exprimant que le langage $Paths(\langle p, m_0 \rangle, \langle p, m_3 \Gamma^* \rangle)$ est approximé par $e_4 = (v_a \odot v_a^\star \odot v_a \odot v_b) \oplus (v_a \odot v_b)$.

Pour les trois abstractions commutatives que nous avons considérées, nous obtenons :

[3] Le fait que cette séquence soit une chaîne découle facilement des axiomes d'une algèbre de Kleene.
[4] La clôture de Kleene d'une matrice A sur K est définie par $A^* = \bigoplus_{i \geq 0} A^i$.

– Dans le cadre de l'abstraction "les ensembles d'actions interdites et nécessaires", $v_a = [\{b\}, \{a\}]$ et $v_b = [\{a\}, \{b\}]$, ce qui veut dire que $e_4 = [\emptyset, \{a, b\}]$. Ceci montre que partant de m_0, a et b sont tous les deux nécessaires pour atteindre le point m_3.
– Dans le cadre de l'abstraction "label bitvectors", $v_a = 10$, $v_b = 01$, et $e_4 = 11$. Ceci exprime que nous devons exécuter au moins un a et un b pour aller de m_0 à m_3.
– Dans le cadre de l'abstraction "image de Parikh" , $v_a = \binom{1}{0}$, $v_b = \binom{0}{1}$, et $e_4 = \{\binom{k}{1} \mid k \geq 1\}$. Ceci exprime que les chemins menant de m_0 à m_3 contiennent un nombre arbitraire (≥ 1) de a, et exactement un b.

7.5.4 Complexité de la procédure

Nous déduisons à partir de (α_1)–(α_4) que le système de contraintes a $O(|\Delta||Q|+|\delta|)$ inégalités sur $O(|\Delta||Q| + |\delta|)$ variables, où Q et δ sont les ensembles d'états et de transitions de $\mathcal{A}$, et Δ est l'ensemble de transitions de l'automate à pile. En effet, l'automate $\mathcal{A}_{pre^*}$ contient les transitions de l'automate initial (δ), et les transitions rajoutées par la règle de saturation. Pour chaque règle de l'automate, la règle de saturation rajoute au plus $|Q|$ transitions. Nous aurons donc au total $O(|\Delta||Q| + |\delta|)$ transitions. De plus, une transition t de l'automate initial qui ne peut pas être rajoutée par la procédure de saturation est étiquetée par $\overline{1}$ (règles α_1), c-à-d., $l(t) = \overline{1}$. Reste alors à déterminer les étiquettes des transitions qui sont vraiment rajoutées par la procédure, c-à-d., de $O(|\Delta||Q|)$ transitions. Le problème est donc de résoudre un système ayant $O(|\Delta||Q|)$ inégalités et $O(|\Delta||Q|)$ variables.

7.5.4.1 Le cas des abstractions à chaînes finies

Nous avons $O(|\Delta||Q|)$ inégalités de la forme

$$f_i(x_1, \ldots, x_m) \leq x_i,$$

où les f_i sont des polynômes ayant $O(|Q|)$ opérations. Et donc le point fixe peut être atteint après $O\big(h|\Delta||Q|\big)$ étapes, où h est la longueur maximale d'une chaîne dans le treillis abstrait : Il s'agit de partir des $f_i(\overline{0}, \ldots, \overline{0})$, et de mettre à jour à chaque étape l'étiquette correspondant à une variable. Comme chaque variable peut être mise à jour au plus h fois, nous avons en tout $O\big(h|\Delta||Q|\big)$ étapes. A chaque étape, il faut effectuer $O(|Q|)$ opérations, et donc le coût total est $O(hc|\Delta||Q|^2)$, où c est le coût d'une opération du treillis.

Nous étudions avec plus de détails les complexités des différentes abstractions à chaînes finies que nous avons considérées :
Les ensembles d'actions interdites et nécessaires : Puisque $[F_1, R_1] \leq [F_2, R_2]$ si et seulement si $F_2 \subseteq F_1$ et $R_2 \subseteq R_1$, et puisque pour chaque $[F, R] \in K$ nous avons $F, R \subseteq Lab$ et $F \cap R = \emptyset$, la plus longue chaîne de K a au plus une longueur égale à $|Lab|$. Si les ensembles de lettres sont implémentés comme des vecteurs de bits, les opérations $\oplus$ et $\odot$ sont calculées en appliquant les opérations OU et ET bit pat bit. Le coût de ces opérations est $O(|Lab|)$. Donc, l'algorithme s'exécute en un temps en $O(|Lab| \cdot |\Delta||Q|^2)$.

Label bitvectors : Dans ce cas, l'ordre $\leq$ est simplement l'inclusion entre les ensembles, et donc la plus longue chaîne de K a $2^{|Lab|}$ éléments. Une opération prend un temps en $O(2^{|Lab|})$. Par conséquent, le temps d'exécution du calcul itératif est $2^{O(|Lab|)} \cdot |\Delta||Q|^2$.
Ordre des premières occurrences : Comme dans le cas précédent , l'ordre $\leq$ est l'inclusion entre les ensembles. La plus longue chaîne de K a $O(2^{|Lab|!})$ éléments. Une opération prend un temps en $O(2^{|Lab|!})$. Par conséquent, le temps d'exécution du calcul itératif est $2^{O(|Lab|!)} \cdot |\Delta||Q|^2$.

7.5.4.2 Le cas commutatif

La complexité est dominée par l'algorithme de Hopkins-Kozen [HK99], nous en donnons juste une estimation. La matrice du jacobien a une dimension $m \times m$, où m est le nombre de variables du système d'inégalités. Comme observé précédemment, $m \in O(|\Delta||Q|)$. La clôture de Kleene de $\frac{\partial \mathbf{f}}{\partial \mathbf{x}}(\mathbf{a_k})$ peut donc être calculée en $O(m^3)$ $\oplus/\odot/\star$-opérations en utilisant par exemple l'algorithme de Floyd-Warshall.

Il est montré dans [HK99] que pour une algèbre de Kleene commutative *arbitraire*, le point fixe de la chaîne $(\mathbf{a_i})$ est atteint après au plus $O(3^m)$ itérations. Puisque chaque itération nécessite de calculer une clôture de Kleene, la taille des expressions peut croître exponentiellement à chaque itération. Donc, la taille maximale d'une expression peut être double exponentielle en m, et donc le coût d'une opération est aussi double exponentiel en m. Ceci est aussi le facteur dominant dans la complexité totale.

7.5.4.3 Conclusion

De cette analyse, nous déduisons que l'analyse par l'abstraction des ensembles d'actions interdites et nécessaires a un coût faible, celle qui considère l'ordre des premières occurrences des lettres ou les label bitvectors ont des coûts acceptables s'il y a peu de signaux de synchronisation, et l'analyse par image de Parikh peut avoir un coût très élevé dans le pire des cas. Il y a donc un compromis à faire entre la précision de l'abstraction et le coût de l'analyse.

Notons que puisque les abstractions des ensembles d'actions interdites et nécessaires et des label bitvectors sont à la fois des abstractions finies et commutatives, nous pouvons appliquer soit un calcul itératif du plus petit point fixe, soit l'algorithme de Hopkins et Kozen. Observons que le nombre d'itérations de ce dernier algorithme est toujours borné, alors que dans le cas du calcul itératif, ce nombre dépend de la longueur maximale des chaînes du treillis. L'algorithme de [HK99] calcule une sorte d'accélération de point fixe en introduisant l'opération "$\star$" à chaque itération. Ceci peut être vu comme des étapes d'accélération. Ces accélérations peuvent être utiles pour les domaines à chaînes finies ou les chaînes peuvent être très longues.

7.6 Calcul des K-successeurs

Nous présentons dans cette section une procédure qui calcule les images $post^*_K$. Elle est légérement plus compliquée que la procédure pour le calcul des images pre^*, mais l'idée principale est la même. Même si la complexité de calculer les images $post^*_K$

est plus importante, il s'avère que les successeurs sont plus utiles dans plusieurs cas pratiques. En particulier, les K-successeurs de la configuration initiale constituent une abstraction de tous les chemins d'exécution du programme, qui est utile dans plusieurs cas.

Etant donné un ensemble régulier de configurations C reconnu par un automate $\mathcal{A} = (\Gamma, Q_0, \delta_0, P, F)$, nous construisons un K-automate $\mathcal{A}_{post^*_K}$ qui reconnaît $post^*_K(C)$. Comme dans la section précédente, nous supposons sans perte de généralité que $\mathcal{A}$ n'a pas de transitions menant à un état initial.

Notre algorithme est en deux étapes. D'abord, nous construisons un automate $\mathcal{A}_{post^*} = (\Gamma, Q, \delta, P, F)$ qui reconnaît $post^*(C)$, et ensuite nous étiquetons les transitions de cet automate avec des éléments de K pour obtenir $\mathcal{A}_{post^*_K}$.

Pour la première étape nous appliquons la procédure de [EHRS00]. Cette procédure construit un automate avec des ϵ-transitions $\mathcal{A}_{post^*}$, en calculant itérativement la séquence finie $\mathcal{A}_i$ définie comme suit :

- $\mathcal{A}_0$ est l'automate obtenu à partir de $\mathcal{A}$ en rajoutant pour chaque règle de transition $r \in \Delta$ de la forme $(p, \gamma) \overset{a}{\hookrightarrow} (p', \gamma'\gamma'')$ un nouvel état r et une nouvelle transition (p', γ', r),
- pour chaque règle $(p, \gamma) \overset{a}{\hookrightarrow} (p', \epsilon)$ et chaque état q tel que $p \overset{\gamma}{\rightsquigarrow}_i q$, rajouter la nouvelle transition $p' \xrightarrow{\epsilon}_{i+1} q$,
- pour chaque règle $(p, \gamma) \overset{a}{\hookrightarrow} (p', \gamma')$ et chaque état q tel que $p \overset{\gamma}{\rightsquigarrow}_i q$, rajouter la nouvelle transition $p' \xrightarrow{\gamma'}_{i+1} q$,
- pour chaque règle $(p, \gamma) \overset{a}{\hookrightarrow} (p', \gamma'\gamma'')$ et chaque état q tel que $p \overset{\gamma}{\rightsquigarrow}_i q$, rajouter la nouvelle transition $r \xrightarrow{\gamma''}_{i+1} q$,

où $\rightarrow_i$ est la relation de transition de $\mathcal{A}_i$, et $\overset{\gamma}{\rightsquigarrow}$ dénote la relation $\xrightarrow{\epsilon}{}^* \circ \xrightarrow{\gamma} \circ \xrightarrow{\epsilon}{}^*$. Observons que l'automate initial $\mathcal{A}$ ne contient pas de transitions menant à un état initial, et la procédure décrite ci-dessus ne rajoute pas de telles transitions. En plus, toutes les ϵ-transitions rajoutées par cette procédure partent d'un état initial. Par conséquent, $\overset{\gamma}{\rightsquigarrow} = \xrightarrow{\gamma} \cup \xrightarrow{\epsilon} \circ \xrightarrow{\gamma}$.

Ensuite, la seconde étape consiste à étiqueter les transitions t de $\mathcal{A}_{post^*}$ avec des éléments $h(t) \in K$ définis comme les plus petits éléments satisfaisant les inégalités suivantes :

(β_1) si $t = (q, \gamma, q')$ est déjà une transition de $\mathcal{A}_0$, alors $\bar{1} \leq h(t)$,

(β_2) pour chaque règle $r = (p, \gamma) \overset{a}{\hookrightarrow} (p', \gamma')$ et chaque état q,

$$v_a \odot h'\big((p, \gamma, q)\big) \leq h\big((p', \gamma', q)\big)$$

(β_3) pour chaque règle $r = (p, \gamma) \overset{a}{\hookrightarrow} (p', \epsilon)$ et chaque état q,

$$v_a \odot h'\big((p, \gamma, q)\big) \leq h\big((p', \epsilon, q)\big)$$

(β_4) pour chaque règle $r = (p, \gamma) \overset{a}{\hookrightarrow} (p', \gamma'\gamma'')$ et chaque état q,

$$v_a \odot h'\big((p, \gamma, q)\big) \leq h\big((r, \gamma'', q)\big)$$

où $h'\big((p,\gamma,q)\big)$ dénote la somme des étiquettes de tous les chemins de $\mathcal{A}_{post^*}$ de la forme $p \overset{\gamma}{\rightsquigarrow} q$, c-à-d. (puisque $\overset{\gamma}{\rightsquigarrow} = \xrightarrow{\gamma} \cup \xrightarrow{\epsilon} \circ \xrightarrow{\gamma}$) :

$$h'\big((p,\gamma,q)\big) = h\big((p,\gamma,q)\big) \oplus \bigoplus_{q' \in Q} \Big(h\big((p,\epsilon,q')\big) \odot h\big((q',\gamma,q)\big) \Big)$$

Intuitivement, les inégalités définissant h expriment que, pour chaque règle $r = (p,\gamma) \overset{a}{\hookrightarrow} (p',u)$, si $(\langle p, \gamma w\rangle, \pi)$ est un K-successeur de $\mathit{Conf}(\mathcal{A})$, alors $(\langle p', uw\rangle, v_a \odot \pi)$ l'est aussi.

Pour calculer les étiquettes $h(t)$ pour chaque transition t, nous procédons comme dans la section précédente. Nous considérons un numérotage arbitraire des transitions de $\mathcal{A}_{post^*} : t_1, t_2, \ldots, t_m$. Soient alors $x_1, \ldots, x_m$ les variables associées à ces transitions. A partir de (β_1)–(β_4), nous avons que $h(t_1), \ldots, h(t_m)$ est la plus petite solution d'un système d'inégalités de la forme

$$g_i(x_1, \ldots, x_m) \leq x_i, \qquad 1 \leq i \leq m \tag{7.4}$$

où les g_i sont des monômes dans $K[x_1, \ldots, x_m]$. Ces inégalités peuvent être résolues dans les cas des abstractions à chaînes finies et des abstractions commutatives de la même manière que pour le cas précédent du calcul des K-prédécesseurs.

Si $\mathcal{A}_{post^*} = (\Gamma, Q, \delta, P, F)$ est l'automate défini dans [EHRS00] qui reconnaît $post^*\big(\mathit{Conf}(\mathcal{A})\big)$, alors l'automate construit est défini par $\mathcal{A}_{post^*_K} = (\Gamma, Q, \delta', P, F)$ où

$$\delta' = \{(q,\gamma,e,q') \mid (q,\gamma,q') \in \delta, e = h\big((q,\gamma,q')\big)\}.$$

Nous montrons que cet automate reconnaît $post^*_K\big(\mathit{Conf}(\mathcal{A})\big)$:

Théorème 7.6.1 *Etant donnés un PDS $\mathcal{P}$ et un ensemble régulier de configurations reconnu par un automate $\mathcal{A}$, la procédure ci-dessus construit un K-automate $\mathcal{A}_{post^*_K}$ qui reconnaît $post^*_K\big(\mathit{Conf}(\mathcal{A})\big)$.*

Ce théorème est la conséquence immédiate des deux lemmes suivants :

Lemme 7.6.1 *Pour chaque configuration $\langle p, v\rangle \in \mathit{Conf}(\mathcal{A})$, si $\big(\langle p, v\rangle, \bar{1}\big) \Rightarrow_K \big(\langle p', w\rangle, \pi\big)$ alors il existe $e \in K$ tel que $\pi \leq e$ et $p' \overset{(w,e)}{\longrightarrow} q$ pour un état final q de $\mathcal{A}_{post^*_K}$.*

Preuve : Soit $\langle p, v\rangle \in \mathit{Conf}(\mathcal{A})$. Nous montrons que si $\big(\langle p, v\rangle, \bar{1}\big) \Rightarrow_K \big(\langle p', w\rangle, \pi\big)$ alors il existe un état final q tel que $\rho = p' \xrightarrow{w} q$ est un chemin de $\mathcal{A}_{post^*}$ et $\pi \leq \lambda\big((p', w, q)_\rho\big)$.

Nous écrivons $\big(\langle p, v\rangle, \bar{1}\big) \overset{k}{\Rightarrow}_K \big(\langle p', w\rangle, \pi\big)$ si $\big(\langle p', w\rangle, \pi\big)$ est un K-successeur de $\big(\langle p, v\rangle, \bar{1}\big)$ après k étapes. Nous procédons par induction sur k.

- $k = 0$. Alors $p' = p$, $w = v$, et $\pi = \bar{1}$. Puisque $\langle p, v\rangle \in \mathit{Conf}(\mathcal{A})$, il existe un état final q tel que $\rho = p \xrightarrow{v} q$ est déjà un chemin de $\mathcal{A}$ (et donc de $\mathcal{A}_{post^*}$). Par conséquent, les règles (β_1) impliquent que $\bar{1} \leq \lambda\big((p, v, q)_\rho\big)$.

– $k > 0$. Soit alors $p'' \in P, u \in \Gamma^*, \pi' \in \Pi_K$ tel que

$$\big(\langle p, v\rangle, \bar{1}\big) \stackrel{k-1}{\Rightarrow}_K \big(\langle p'', u\rangle, \pi'\big) \stackrel{1}{\Rightarrow}_K \big(\langle p', w\rangle, \pi\big)$$

Par hypothèse d'induction, il existe un chemin $\rho' = p'' \xrightarrow{u} q$ de $\mathcal{A}_{post^*}$ tel que q est un état final et $\pi' \leq \lambda\big((p'', u, q)_{\rho'}\big)$.

Puisque $\big(\langle p'', u\rangle, \pi'\big) \stackrel{1}{\Rightarrow}_K \big(\langle p', w\rangle, \pi\big)$, il existe $\gamma \in \Gamma, w_1, u_1 \in \Gamma^*$ et une règle $r = (p'', \gamma) \stackrel{a}{\hookrightarrow} (p', w_1)$ de Δ tels que $u = \gamma u_1$, $w = w_1 u_1$, et $\pi = v_a \odot \pi'$. Soit alors $q' \in Q$ tel que $\rho' = p'' \stackrel{\gamma}{\rightsquigarrow} q' \xrightarrow{u_1} q$. Soit $\rho_1 = p'' \stackrel{\gamma}{\rightsquigarrow} q'$ et $\rho_2 = q' \xrightarrow{u_1} q$ les deux parties de ρ'. Il y a deux cas selon que $|w_1| < 2$ ou $|w_1| = 2$:

– $|w_1| < 2$. A partir des règles de saturation, il s'en suit qu'il y a une transition $p' \xrightarrow{w_1} q'$ dans $\mathcal{A}_{post^*}$, ce qui veut dire que $\rho = p' \xrightarrow{w_1} q' \xrightarrow{u_1} q$ est un chemin de $\mathcal{A}_{post^*}$. A partir des règles (β_2) et (β_3), il s'en suit que

$$v_a \odot \lambda\big((p'', \gamma, q')_{\rho_1}\big) \leq \lambda\big((p', w_1, q')\big)$$

Par conséquent, puisque $w_1 u_1 = w$, $\rho = p' \xrightarrow{w} q$ est un chemin acceptant de $\mathcal{A}_{post^*}$, et

$$\begin{array}{rcl} \pi & = & v_a \odot \pi' \\ & \leq & v_a \odot \lambda\big((p'', u, q)_{\rho'}\big) \\ & = & v_a \odot \lambda\big((p'', \gamma, q')_{\rho_1}\big) \odot \lambda\big((q', u_1, q)_{\rho_2}\big) \\ & \leq & \lambda\big((p', w_1, q')\big) \odot \lambda\big((q', u_1, q)_{\rho_2}\big) \\ & = & \lambda\big((p', w, q)_{\rho}\big) \end{array}$$

– $|w_1| = 2$, soit alors $w_1 = \gamma'\gamma''$. Il s'en suit qu'il existe un chemin

$$p' \xrightarrow{\gamma'} r \xrightarrow{\gamma''} q'$$

dans $\mathcal{A}_{post^*}$, ce qui veut dire que $\rho = p' \xrightarrow{\gamma'} r \xrightarrow{\gamma''} q' \xrightarrow{u_1} q$ est un chemin dans $\mathcal{A}_{post^*}$. Les règles (β_1) impliquent que $\bar{1} \leq \lambda\big((p', \gamma', r)\big)$, et à partir des règles (β_4), nous obtenons que

$$v_a \odot \lambda\big((p'', \gamma, q')_{\rho_1}\big) \leq \lambda\big((r, \gamma'', q')\big)$$

Par conséquent, $\rho = p' \xrightarrow{w} q$ est un chemin acceptant de $\mathcal{A}_{post^*}$ (puisque $\gamma'\gamma'' u_1 = w_1 u_1 = w$) et π satisfait :

$$\begin{array}{rcl} \pi & = & \bar{1} \odot v_a \odot \pi' \\ & \leq & \lambda\big((p', \gamma', r)\big) \odot v_a \odot \lambda\big((p'', u, q)_{\rho'}\big) \\ & = & \lambda\big((p', \gamma', r)\big) \odot v_a \odot \lambda\big((p'', \gamma, q')_{\rho_1}\big) \odot \lambda\big((q', u_1, q)_{\rho_2}\big) \\ & \leq & \lambda\big((p', \gamma', r)\big) \odot \lambda\big((r, \gamma'', q')\big) \odot \lambda\big((q', u_1, q)_{\rho_2}\big) \\ & = & \lambda\big((p', w, q)_{\rho}\big) \end{array}$$

□

Lemme 7.6.2 *Soit $p \xrightarrow{(w,e)} q$ un chemin de $\mathcal{A}_{post^*_K}$ tel que $q \in Q_0$, alors pour chaque expression de chemin $\pi \leq e$, il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin de $\mathcal{A}$, et $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$.*

Preuve : Nous montrons par induction sur $|\pi|$ que si $\pi \leq \lambda\big((p, w, q)_\rho\big)$ pour un chemin $\rho = p \xrightarrow{w} q$ de $\mathcal{A}_{post^*}$ tel que $q \in Q_0$, alors il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin de $\mathcal{A}$ et $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$.

Soit $\rho = p \xrightarrow{w} q$ un chemin de $\mathcal{A}_{post^*}$ tel que $q \in Q_0$, et soit π une expression de chemin telle que $\pi \leq \lambda\big((p, w, q)_\rho\big)$.

- $|\pi| = 0$, c-à-d., $\pi = \bar{1}$. Alors nécessairement, selon les inégalités (β_1), $\rho = p \xrightarrow{w} q$ est un chemin de $\mathcal{A}'_0$ et donc, il est déjà un chemin de $\mathcal{A}$ puisque $\mathcal{A}'_0$ contient les transitions de $\mathcal{A}$ et les transitions de la forme $p \xrightarrow{\gamma} r$ où $r \notin Q_0$. Il s'en suit que dans $\mathcal{A}'_0$ il n'y a pas de chemin menant à un état dans Q_0 qui passe par un état r. Le chemin ρ ne considère alors que des transitions de $\mathcal{A}$. Par conséquent, la propriété est vraie avec $p' = p$ et $w' = w$.
- $|\pi| > 0$. Il y a deux cas :
  - Il existe $\gamma \in \Gamma \cup \{\epsilon\}$, $w_1 \in \Gamma^*$, et $q' \in Q$, où $w = \gamma w_1$ et

    $$\rho = p \xrightarrow{\gamma} q' \xrightarrow{w_1} q$$

    tel que $\pi = \pi_1 \odot \pi_2$, $\pi_1 \neq \bar{1}$, $\pi_1 \leq \lambda\big((p, \gamma, q')\big)$, $\pi_2 \leq \lambda\big((q', w_1, q)_{\rho_1}\big)$, où $\rho_1 = q' \xrightarrow{w_1} q$ est la dernière partie de ρ.
    Comme, $\pi_1 \neq \bar{1}$, les inégalités (β_2) et (β_3) impliquent qu'il existe une règle $r = (p_1, \gamma_1) \overset{a}{\hookrightarrow} (p, \gamma)$ de Δ, un chemin $\rho' = p_1 \overset{\gamma_1}{\rightsquigarrow} q' \xrightarrow{w_1} q$ de $\mathcal{A}_{post^*}$, tel que $v_a \odot \lambda\big((p_1, \gamma_1, q')_{\rho_2}\big) \leq \lambda\big((p, \gamma, q')\big)$, où $\rho_2 = p_1 \overset{\gamma_1}{\rightsquigarrow} q'$ est la première partie de ce chemin. Soit donc une expression de chemin $\pi'_1 \leq \lambda\big((p_1, \gamma_1, q')_{\rho_2}\big)$ telle que $\pi_1 = v_a \odot \pi'_1$.
    Nous avons $\rho' = p_1 \overset{\gamma_1}{\rightsquigarrow} q' \xrightarrow{w_1} q$ est un chemin de $\mathcal{A}_{post^*}$, $\pi'_1 \odot \pi_2 \leq \lambda\big((p_1, \gamma_1, q')_{\rho'}\big)$, et $|\pi'_1 \odot \pi_2| < |\pi|$; donc par induction, il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin de $\mathcal{A}$ et

    $$\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p_1, \gamma_1 w_1 \rangle, \pi'_1 \odot \pi_2\big).$$

    En plus, en appliquant r nous obtenons que

    $$\big(\langle p_1, \gamma_1 w_1 \rangle, \pi_1 \odot \pi_2\big) \Rightarrow_K \big(\langle p, \gamma w_1 \rangle, v_a \odot \pi'_1 \odot \pi_2\big).$$

    Par conséquent, puisque $w = \gamma w_1$ et $\pi = v_a \odot \pi'_1 \odot \pi_2$, nous obtenons que $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$, où $\langle p', w' \rangle$ est tel que $p' \xrightarrow{w'} q$ est un chemin de $\mathcal{A}$.

– Il existe $\gamma \in \Gamma \cup \{\epsilon\}$, $w_1 \in \Gamma^*$, et $q' \in Q$, où $w = \gamma w_1$ et

$$\rho = p \xrightarrow{\gamma} q' \xrightarrow{w_1} q$$

tel que $\pi \leq \lambda\big((q', w_1, q)_{\rho_1}\big)$, où $\rho_1 = q' \xrightarrow{w_1} q$ est la dernière partie de ρ. Comme $\pi \neq \bar{1}$, que $\mathcal{A}_{post^*}$ ne contient pas de transitions menant vers un état initial, et que les seules transitions ayant une étiquette différente de $\bar{1}$ sont les transitions de la forme (s, a, q) où s est soit un état initial de P, soit un état r pour une règle r de Δ, il s'en suit qu'il existe une règle r de Δ telle que $q' = r$. De plus, puisque $\bar{1} \leq \lambda\big((p, \gamma, r)\big)$, r est de la forme $r = (p_1, \gamma_1) \overset{a}{\hookrightarrow} (p, \gamma\gamma')$ et ρ est de la forme

$$\rho = p \xrightarrow{\gamma} r \xrightarrow{\gamma'} q'' \xrightarrow{w_2} q$$

tel que $\pi = \pi_1 \odot \pi_2$, où $\pi_1 \leq \lambda\big((r, \gamma', q'')\big)$ et $\pi_2 \leq \lambda\big((q'', w_2, q)_{\rho_2}\big)$, $\rho_2 = q'' \xrightarrow{w_2} q$ étant la dernière partie de ρ. Il s'en suit d'après les règles de saturations de l'automate et les contraintes (β_4) qu'il existe dans $\mathcal{A}_{post^*}$ un chemin

$$p_1 \overset{\gamma_1}{\rightsquigarrow} q'' \xrightarrow{w_2} q$$

tel que

$$v_a \odot \lambda\big((p_1, \gamma_1, q'')_{\rho_3}\big) \leq \lambda\big((r, \gamma', q'')\big)$$

où $\rho_3 = p_1 \overset{\gamma'}{\rightsquigarrow} q''$ est la première partie du chemin. Soit donc π_1' une expression de chemin telle que $\pi_1' \leq \lambda\big((p_1, \gamma_1, q'')_{\rho_3}\big)$ et $\pi_1 = v_a \odot \pi_1'$. Nous avons $\rho' = p_1 \overset{\gamma_1}{\rightsquigarrow} q'' \xrightarrow{w_2} q$ est un chemin de $\mathcal{A}_{post^*}$ et

$$\pi_1' \odot \pi_2 \leq \lambda\big((p_1, \gamma_1 w_2, q)_{\rho'}\big)$$

Comme $|\pi_1' \odot \pi_2| < |\pi|$, nous déduisons par induction qu'il existe une configuration $\langle p', w' \rangle$ telle que $p' \xrightarrow{w'} q$ est un chemin de $\mathcal{A}$ et

$$\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p_1, \gamma_1 w_2 \rangle, \pi_1' \odot \pi_2\big).$$

En plus, en appliquant r nous obtenons que

$$\big(\langle p_1, \gamma_1 w_2 \rangle, \pi_1' \odot \pi_2\big) \Rightarrow_K \big(\langle p, \gamma\gamma' w_2 \rangle, v_a \odot \pi_1' \odot \pi_2\big)$$

c-à-d. $\big(\langle p', w' \rangle, \bar{1}\big) \Rightarrow_K \big(\langle p, w \rangle, \pi\big)$, puisque $w = \gamma\gamma' w_2$ et $\pi = v_a \odot \pi_1' \odot \pi_2$.

□

7.6.1 Complexité de la procédure

Nous déduisons à partir des règles de saturation que le système de contraintes (β_1)–(β_4) a $O\big((|\Delta|+|Q|)|\Delta|+|\delta|+|\Delta|\big)$ inégalités sur $O\big((|\Delta|+|Q|)|\Delta|+|\delta|+|\Delta|\big)$ variables, où Q et δ sont les ensembles d'états et de transitions de $\mathcal{A}$, et Δ est l'ensemble de transitions de l'automate à pile. En effet, l'automate $\mathcal{A}_{post^*}$ contient les transitions de

l'automate $\mathcal{A}_0$ ($|\delta| + |\Delta|$), et les transitions rajoutées par la procédure de saturation. Pour chaque règle de l'automate, la procédure de saturation rajoute au plus $|\Delta| + |Q|$ transitions. Nous aurons donc au total $O\big((|\Delta| + |Q|)|\Delta| + |\delta| + |\Delta|\big)$ transitions. De plus, comme dans le cas du pre^*, une transition t de l'automate $\mathcal{A}_0$ qui ne peut pas être rajoutée par la procédure de saturation est étiquetée par $\bar{1}$ (règles β_1), c-à-d., $h(t) = \bar{1}$. Reste alors à déterminer les étiquettes des transitions qui sont vraiment rajoutées par la procédure de saturation, c-à-d., de $O\big((|\Delta| + |Q|)|\Delta|\big)$ transitions. Le problème est donc de résoudre un système ayant $O\big((|\Delta| + |Q|)|\Delta|\big)$ inégalités et $O\big((|\Delta| + |Q|)|\Delta|\big)$ variables. Dans le cas où l'abstraction considérée est à chaînes finies, il est facile de voir comme dans le cas du pre^*, que le coût total de calcul de la solution du système d'inégalités est $O(hc(|\Delta| + |Q|)|\Delta||Q|)$, où c et h sont respectivement le coût d'une opération , et a longueur maximale d'une chaîne dans le treillis abstrait. Dans le cas commutatif, la même analyse faite pour le cas du pre^* s'applique pour ce cas.

7.7 Conclusion

7.7.1 Résumé

Nous avons présenté une approche générique pour l'analyse statique des programmes séquentiels concurrents (avec un nombre fixe de processus en parallèle). Nous avons modélisé ces programmes de manière *précise* en utilisant les automates à piles communicants (un automate à pile pour chaque programme séquentiel). Comme le problème de la vérification de tels programmes est indécidable [Ram00], l'analyse exacte de ces systèmes n'est pas possible. Avec notre approche, seul l'effet de la synchronisation est approximé, l'effet des procédures (de la séquentialité) est déterminé de manière précise. L'idée est d'analyser chaque système séquentiel à part en tenant compte des actions de synchronisation : si le système contient deux programmes séquentiels concurrents, alors ces derniers peuvent accéder à une certaine configuration ssi ils effectuent une séquence d'actions de synchronisation commune. Ceci revient à décider le vide de l'intersection de langages hors-contextes, puisque le langage des chemins d'un automate à pile est un langage hors-contexte. Ce problème étant indécidable, nous calculons des abstractions (sur-approximations) de ces langages de chemins pour lesquelles le problème du vide d'une intersection est décidable. Pour calculer ces abstractions, nous posons un système de contraintes qui caractérise l'ensemble des chemins d'exécution d'un automate à pile, et nous le résolvons dans des domaines abstraits. Nous proposons quatre abstractions différentes qui permettent de faire des analyses à différents coûts et précisions.

7.7.2 Autre application

Notre technique peut également être appliquée dans le cadre de l'analyse interprocédurale des programmes séquentiels, pour résoudre le problème de propagation de constantes lorsque les affectations présentes dans le programme sont linéaires (de la forme $y := ax + b$) [JRS03]. Ce problème consiste à déterminer, à un certain point de contrôle, les valeurs possibles que peut prendre une certaine variable. Pour résoudre

ce problème, Jha, Reps, et Schwoon modélisent un programme séquentiel par un automate à pile dont chaque règle a un poids (qui est une valeur dans un certain domaine abstrait D). Chaque poids code une fonction d'affectation relative à la transition du programme représentée par la règle en question. En d'autres termes, chaque instruction du programme est représentée par une règle de l'automate à pile. Si l'instruction n'est pas une affectation, le poids de la règle est 1 (l'élément neutre de la concaténation dans le domaine abstrait), sinon, ce poids code la fonction d'affectation de l'instruction.

Le problème est alors réduit au calcul de l'accumulation des effets des règles de l'automate à pile qui peuvent mener du point initial I du programme à un ensemble de configurations C. Si l'accumulation de l'effet de deux fonctions d'affectation est codée dans le domaine abstrait par une opération de concaténation des poids, le problème revient à calculer la concaténation (dans le domaine abstrait) des poids des séquences des règles qui mènent de I à C. Ceci peut donc être calculé par notre algorithme si nous interprétons les étiquettes des règles des automates à pile comme des poids.

Dans leur article, Reps et al. ont indépendamment proposé un algorithme qui calcule des abstractions des chemins d'exécutions d'un automate à pile. Leur algorithme est aussi basé sur le calcul de $pre^*(C)$ par la méthode de [BEM97, EHRS00]. Leur technique étiquette les transitions de cet automate au fur et à mesure de sa construction. Elle ne peut cependant être appliquée que dans le cadre des abstractions à chaînes finies. Dans ce cadre, notre technique a la même complexité que la leur.

7.7.3 Comparaison avec d'autres travaux

Dans le cadre de l'analyse statique des programmes, nous trouvons dans la littérature plusieurs travaux qui traitent les programmes constitués d'un nombre fini fixé de processus finis concurrents. Ces travaux ne considèrent pas la récursivité. Dans notre cadre, ce cas correspond à des automates finis communicants, au lieu des automates à pile communicants. L'ensemble des chemins d'exécutions est alors régulier, et donc le problème du vide de l'intersection de deux ensembles de chemins d'exécutions est décidable. Dans ce cas, l'approximation est utilisée pour réduire le coût du calcul. Dans [NA98], une technique basée sur la définition de "produit faible" d'automates finis a été introduite pour éviter le problème d'explosion combinatoire. L'approche de Corbett [Cor92], dans laquelle les ensembles des chemins du programme sont approximés en utilisant la programmation linéaire est moins précise que notre approximation par cacul d'images de Parikh, mais elle a un moindre coût. Dans [Mer91], Mercouroff propose une approche basée sur le comptage des différentes actions de communication qui se sont produites dans les différents canaux du système. Les approximations calculées sont moins précises que la nôtre (celle des images de Parikh) puisqu'elles sont basées sur un calcul de point fixe qui termine en utilisant des techniques d'extrapolation peu précises.

A notre connaissance, le seul travail qui considère à la fois la récursivité et la synchronisation est le travail de [DS91]. L'article fournit une technique approximative pour déterminer les actions concurrentes : si les actions a et b sont concurrentes, alors il est possible d'exécuter a avant b ou b avant a. La méthode de [DS91] consiste à calculer pour chaque action a un ensemble d'actions B (resp. A) qui contient les actions qui ne peuvent apparaître qu'avant (resp. après) l'action a lors d'une exécution. Ce problème

peut être formulé dans notre cadre comme suit : [5]

Le treillis abstrait a comme éléments des paires (Lab', D), où $Lab' \subseteq Lab$ et $D\colon Lab' \times Lab' \to \{\mathbf{none}, \mathbf{12}, \mathbf{21}, \mathbf{both}\}$. Etant donné un langage de chemins $Paths(C', C)$, nous avons $\alpha(Paths(C', C)) = (Lab', D)$, où Lab' est l'ensemble des actions qui apparaissent dans $Paths(C', C)$, et

- $D(a, b) = \mathbf{none}$ si ab et ba ne sont pas des sous mots de $Paths(C', C)$;
- $D(a, b) = \mathbf{12}$ si ab est un sous mot de $Paths(C', C)$, mais ba ne l'est pas ;
- $D(a, b) = \mathbf{21}$ si ba est un sous mot de $Paths(C', C)$, mais ab ne l'est pas ;
- $D(a, b) = \mathbf{both}$ si ab et ba sont tous les deux des sous mots de $Paths(C', C)$.

Il est facile de définir des opérations $\oplus$ et $\odot$ correspondant à l'union et la concaténation des langages. De même, nous pouvons définir l'opération $\sqcap$ correspondant à l'intersection. Puisque le treillis est fini, l'approximation peut être calculée de manière effective. Si nous obtenons que $D(a, b) \neq \mathbf{both}$, nous en déduisons que a et b ne peuvent pas être concurrentes dans un chemin qui mène de C' à C. Notons que dans ce cas, le but n'est pas de déterminer si l'intersection des abstractions des langages de chemins est vide, mais de la calculer.

L'avantage majeur de notre approche par rapport à celle de [DS91] est sa généricité, puisqu'elle peut s'appliquer dans plusieurs cadres et à plusieurs instances d'abstractions. Notons cependant que la création dynamique de processus est permise dans [DS91], alors qu'elle ne l'est pas dans notre cadre. Nous allons voir dans le chapitre suivant comment étendre ces techniques pour tenir compte du dynamisme.

[5] Cette traduction est dûe à Javier Esparza et est parue dans [BET03a, BET03b].

Chapitre 8

Extension au cas dynamique

Dans le chapitre précédent, nous avons considéré les programmes constitués d'un nombre fixe de processus séquentiels qui tournent en parallèle. Le dynamisme n'est pas autorisé dans ces programmes. Dans ce chapitre, nous étendons les résultats précédents au cas des programmes avec création dynamique de processus. Nous introduisons un troisième modèle (SPA) qui permet de modéliser de manière *précise* dynamisme et synchronisation, sans cependant tenir compte des variables de retour des procédures appelées. Nous présentons des techniques de calcul d'abstractions des langages de chemins $Paths(\mathcal{L}, \mathcal{L}')$ de ces systèmes. L'approche que nous développons est, comme précédemment, basée sur (1) la caractérisation de l'ensemble $Paths(\mathcal{L}, \mathcal{L}')$ comme la plus petite solution d'un système de contraintes, et (2) la résolution de ce système dans des domaines abstraits.

Le contenu de ce chapitre fait l'objet du rapport de recherche [BET03c].

8.1 Le modèle : systèmes PA synchronisés

Nous introduisons dans cette section notre troisième modèle de description des programmes parallèles récursifs. Ce modèle est une sorte d'extension des systèmes PA "à-la CCS" [Mil80] par des opérateurs de synchronisation et de restriction.

8.1.1 Syntaxe

Soit Var un ensemble de variables de processus, et $\mathcal{T}$ l'ensemble des termes de processus sur Var. Soit $Lab = \{a, b, c, \ldots\}$ un ensemble des actions visibles. Soit $Sync$ et $Async$ deux ensembles disjoints tels que $Lab = Sync \cup Async$, et pour chaque action $a \in Sync$ correspond une co-action $\bar{a}$ dans $Sync$ telle que $\bar{\bar{a}} = a$. Soit $Act = Lab \cup \{\tau\}$ l'ensemble de toutes les actions, où τ est une action interne spéciale (comme nous allons le voir, cette action spéciale représentera les rendez-vous (handshakes)).

Définition 8.1.1 *Un* système PA synchronisé *(SPA pour Synchronized PA) est un ensemble fini de règles de la forme* $X \overset{a}{\hookrightarrow} t$*, où* $t \in \mathcal{T}$ *et* $a \in Lab$*.*

Nous définissons l'ensemble des termes de processus restreints comme suit :

$$\mathcal{T}_r = \{t\backslash S \mid t \in \mathcal{T}, S \subseteq Sync\}.$$

Intuitivement, le terme "$t\backslash S$" correspond à la restriction du comportement du terme t aux actions qui ne sont pas dans S. Nous considérons que $t\backslash\emptyset$ est égal à t.

Soit $\mathcal{L}$ un ensemble de termes de processus et S un sous ensemble de $Sync$, nous dénotons par $\mathcal{L}\backslash S$ l'ensemble $\{t\backslash S \mid t \in \mathcal{L}\}$.

8.1.2 Sémantique

Un SPA $\mathcal{R}$ induit une relation de transition $\overset{a}{\rightarrowtail}$ sur $\mathcal{T}$ et une relation de transition $\overset{a}{\twoheadrightarrow}$ sur $\mathcal{T}_r$ définies par les règles d'inférence suivantes :

$$\theta_1 : \frac{t_1 \overset{a}{\hookrightarrow} t_2 \in \mathcal{R}}{t_1 \overset{a}{\rightarrowtail} t_2}; \quad \theta_2 : \frac{t_1 \overset{a}{\rightarrowtail} t_1'}{t_1 \cdot t_2 \overset{a}{\rightarrowtail} t_1' \cdot t_2}; \quad \theta_3 : \frac{t_1 \sim_0 0 \ , \ t_2 \overset{a}{\rightarrowtail} t_2'}{t_1 \cdot t_2 \overset{a}{\rightarrowtail} t_1 \cdot t_2'}$$

$$\theta_4 : \frac{t_1 \overset{a}{\rightarrowtail} t_1'}{t_1||t_2 \overset{a}{\rightarrowtail} t_1'||t_2 \ ; \ t_2||t_1 \overset{a}{\rightarrowtail} t_2||t_1'}; \quad \theta_5 : \frac{t_1 \overset{a}{\rightarrowtail} t_1'; t_2 \overset{\bar{a}}{\rightarrowtail} t_2'; a, \bar{a} \in Sync}{t_1||t_2 \overset{\tau}{\rightarrowtail} t_1'||t_2'};$$

$$\theta_6 : \frac{t_1 \overset{a}{\rightarrowtail} t_2; a \notin S, \bar{a} \notin S}{t_1\backslash S \overset{a}{\twoheadrightarrow} t_2\backslash S}$$

Nous définissons de manière standard les relations $\overset{w}{\rightarrowtail}$ et $\overset{w}{\twoheadrightarrow}$, pour $w \in Act^*$.

Chaque équivalence $\equiv$ de $\{\sim, =\}$ induit une relation de transition $\overset{a}{\rightarrowtail}_{\equiv}$ sur $\mathcal{T}$ et une relation de transition $\overset{a}{\twoheadrightarrow}_{\equiv}$ sur $\mathcal{T}_r$ définies par :

$$\forall t, t' \in \mathcal{T}, t \overset{a}{\rightarrowtail}_{\equiv} t' \text{ ssi } \exists u, u' \text{ tels que } t \equiv u, u \overset{a}{\rightarrowtail} u', \text{ et } u' \equiv t'$$

et

$$\forall t, t' \in \mathcal{T}, t\backslash S \overset{a}{\twoheadrightarrow}_{\equiv} t'\backslash S \text{ ssi } \exists u, u' \text{ tels que } t \equiv u, u\backslash S \overset{a}{\twoheadrightarrow} u'\backslash S, \text{ et } u' \equiv t'$$

Ces relations sont étendues aux séquences d'actions de manière usuelle.

Soient pour $t \in \mathcal{T}$,

$$Post^*_{\rightarrowtail,\equiv}[w](t) = \{t' \in \mathcal{T} \mid t \overset{w}{\rightarrowtail}_{\equiv} t'\},$$

et

$$Post^*_{\rightarrowtail,\equiv}(t) = \bigcup_{w \in Act^*} Post^*_{\rightarrowtail,\equiv}[w](t).$$

Et pour $t \in \mathcal{T}_r$,

$$Post^*_{\twoheadrightarrow,\equiv}[w](t) = \{t' \in \mathcal{T}_r \mid t \overset{w}{\twoheadrightarrow}_{\equiv} t'\},$$

et

$$Post^*_{\twoheadrightarrow,\equiv}(t) = \bigcup_{w \in Act^*} Post^*_{\twoheadrightarrow,\equiv}[w](t).$$

Parfois, nous écrivons simplement $Post^*_{\rightarrowtail}[w](t)$, $Post^*_{\rightarrowtail}(t)$, $Post^*_{\twoheadrightarrow}[w](t)$, et $Post^*_{\twoheadrightarrow}(t)$ pour désigner respectivement $Post^*_{\rightarrowtail,\sim}[w](t)$, $Post^*_{\rightarrowtail,\sim}(t)$, $Post^*_{\twoheadrightarrow,\sim}[w](t)$, et $Post^*_{\twoheadrightarrow,\sim}(t)$.

Ces définitions s'étendent aux ensembles de termes de $\mathcal{T}$ et $\mathcal{T}_r$ de manière standard.

8.1.3 SPA vs. PA

Observons que syntaxiquement, un SPA correspond à un système PA tel que défini dans la section 4.3.5. De même, la relation de transition $\stackrel{a}{\Rightarrow}_{\equiv}$ induite par les règles d'inférence θ_1–θ_4 correspond à la sémantique d'un système PA. Soient alors

$$Post^*_{PA,\equiv}[w](t) = \{t' \in \mathcal{T} \mid t \stackrel{w}{\Rightarrow}_{\equiv} t'\}$$

et

$$Post^*_{PA,\equiv}(t) = \bigcup_{w \in Act^*} Post^*_{PA,\equiv}[w](t).$$

Par rapport aux notations introduites aux chapitre 4 pour désigner les ensembles des accessibles par un système PA, nous rajoutons ici l'indice "PA" pour distinguer entre un système SPA et son PA correspondant.

Il est facile de voir que pour tout $t \in \mathcal{T}$, les termes accessibles à partir de t par la relation $\rightarrowtail_{\equiv}$ le sont aussi par $\stackrel{a}{\Rightarrow}_{\equiv}$:

Proposition 8.1.1 *Pour tout $t \in \mathcal{T}$ et $\equiv \in \{\sim, =\}$, nous avons :*

$$Post^*_{\rightarrowtail,\equiv}(t) = Post^*_{PA,\equiv}(t).$$

Preuve : Ceci est dû au fait que les termes obtenus par application de (θ_5) peuvent aussi être obtenus par application de (θ_4). □

8.1.4 Passage d'un PFG à un SPA

Soit un programme donné par un parallel flow graph $G = \{G_p \mid p \in \texttt{Proc}\}$. Nous définissons un SPA qui modélise ce programme. Le système obtenu modélise la synchronisation mais oublie les variables globales et les résultats de retour des procédures appelées. Cet SPA est défini comme suit :

Ensemble des actions : $Lab = (Stmt \setminus \{\textbf{skip}\}) \cup \{\epsilon\}$ est l'ensemble des actions du programme plus l'action silencieuse ϵ, tel que $Sync = \{a, \bar{a} \mid a \in Sig\}$ et

$$Async = \{b \in Stmt \mid \nexists a \in Sig,\ b = a! \text{ ou } b = a?\}.$$

Nous posons par définition que pour tout $a \in Sig$, $\bar{\bar{a}} = a$.

Ensemble des variables de processus : Var est l'ensemble des paires (n, loc) pour chaque nœud n du PFG, et chaque valuation possible loc des variables locales.

Règles : Le SPA contient les règles suivantes, pour chaque Π dans `Proc` :

Instruction vide : $(n_1, loc) \stackrel{\epsilon}{\hookrightarrow} (n_2, loc)$, si $n_1 \rightarrow n_2 \in E_\Pi$;

Affectation : $(n_1, loc) \stackrel{a}{\hookrightarrow} (n_2, loc')$, si $n_1 \stackrel{a}{\rightarrow} n_2 \in E_\Pi$ et a est une affectation, où loc et loc' sont les valeurs des variables locales de la procédure Π avant et après l'affectation ;

Conditionnelle "if-then-else" : $(n_1, loc) \overset{a}{\hookrightarrow} (n_2, loc)$, si $n_1 \overset{a}{\to} n_2 \in E_\Pi$ et a est une conditionnelle "if-then-else", où loc est tel que la condition de l'instruction est satisfaite ;

Appel récursif : si $n_1 \xrightarrow{call(p)} n_2 \in E_\Pi$, alors considérer la règle

$$(n_1, loc) \overset{\epsilon}{\hookrightarrow} (e_p, loc_p) \cdot (n_2, loc),$$

où e_p est le point d'entrée de la procédure p, loc_p les valeurs des arguments passées par la procédure Π à la procédure p, et loc mémorise les variables locales de la procédure appelante Π ;

Terminaison : $(r_\Pi, loc) \overset{\epsilon}{\hookrightarrow} 0$, où r_Π est le point de sortie de la procédure Π.

Création de processus : Si $n_1 \xrightarrow{call(p_1||\cdots||p_k)} n_2 \in E_\Pi$, alors considérer la règle

$$(n_1, loc) \overset{\epsilon}{\hookrightarrow} \big((e_{p_1}, loc_1)|| \cdots ||(e_{p_k}, loc_k)\big) \cdot (n_2, loc),$$

où e_{p_i} est le point d'entrée de la procédure p_i, loc_i les valeurs des arguments passées par la procédure Π à p_i, et loc mémorise les variables locales de la procédure appelante Π ;

Synchronisation :

- Si $n_1 \overset{a!}{\to} n_1' \in E_\Pi$, considérer la règle $(n_1, loc) \overset{a}{\hookrightarrow} (n_1', loc)$,
- Si $n_1 \overset{a?}{\to} n_1' \in E_\Pi$, considérer la règle $(n_1, loc) \overset{\bar{a}}{\hookrightarrow} (n_1', loc)$.

Observons qu'avec cette modélisation, les règles θ_5 permettent de modéliser la synchronisation avec *exactitude* : si les processus t_1 et t_2 sont tous les deux actifs, alors si t_1 envoie le signal a et que t_2 le reçoive, ils se synchronisent et avancent tous les deux. Cependant, les règles θ_4 permettent à chaque processus d'avancer sans attendre le processus avec lequel il devrait se synchroniser. Ces règles introduisent donc des comportements supplémentaires dans le système. Pour éviter ceci, une façon de faire est de n'appliquer les règles θ_4

$$\theta_4 : \frac{t_1 \overset{a}{\rightarrowtail} t_1'}{t_1||t_2 \overset{a}{\rightarrowtail} t_1'||t_2 \quad ; \quad t_2||t_1 \overset{a}{\rightarrowtail} t_2||t_1'}$$

que si a n'est pas une action de synchronisation, c-à-d., que si $a \in Async$. Avec la sémantique de SPA, ceci revient à considérer des termes de processus restreints $t\backslash Sync$, empêchant ainsi la mauvaise application des règles θ_4. En effet, les règles θ_6 n'autorisent à un processus d'avancer indépendamment de son entourage que si l'action qu'il doit faire n'est pas une action de synchronisation, c-à-d., si elle ne correspond ni à une émission ni à une réception d'un signal. Donc, pour calculer l'ensemble des accessibles par le programme d'un ensemble de termes de processus $\mathcal{L}$, il suffit de calculer

$$Post^*_{\twoheadrightarrow}(\mathcal{L}\backslash Sync).$$

8.2 Le problème d'accessibilité pour SPA

Soient $\mathcal{R}$ un SPA, et $\mathcal{L}$ et $\mathcal{L}'$ sont deux langages de termes de processus sur $\mathcal{T}$. Le problème que nous nous proposons de résoudre est celui de vérifier s'il existe des termes de processus $t_1 \in \mathcal{L}$ et $t_2 \in \mathcal{L}'$ tels que t_2 peut être atteint à partir de t_1. Comme expliqué dans le section précédente, ceci revient à vérifier si

$$Post^*_{\twoheadrightarrow}(\mathcal{L}\backslash Sync) \cap \mathcal{L}'\backslash Sync \stackrel{?}{=} \emptyset. \tag{8.1}$$

Malheureusement, nous montrons que le problème d'arrêt d'une machine à deux compteurs peut être réduit à ce problème :

Théorème 8.2.1 *Le problème d'accessibilité (8.1) est indécidable pour SPA.*

Preuve : Nous codons le problème d'arrêt d'une machine à deux compteurs. La réduction est similaire à celle présentée dans [Hab98] utilisée pour montrer l'indécidabilité du Model-checking de LTL pour les systèmes PA.

Soit $\mathcal{M}$ une machine à deux compteurs ayant m états. Soit
- $Var = \{X, Y, X_1^1, X_2^1, X_1^2, X_2^2\} \cup \{X_i^c \mid 1 \leq i \leq m\}$,
- $Sync = \{i_1, i_2, \bar{i}_1, \bar{i}_2, d_1, d_2, \bar{d}_1, \bar{d}_2, z_1, z_2, \bar{z}_1, \bar{z}_2\}$,
- $Async = \{halt, a\}$.

Soit $\mathcal{R}$ le SPA ayant les règles suivantes, où $j \in \{1, 2\}$ et $k \in \{1, \ldots, m\}$:

1. $X \stackrel{a}{\hookrightarrow} X_1^1 || X_1^2 || X_1^c$,
2. $X_1^j \stackrel{z_j}{\hookrightarrow} X_1^j$,
3. $X_1^j \stackrel{i_j}{\hookrightarrow} X_2^j \cdot X_1^j$,
4. $X_2^j \stackrel{i_j}{\hookrightarrow} X_2^j \cdot X_2^j$,
5. $X_2^j \stackrel{d_j}{\hookrightarrow} 0$,
6. $X_k^c \stackrel{\bar{i}_j}{\hookrightarrow} X_h^c$, si $(s_k : c_j := c_j + 1;\ \text{goto } s_h)$ est une instruction,
7. $X_k^c \stackrel{\bar{z}_j}{\hookrightarrow} X_{h_1}^c$ et $X_k^c \stackrel{\bar{d}_j}{\hookrightarrow} X_{h_2}^c$ si l'instruction suivante existe :

$$(s_k :\ \text{si } c_j = 0 \text{ aller à } s_{h_1} \text{ sinon } c_j := c_j - 1;\ \text{aller à } s_{h_2})$$

8. $X_m \stackrel{halt}{\hookrightarrow} Y$.

Intuitivement, la variable X_1^j représente la valeur 0 du compteur j, et le terme $X_2^j \cdots X_2^j \cdot X_1^j$ avec n X_2^j représente la valeur n du compteur j. La deuxième règle simule le test d'égalité à 0 du compteur c_j, les troisième et quatrième règles représentent l'incrémentation de c_j, et la cinquième règle simule la décrémentation de c_j. Les trois dernières règles simulent le contrôle fini de la machine. La dernière règle simule l'arrêt de la machine. Les actions i_j et $\bar{i}_j$ répresentent l'incrémentation de c_j. La synchronisation entre ces deux co-actions impose que les compteurs ne peuvent être incrémentés que si le contrôle le permet. Cette même intuition est valable pour les autres actions où d_j et $\bar{d}_j$ représentent la décrémentation de c_j, et z_j et $\bar{z}_j$ le test à zéro de c_j.

Alors il est clair que $\mathcal{M}$ s'arrête ssi

$$Post^*_{\twoheadrightarrow}(X \backslash Sync) \cap \mathcal{L} \backslash Sync \neq \emptyset,$$

où $\mathcal{L}$ est l'ensemble des termes de la forme $t_1||t_2||Y$, où t_j est un terme de la forme $X_2^j \cdots X_2^j \cdot X_1^j$ qui représente la valeur du compteur j.

□

Pour résoudre le problème (8.1), nous le traduisons en un problème sur les chemins d'exécution du système. Soient $\mathcal{L}$ et $\mathcal{L}'$ deux ensembles de termes de processus sur $\mathcal{T}$. Nous définissons l'ensemble des séquences d'actions $Paths(\mathcal{L}, \mathcal{L}')$ par :

$$Paths(\mathcal{L}, \mathcal{L}') = \{w \in Act^* \mid \exists t \in \mathcal{L}, t' \in \mathcal{L}', t' \in Post^*_{\rightarrow}[w](t)\}.$$

Cet ensemble correspond à tous les chemins d'exécution qui peuvent mener à un terme dans $\mathcal{L}'$ en partant d'un terme dans $\mathcal{L}$ sans se restreindre aux actions qui ne sont pas dans $Sync$, et en appliquant les règles (θ_4) même quand l'action a est une action de synchronisation. Si nous restreignons le comportement des termes aux actions qui ne sont pas dans $Sync$, les chemins d'exécution qui permettent d'atteindre $\mathcal{L}' \backslash Sync$ en partant de $\mathcal{L} \backslash Sync$ est égal à $Paths(\mathcal{L}, \mathcal{L}') \cap (Async \cup \{\tau\})^*$. Par conséquent, le problème (8.1) est équivalent à vérifier si

$$Paths(\mathcal{L}, \mathcal{L}') \cap (Async \cup \{\tau\})^* \stackrel{?}{=} \emptyset, \tag{8.2}$$

En effet, les règles de synchronisation (θ_5) font que deux processus se synchronisent *correctement* ssi ils exécutent l'action τ. Donc toute occurrence d'une action a de $Sync$ dans $Paths(\mathcal{L}, \mathcal{L}')$ correspond à une mauvaise application des règles (θ_4). Plus précisément, elle correspond à l'exécution par un processus d'une action de synchronisation de $Sync$ sans consulter les autres processus concurrents. Et donc le problème revient à trouver un chemin qui mène de $\mathcal{L}$ à $\mathcal{L}'$ où ces actions n'appraissent pas, c-à-d. à tester 8.2.

Seulement, cet ensemble de séquences d'actions $Paths(\mathcal{L}, \mathcal{L}')$ ne peut pas être calculé à cause du résultat d'indécidabilité ci-dessus. Comme dans le chapitre précédent, notre approche consiste alors à calculer une *abstraction* du langage des chemins $Paths(\mathcal{L}, \mathcal{L}')$, c-à-d. une sur-approximation $A(\mathcal{L}, \mathcal{L}')$ de $Paths(\mathcal{L}, \mathcal{L}')$. Si $A(\mathcal{L}, \mathcal{L}') \cap (Async \cup \{\tau\})^*$ est vide, nous pouvons conclure que $Paths(\mathcal{L}, \mathcal{L}') \cap (Async \cup \{\tau\})^*$ l'est aussi.

Nous définissons une approche générique pour calculer des abstractions de $Paths(\mathcal{L}, \mathcal{L}')$ basée sur (1) la caractérisation de l'ensemble $Paths(\mathcal{L}, \mathcal{L}')$ comme la plus petite solution d'un système de contraintes sur des langages de mots (cette solution ne peut pas être calculée en général), et (2) le calcul de la plus petite solution dans un domaine abstrait pour obtenir une sur-approximation de l'ensemble $Paths(\mathcal{L}, \mathcal{L}')$.

8.3 Caractérisation des langages de chemin

Nous expliquons ci-dessous l'approche que nous adoptons pour caractériser

$Paths(\mathcal{L}, \mathcal{L}')$ comme la plus petite solution d'un système de contraintes sur des langages de mots. Suivant l'approche adoptée dans le chapitre 5, (1) nous représentons les termes de processus de $\mathcal{T}$ par des arbres binaires où les variables de processus sont les constantes et les opérateurs "$\cdot$" et "$||$" sont binaires, (2) nous utilisons les automates d'arbres binaires pour représenter les ensembles réguliers de termes de processus, et (3) nous nous restreignons au cas où $\mathcal{L}'$ est $\sim$-compatible, et nous calculons l'ensemble des séquences d'exécution $\{w \in Act^* \mid Post^*_{\rightarrow,=}[w](\mathcal{L}) \cap \mathcal{L}' \neq \emptyset\}$. En effet, si $\mathcal{L}'$ est $\sim$-compatible alors cet ensemble est égal à $Paths(\mathcal{L}, \mathcal{L}')$:

Proposition 8.3.1 *Si $\mathcal{L}'$ est $\sim$-compatible alors*

$$Paths(\mathcal{L}, \mathcal{L}') = \{w \in Act^* \mid Post^*_{\rightarrow,=}[w](\mathcal{L}) \cap \mathcal{L}' \neq \emptyset\}$$

Preuve : L'inclusion $\supseteq$ est directe. Pour l'autre direction, il suffit de montrer par induction sur $|w|$ que si $u \overset{w}{\rightarrowtail}_{\sim} u'$, alors il existe $u'' \sim u'$ tel que $u \overset{w}{\rightarrowtail}_{=} u''$. Ceci est dû au fait que les réécritures se font toujours aux feuilles des arbres (voir la proposition 5.2.1). □

Soient alors $\mathcal{L}$ et $\mathcal{L}'$ deux ensembles réguliers de termes tels que $\mathcal{L}'$ est $\sim$-compatible. Soient $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini qui reconnaît $\mathcal{L}$, et $\mathcal{A}' = (Q', \Sigma, F', \delta')$ un automate qui reconnaît $\mathcal{L}'$. Nous supposons sans perte de généralité que Q' est tel que si $s \in Q'$, alors il existe un état s^{null} dans Q' qui reconnaît *exactement* les termes de L_s qui sont nuls, c-à-d., qui reconnaît le langage $L_s \cap \{u \in \mathcal{T} \mid u \sim_0 0\}$. L'état $s^{null^{null}}$ est bien sûr égal à s^{null}.[1]

Soient $Q^{\mathcal{R}} = \{q_t \mid t$ est un sous terme des membres des règles de $\mathcal{R}\}$, et $\delta^{\mathcal{R}}$ l'ensemble des règles de transitions suivantes :

- $X \rightarrow q_X$ si $q_X \in Q^{\mathcal{R}}$, pour $X \in Var$,
- $||(q_{t_1}, q_{t_2}) \rightarrow q_t$ si $t = ||(t_1, t_2)$ et $q_t \in Q^{\mathcal{R}}$,
- $\cdot(q_{t_1}, q_{t_2}) \rightarrow q_t$ si $t = \cdot(t_1, t_2)$ et $q_t \in Q^{\mathcal{R}}$.

Il est alors clair que pour tout terme t qui est un sous terme d'un des membres des règles de $\mathcal{R}$, $L_{q_t} = \{t\}$. Posons $\mathcal{Q} = Q \cup Q^{\mathcal{R}}$, $\Delta = \delta \cup \delta^{\mathcal{R}}$, $\mathcal{Q}' = Q' \cup Q^{\mathcal{R}}$, et $\Delta' = \delta' \cup \delta^{\mathcal{R}}$. Soient $q \in \mathcal{Q}$ et $s \in \mathcal{Q}'$. Nous définissons le langage de séquences d'exécution $\lambda(q, s)$ par :

$$\lambda(q, s) = \{w \in Act^* \mid Post^*_{\rightarrow,=}[w](L_q) \cap L_s \neq \emptyset\}.$$

Nous avons alors d'après la proposition 8.3.1 :

Proposition 8.3.2 *Si $\mathcal{L}'$ est $\sim$-compatible, alors*

$$Paths(\mathcal{L}, \mathcal{L}') = \bigcup_{\substack{q \in F \\ s \in F'}} \lambda(q, s).$$

[1] De tels états peuvent être obtenus en calculant le produit de $\mathcal{A}'$ et de l'automate qui reconnaît les termes nuls, et contenant les règles $0 \rightarrow q_{null}$, $\cdot(q_{null}, q_{null}) \rightarrow q_{null}$, et $||(q_{null}, q_{null}) \rightarrow q_{null}$. L'état (s, q_{null}) correspond alors à s^{null}.

Par conséquent, notre but dans cette section est de caractériser les ensembles $\lambda(q,s)$. Pour ce faire, nous posons un système de contraintes dont la plus petite solution correspond à ces ensembles. Pour la définition de ce système, nous avons besoin de définir des opérateurs sur les séquences d'actions qui correspondent aux opérateurs de composition séquentielle et parallèle sur les termes.

8.3.1 Des opérateurs sur les séquences d'actions

L'opérateur séquentiel correspond à la concaténation (sur les mots) des chemins, puisque si $u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$ et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](v_2)$, alors soit $u_1 \sim 0$, soit u_1 est non nul et $w_2 = \epsilon$. Dans les deux cas, $\cdot(u_1,u_2) \in Post^*_{\rightarrowtail,=}[w_1 \cdot w_2]\big(\cdot(v_1,v_2)\big)$, où $w_1 \cdot w_2$ est le mot obtenu en concaténant w_1 et w_2. Pour l'opérateur de composition parallèle, nous introduisons l'opérateur de synchronisation défini inductivement sur les mots comme suit :

$$\begin{array}{rcl} \epsilon \amalg w & = & w \amalg \epsilon = w \\ aw_1 \amalg \bar{a}w_2 & = & a(w_1 \amalg \bar{a}w_2) + \bar{a}(aw_1 \amalg w_2) + \tau(w_1 \amalg w_2) \\ aw_1 \amalg bw_2 & = & a(w_1 \amalg bw_2) + b(aw_1 \amalg w_2) \text{ si } b \neq \bar{a} \end{array}$$

Avec cette définition, nous avons que si u_1 est accessible à partir de v_1 avec la séquence d'actions w_1, et u_2 est accessible à partir de v_2 avec la séquence d'actions w_2, alors $u = ||(u_1,u_2)$ est accesssible à partir de $v = ||(v_1,v_2)$ avec une séquence d'actions de $w_1 \amalg w_2$:

Proposition 8.3.3 *Soient $u_1,u_2,v_1,v_2 \in \mathcal{T}$, et $w \in Act^*$. Alors*
$||(u_1,u_2) \in Post^*_{\rightarrowtail,=}[w]\big(||(v_1,v_2)\big)$ ssi il existe $w_1,w_2 \in Act^*$ t.q. $w \in w_1 \amalg w_2, u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](v_2)$.

Preuve : Nous montrons d'abord la direction $\Rightarrow$. Soient alors $u_1,u_2,v_1,v_2 \in \mathcal{T}$, et $w \in Act^*$ tels que

$$||(u_1,u_2) \in Post^*_{\rightarrowtail,=}[w]\big(||(v_1,v_2)\big).$$

Nous procédons par induction sur $|w|$:

- $|w| = 0$. Alors $w = \epsilon$, et la propriété est satisfaite avec $w_1 = w_2 = \epsilon, u_1 = v_1$, et $u_2 = v_2$.
- $|w| > 0$, c-à-d., $w = bw'$ pour un $b \in Act$. Soient alors $u'_1,u'_2 \in \mathcal{T}$ tels que $||(u'_1,u'_2) \in Post^*_{\rightarrowtail,=}[b]\big(||(v_1,v_2)\big)$, et $||(u_1,u_2) \in Post^*_{\rightarrowtail,=}[w']\big(||(u'_1,u'_2)\big)$. Nous déduisons par induction qu'il existe deux séquences w'_1 et w'_2 telles que $w' \in w'_1 \amalg w'_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w'_1](u'_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w'_2](u'_2)$. Il y a deux cas en fonction de la nature de b :
  1. $b \neq \tau$. Alors soit $v_1 = u'_1$ et $v_2 \overset{b}{\rightarrowtail}_= u'_2$, soit $v_2 = u'_2$ et $v_1 \overset{b}{\rightarrowtail}_= u'_1$. Les deux cas sont symétriques. Supposons par exemple que nous avons le premier cas. Soient alors $w_1 = w'_1$ et $w_2 = bw'_2$. Il est facile de voir que $w \in w_1 \amalg w_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](v_2)$.
  2. $b = \tau$. Soit alors $a \in Sync$ tel que $v_1 \overset{a}{\rightarrowtail}_= u'_1$ et $v_2 \overset{\bar{a}}{\rightarrowtail}_= u'_2$. Considérons $w_1 = aw'_1$ et $w_2 = \bar{a}w'_2$. Il est facile de voir que $w \in w_1 \amalg w_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](v_2)$.

Nous montrons maintenant l'autre direction. Soient alors $w, w_1, w_2 \in Act^*$ et $u_1, u_2, v_1, v_2 \in \mathcal{T}$ t.q. $w \in w_1 \amalg w_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](v_2)$. Nous montrons par induction sur $|w_1| + |w_2|$ que $||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w]\big(||(v_1, v_2)\big)$:

- $|w_1| + |w_2| = 0$. Alors $w = w_1 = w_2 = \epsilon$, et la propriété est satisfaite.
- $|w_1| + |w_2| > 0$. Soit $w \in w_1 \amalg w_2$, nous allons montrer que

$$||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w]\big(||(v_1, v_2)\big)$$

Soient alors w'_1, w'_2, b, et b' tels que $w_1 = bw'_1$ et $w_2 = b'w'_2$. Les cas où $w_1 = \epsilon$ (ou $w_2 = \epsilon$) sont directs puisque dans ce cas $w_1 \amalg w_2 = w_2$ ($w_1 \amalg w_2 = w_1$). Soient u'_1 et u'_2 tels que $v_1 \overset{b}{\rightarrowtail}_= u'_1$, $v_2 \overset{b'}{\rightarrowtail}_= u'_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w'_1](u'_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w'_2](u'_2)$. Il y a deux cas en fonction des natures de b et b' :

1. $b' \neq \bar{b}$. Dans ce cas nous avons $w_1 \amalg w_2 = b(w'_1 \amalg w_2) + b'(w_1 \amalg w'_2)$. Soit alors $w' \in w'_1 \amalg w_2$ tel que $w = bw'$ (le cas où $w = b'w'$ pour un w' dans $w_1 \amalg w'_2$ est symétrique). Comme $u_1 \in Post^*_{\rightarrowtail,=}[w'_1](u'_1)$, $u_2 \in Post^*_{\rightarrowtail,=}[w_2](u_2)$, et $|w'_1| + |w_2| < |w_1| + |w_2|$, nous obtenons par induction que $||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w']\big(||(u'_1, v_2)\big)$, ce qui implique que $||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w]\big(||(v_1, v_2)\big)$.
2. $b' = \bar{b}$. Dans ce cas, nous avons $w_1 \amalg w_2 = b(w'_1 \amalg w_2) + \bar{b}(w_1 \amalg w'_2) + \tau(w'_1 \amalg w'_2)$ Soit alors $w' \in w'_1 \amalg w'_2$ tel que $w = \tau w'$ (les autres cas se traitent comme précédemment). Par induction nous déduisons que

$$||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w']\big(||(u'_1, u'_2)\big)$$

et donc nous obtenons directement que $||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w]\big(||(v_1, v_2)\big)$ en appliquant les règles θ_5 puisque $v_1 \overset{b}{\rightarrowtail}_= u'_1$ et $v_2 \overset{\bar{b}}{\rightarrowtail}_= u'_2$.

□

8.3.2 Le système de contraintes :

Nous associons aux états $q \in \mathcal{Q}$ et $s \in \mathcal{Q}'$ des variables $l(q, s)$ représentant des sous ensembles de Act^*, et nous définissons un ensemble de contraintes sur ces variables. Nous montrons que la plus petite solution de ce système correspond aux ensembles $\lambda(q, s)$. Le système de contraintes est défini comme suit :

(β_1) si $L_q \cap L_s \neq \emptyset$, alors

$$\epsilon \in l(q, s)$$

(β_2) si $q_1 \rightarrow q_2$ est une règle de Δ et que $s_1 \rightarrow s_2$ est une règle de Δ', alors

$$l(q_1, s_1) \subseteq l(q_2, s_2)$$

(β_3) si $\cdot(q_1, q_2) \rightarrow q$ est une règle de Δ et que $\cdot(s_1, s_2) \rightarrow s$ est une règle de Δ', alors

$$l(q_1, s_1^{null}) \cdot l(q_2, s_2) \subseteq l(q, s)$$

et

$$l(q_1, s_1) \subseteq l(q, s) \text{ si } L_{q_2} \cap L_{s_2} \neq \emptyset$$

(β_4) si $||(q_1, q_2) \to q$ est une règle de Δ et que $||(s_1, s_2) \to s$ est une règle de Δ', alors

$$l(q_1, s_1) \amalg l(q_2, s_2) \subseteq l(q, s)$$

(β_5) si $X \overset{a}{\hookrightarrow} t \in \mathcal{R}$, alors

$$l(q, q_X) \cdot a \cdot l(q_t, s) \subseteq l(q, s)$$

Nous expliquons ci-dessous l'intuition exprimée par ces règles. Rappelons que les variables $l(q, s)$ devraient représenter les ensembles $\lambda(q, s)$. Les règles (β_1) expriment que si $L_q \cap L_s \neq \emptyset$, alors $\epsilon \in \lambda(q, s)$. Les règles ($\beta_2$) expriment que si $L_{q_1} \subseteq L_{q_2}$ et $L_{s_1} \subseteq L_{s_2}$, alors $\lambda(q_1, s_1) \subseteq \lambda(q_2, s_2)$.

Les règles (β_3) expriment que si $\cdot(q_1, q_2) \to q$ est une règle de Δ, et $\cdot(s_1, s_2) \to s$ est une règle de Δ' , alors si $u_1 \in L_{s_1}$, $u_2 \in L_{s_2}$, $v_1 \in L_{q_1}$, et $v_2 \in L_{q_2}$ sont tels que :

- si u_1 est nul ($u_1 \sim_0 0$), et donc reconnu par s_1^{null}, et s'il est accessible à partir de v_1 en appliquant la séquence d'actions w_1 ($w_1 \in \lambda(q_1, s_1^{null})$), et u_2 est accessible à partir de v_2 en appliquant la séquence d'actions w_2 ($w_2 \in \lambda(q_2, s_2)$), alors $u = \cdot(u_1, u_2)$ (u est dans L_s) est accessible à partir de $v = \cdot(v_1, v_2)$ (v est dans L_q) en appliquant $w_1 w_2$ ($w_1 w_2 \in \lambda(q, s)$).
- si u_1 est accessible à partir de v_1 en appliquant la séquence d'actions w_1 ($w_1 \in \lambda(q_1, s_1)$), et $v_2 \in L_{q_2} \cap L_{s_2}$, alors $u = \cdot(u_1, v_2)$ (u est dans L_s) est accessible à partir de $v = \cdot(v_1, v_2)$ (v est dans L_q) en appliquant w_1 ($w_1 \in \lambda(q, s)$).

Les règles (β_4) expriment que si $||(q_1, q_2) \to q$ est une règle de Δ, et $||(s_1, s_2) \to s$ est une règle de Δ' , alors si $u_1 \in L_{s_1}$, $u_2 \in L_{s_2}$, $v_1 \in L_{q_1}$, et $v_2 \in L_{q_2}$ sont tels que u_1 est accessible à partir de v_1 en appliquant la séquence d'actions w_1 ($w_1 \in \lambda(q_1, s_1)$), et u_2 est accessible à partir de v_2 en appliquant la séquence d'actions w_2 ($w_2 \in \lambda(q_2, s_2)$), alors $u = ||(u_1, u_2)$ (u est dans L_s) est accessible à partir de $v = ||(v_1, v_2)$ (v est dans L_q) en appliquant n'importe quelle séquence d'actions w qui soit dans $w_1 \amalg w_2$ ($w_1 \amalg w_2 \subseteq \lambda(q, s)$).

Finalement, les règles (β_5) expriment que si X est accessible à partir de L_q avec la séquence d'actions w_1 ($w_1 \in \lambda(q, q_X)$), si $X \overset{a}{\hookrightarrow} t \in \mathcal{R}$, et il existe un terme u dans L_s qui peut être atteint à partir de t avec la séquence w_2 ($w_2 \in \lambda(q_t, s)$), alors u peut être atteint à partir de L_q avec la séquence $w_1 \cdot a \cdot w_2$ ($w_1 \cdot a \cdot w_2 \in \lambda(q, s)$).

8.3.3 Existence du plus petit point fixe :

Nous montrons que la plus petite solution de l'ensemble de contraintes précédent existe :

Proposition 8.3.4 *La plus petite solution de l'ensemble de contraintes (β_1)–(β_5) existe.*

En effet, soit $x_1, \ldots, x_m$ un numérotage arbitraire des variables $l(q, s)$, pour $q \in \mathcal{Q}$ et $s \in \mathcal{Q}'$. (β_1)–(β_5) est donc un système de contraintes de la forme

$$f_i(x_1, \ldots, x_m) \subseteq x_i, \quad 1 \leq i \leq m \tag{8.3}$$

où les $f_i(x_1, \ldots, x_m)$ sont des fonctions construites à partir des variables x_i et des opérateurs $\cdot$, II, et $\cup$. En effet, deux différentes inclusions de la forme $e_1 \subseteq x_i$ et $e_2 \subseteq x_i$ peuvent être remplacées par l'inclusion $e_1 \cup e_2 \subseteq x_i$.

Soit alors $\mathbf{X} = (x_1, \ldots, x_m)$, et F la fonction

$$F(\mathbf{X}) = \big(f_1(x_1, \ldots, x_m), \ldots, f_m(x_1, \ldots, x_m)\big).$$

La plus petite solution de (8.3) est le plus petit pré-point fixe de F. Soit $\mathcal{L}$ le treilli complet des langages sur Act, c-à-d., $\mathcal{L} = (2^{Act^*}, \subseteq, \cup, \cap, \emptyset, Act^*)$. Il est facile de voir que les opérateurs $\cdot$ et II sont $\cup$-continus. Il s'en suit que F est monotone et $\cup$-continue. Par conséquent, nous obtenons à partir du théorème de Tarski que le plus petit pré-point fixe de F existe et est égal à son plus petit point fixe, et par le théorème de Kleene, nous déduisons que ce point fixe est égal à :

$$\bigcup_{i \geq 0} F^i(\emptyset, \ldots, \emptyset). \tag{8.4}$$

Soit $\big(e(q,s)\big)_{\substack{q \in \mathcal{Q} \\ s \in \mathcal{Q}'}}$ cette solution. Nous montrons que pour chaque $q \in \mathcal{Q}$ et chaque $s \in \mathcal{Q}'$, $e(q,s)$ est égal au langage de chemins $\lambda(q,s)$:

Théorème 8.3.1 *Pour chaque $q \in \mathcal{Q}$ et chaque $s \in \mathcal{Q}'$,*

$$e(q,s) = \lambda(q,s).$$

Preuve : Nous montrons que pour chaque $q \in \mathcal{Q}$ et chaque $s \in \mathcal{Q}'$,

$$Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s \neq \emptyset \Leftrightarrow w \in e(q,s).$$

Nous commençons par l'implication $\Rightarrow$. Soient $q \in \mathcal{Q}$, $s \in \mathcal{Q}'$, $w \in Act^*$, et $u \in Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s$. Soit alors $v \in L_q$ tel que $u \in Post^*_{\rightarrowtail,=}[w](v)$. Nous procédons par induction sur $|w|$.

- $|w| = 0$, alors $w = \epsilon$, et $u \in L_q \cap L_s$, ce qui veut dire que $\epsilon \in e(q,s)$ (par (β_1)).
- $|w| > 0$. Il y a deux cas :

  1. La racine de v a été réécrite. Soient alors $X \overset{a}{\hookrightarrow} t \in \mathcal{R}$, $w_1, w_2 \in Act^*$ tels que
  $$X \in Post^*_{\rightarrowtail,=}[w_1](v), u \in Post^*_{\rightarrowtail,=}[w_2](t),$$
  et $w = w_1 a w_2$. Puisque $|w_1| < |w|$ et $|w_2| < |w|$, l'hypothèse d'induction implique que $w_1 \in e(q, q_X)$, et $w_2 \in e(q_t, s)$. Donc, nous obtenons par (β_5) que
  $$w = w_1 \cdot a \cdot w_2 \in e(q, q_X) \cdot a \cdot e(q_t, s) \subseteq e(q,s)$$

  2. La racine de v n'a pas été réécrite. Nous procédons par induction structurelle sur u :
  - $v = \cdot(v_1, v_2)$ et $u = \cdot(u_1, u_2)$ tels que $v_1 \in L_{q_1}$, $v_2 \in L_{q_2}$, $u_1 \in L_{s_1}$, $u_2 \in L_{s_2}$, où $\cdot(q_1, q_2) \rightarrow q$ est une règle de Δ et $\cdot(s_1, s_2) \rightarrow s$ est une règle de Δ' tels que :

– u_1 est équivalent à 0, c-à-d., $u_1 \in L_{s_1^{null}}$. Soient alors w_1, w_2 tels que $w = w_1 \cdot w_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$, et $u_2 \in Post^*[w_2](v_2)$. Par induction structurelle, nous avons que $w_1 \in e(q_1, s_1^{null})$ et $w_2 \in e(q_2, s_2)$. Nous obtenons alors par (β_3) que

$$w = w_1 \cdot w_2 \in e(q_1, s_1^{null}) \cdot e(q_2, s_2) \subseteq e(q, s).$$

– u_1 n'est pas équivalent à 0, dans ce cas $u_2 = v_2 \in L_{s_2} \cap L_{q_2}$ et $u_1 \in Post^*_{\rightarrowtail,=}[w](v_1)$. Par induction structurelle, nous avons que $w \in e(q_1, s_1)$, et par (β_3), nous obtenons que

$$w \in e(q_1, s_1) \subseteq e(q, s).$$

– $v = ||(v_1, v_2)$ et $u = ||(u_1, u_2)$. La proposition 8.3.3 implique qu'il existe w_1, w_2 t.q. $w \in w_1 \amalg w_2$, $u_1 \in Post^*_{\rightarrowtail,=}[w_1](v_1)$, et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](v_2)$. Soient $q_1, q_2 \in \mathcal{Q}$ t.q. $||(q_1, q_2) \rightarrow q$ est une règle de Δ, $v_1 \in L_{q_1}$, et $v_2 \in L_{q_2}$. En plus, soient $s_1, s_2 \in \mathcal{Q}'$ t.q. $||(s_1, s_2) \rightarrow s$ est une règle de Δ' telle que $u_1 \in L_{s_1}$ et $u_2 \in L_{s_2}$. Alors, par induction structurelle nous déduisons que $w_1 \in e(q_1, s_1)$ et $w_2 \in e(q_2, s_2)$, et grâce à (β_4), il s'en suit que

$$w \in w_1 \amalg w_2 \subseteq e(q_1, s_1) \amalg e(q_2, s_2) \subseteq e(q, s).$$

Considérons l'autre direction. Soit $w \in e(q, s)$, montrons que $Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s \neq \emptyset$. Puisque les étiquettes $e(q, s)$ sont construites par une procédure de saturation, nous considérons les séqences $\big(e_i(q, s)\big)_{0 \leq i \leq n}$ où $e_0(q, s) = \emptyset$, $e_n(q, s) = e(q, s)$, et $e_i(q, s)$ est l'étiquette obtenue après la i$^{\acute{e}me}$ itération. Nous montrons par induction sur i que si $w \in e_i(q, s)$, alors $Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s \neq \emptyset$. Le cas où $i = 0$ est direct. De même pour le cas où $i = 1$, puisque dans ce cas, c'est la règle β_1 qui s'est appliquée. Soit alors $i > 1$. Soit $w \in e_i(q, s)$, alors :

– soit il existe (q', s') tel que $w \in e_{i-1}(q', s')$, et $e_{i-1}(q', s') \subseteq e_i(q, s)$ (règles β_2). Dans ce cas, nous avons forcément que $q' \rightarrow q \in \Delta$ et $s' \rightarrow s \in \Delta'$, et donc $L_{q'} \subseteq L_q$, et $L_{s'} \subseteq L_s$. Par induction, nous avons que

$$Post^*_{\rightarrowtail,=}[w](L_{q'}) \cap L_{s'} \neq \emptyset.$$

Il s'en suit que $Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s \neq \emptyset$ puisque $L_{q'} \subseteq L_q$ et $L_{s'} \subseteq L_s$.

– soit il existe w_1, w_2 tels que $w = w_1 w_2$, et il existe q_1, s_1, q_2, s_2 tels que $\cdot(q_1, q_2) \rightarrow q \in \Delta$, $\cdot(s_1, s_2) \rightarrow s \in \Delta'$, $w_1 \in e_{i-1}(q_1, s_1^{null})$, $w_2 \in e_{i-1}(q_2, s_2)$, $w \in e_i(q, s)$ (règles β_3). Par induction, nous obtenons que

$$Post^*_{\rightarrowtail,=}[w_1](L_{q_1}) \cap L_{s_1^{null}} \neq \emptyset$$

et

$$Post^*_{\rightarrowtail,=}[w_2](L_{q_2}) \cap L_{s_2} \neq \emptyset.$$

Soient alors u_1 et u_2 tels que

$$u_1 \in Post^*_{\rightarrowtail,=}[w_1](L_{q_1}) \cap L_{s_1^{null}}$$

et

$$u_2 \in Post^*_{\rightarrowtail,=}[w_2](L_{q_2}) \cap L_{s_2}.$$

Il est alors clair que (puisque $u_1 \sim_0 0$)

$$\cdot(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s$$

– soit il existe q_1, s_1, q_2, s_2 tels que $\cdot(q_1, q_2) \to q \in \Delta$, $\cdot(s_1, s_2) \to s \in \Delta'$, $w \in e_{i-1}(q_1, s_1)$, et $L_{q_2} \cap L_{s_2} \neq \emptyset$ (règles β_3). Soit alors $u_2 \in L_{q_2} \cap L_{s_2}$. Puisque $w \in e_{i-1}(q_1, s_1)$, nous déduisons par induction qu'il existe $u_1 \in Post^*_{\rightarrowtail,=}[w](L_{q_1}) \cap L_{s_1}$. Il est alors clair que

$$\cdot(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s.$$

– soit il existe w_1, w_2 tels que $w \in w_1 \amalg w_2$, et il existe q_1, s_1, q_2, s_2 tels que $||(q_1, q_2) \to q \in \Delta$, $||(s_1, s_2) \to s \in \Delta'$, $w_1 \in e_{i-1}(q_1, s_1)$, et $w_2 \in e_{i-1}(q_2, s_2)$ (règles β_4). Par induction, nous obtenons que

$$Post^*_{\rightarrowtail,=}[w_1](L_{q_1}) \cap L_{s_1} \neq \emptyset$$

et

$$Post^*_{\rightarrowtail,=}[w_2](L_{q_2}) \cap L_{s_2} \neq \emptyset.$$

Soient alors u_1 et u_2 tels que $u_1 \in Post^*_{\rightarrowtail,=}[w_1](L_{q_1}) \cap L_{s_1}$ et $u_2 \in Post^*_{\rightarrowtail,=}[w_2](L_{q_2}) \cap L_{s_2}$. Nous obtenons par la proposition 8.3.3 que

$$||(u_1, u_2) \in Post^*_{\rightarrowtail,=}[w](L_q) \cap L_s$$

– soit il existe une règle $X \overset{a}{\hookrightarrow} t \in \mathcal{R}$, w_1, w_2 tels que $w = w_1 a w_2$, $w_1 \in e_{i-1}(q, q_X)$, $w_2 \in e_{i-1}(q_t, s)$ (règles β_5). Par induction, nous obtenons que

$$X \in Post^*_{\rightarrowtail,=}[w_1](L_q)$$

et qu'il existe $u \in L_s$ tel que

$$u \in Post^*_{\rightarrowtail,=}[w_2](t).$$

Il est alors clair que

$$u \in Post^*_{\rightarrowtail,=}[w_1 a w_2](L_q).$$

□

8.4 Abstraction des langages de chemins

8.4.1 Le cadre d'abstraction

Le calcul itératif (8.4) de la plus petite solution du système (8.3) ne termine en général pas (puisque le problème d'accessiblité est indécidable pour SPA). Donc,

pour pouvoir résoudre ce problème, nous calculons des sur-approximations des langages $\lambda(q,s)$. Pour ce faire, nous procédons comme dans le chapitre précédent : nous nous plaçons dans un treillis abstrait $\mathcal{D} = (D, \leq, \sqcup, \sqcap, \bot, \top)$ qui est compatible avec une algèbre de Kleene $\mathcal{K} = (K, \oplus, \odot, \star, \bar{0}, \bar{1})$, et nous le relions au treillis concret $\mathcal{L} = (2^{Act^*}, \subseteq, \cup, \cap, \emptyset, Act^*)$ par une connexion de Galois (α, γ), c-à-d., une fonction d'abstraction $\alpha : 2^{Act^*} \to D$, et une fonction de concrétisation $\gamma : D \to 2^{Act^*}$ telles que

$$\forall x \in 2^{Act^*},\ \forall y \in D.\ \alpha(x) \leq y \Longleftrightarrow x \subseteq \gamma(y)\,.$$

En plus, nous considérons un opérateur $\otimes$ sur K qui est associatif, commutatif, et $\oplus$-continu tel que pour tous x, y dans K, $x \otimes y \in K$. $\otimes$ est l'opérateur abstrait qui correspond à l'opérateur de synchronisation II.

Comme précédemment, nous considérons des abstractions où K est engendré par des éléments de base v_a pour chaque $a \in Act$ (v_τ est égal à $\bar{1}$). Comme nous précédemment, v_a est l'élément abstrait qui correspond au langage $\{a\}$.

Pour définir une connexion de Galois entre les domaines concret et abstrait, nous considérons une fonction α telle que $\alpha(\{\epsilon\}) = \bar{1}$, pour tout $a \in Act$, $\alpha(\{a\}) = v_a$, et pour chaque langages de mots L_1, L_2, nous avons $\alpha(L_1 \cdot L_2) = \alpha(L_1) \odot \alpha(L_2)$, $\alpha(L_1 \cup L_2) = \alpha(L_1) \oplus \alpha(L_2)$, et $\alpha(L_1 \amalg L_2) = \alpha(L_1) \otimes \alpha(L_2)$. Il s'en suit que

$$\alpha(L) = \bigoplus_{a_1 \cdots a_n \in L} v_{a_1} \odot \cdots \odot v_{a_n}.$$

La fonction de concrétisation γ est alors définie comme précédemment :

$$\gamma(x) = \{a_1 \cdots a_n \in 2^{Act^*} \mid v_{a_1} \odot \cdots \odot v_{a_n} \leq x\}.$$

Il est facile de voir que (α, γ) est une connexion de Galois entre $\mathcal{L}$ et $\mathcal{D}$. D'après la proposition 7.3.1 nous avons :

$$\forall L_1, L_2 \in 2^{Act^*}, \alpha(L_1) \sqcap \alpha(L_2) = \bot \Rightarrow L_1 \cap L_2 = \emptyset.$$

Par conséquent, pour résoudre le problème (8.2), il suffit de vérifier si

$$\alpha\big(Paths(\mathcal{L}, \mathcal{L}')\big) \sqcap \alpha\big((Async \cup \{\tau\})^*\big) \stackrel{?}{=} \bar{0}, \tag{8.5}$$

où $\alpha\big(Paths(L, L')\big)$ est la plus petite solution du système de contraintes suivant :

$$f_i^\alpha(x_1, \ldots, x_m) \leq x_i, \quad 1 \leq i \leq m, \tag{8.6}$$

où $f_i^\alpha(x_1, \ldots, x_m)$ est une expression obtenue en substituant dans $f_i(x_1, \ldots, x_m)$ les opérations $\cdot$ par $\odot$, II par $\otimes$, et $\cup$ par $\oplus$.

Pour résoudre le système (8.6), nous considérons des **abstractions à chaînes finies**, puisque dans ce cas, le calcul itératif du plus petit point fixe termine toujours.

Remarque 8.4.1 *Notons que nous ne considérons pas les abstractions commutatives comme dans le chapitre précédent parce que dans ce cas, nous devons manipuler l'opérateur $\otimes$ qui fait que nous n'avons plus affaire à un système d'inégalités polynômiales dans des algèbres de Kleene. Donc dans ce cas, les résultats de [HK99] ne s'appliquent plus, et donc nous n'avons pas de moyens pour résoudre les systèmes de contraintes 8.6.*

8.4.2 Instances d'abstractions à chaînes finies

Nous reprenons dans cette section les trois abstractions à chaînes finies définies au chapitre précédent, nous définissons l'opérateur $\otimes$ pour chaque cas, et nous montrons informellement comment ces abstractions permettent d'analyser les programmes des figures **??** dans ce nouveau cadre. Nous donnons un exemple complet qui illustre les différentes étapes de notre procédure dans la section suivante.

8.4.2.1 Les ensembles d'actions interdites et nécessaires :

Dans ce cas, $\otimes$ est défini par :

$$[F_1, R_1]\otimes[F_2, R_2] = \big[F_1\cap F_2, \big((R_1\cup R_2)\cap Async\big)\cup\{a \in R_1 \mid \bar{a} \in F_2\}\cup\{a \in R_2 \mid \bar{a} \in F_1\}\big].$$

Cette définition exprime qu'étant donnés deux langages de séquences L_1 et L_2, si les lettres de F_1 n'apparaissent dans aucune séquence de L_1 et les lettres de F_2 n'apparaissent dans aucune séquence de L_2, alors $L_1 \amalg L_2$ ne contient aucune lettre de $F_1\cap F_2$. De même, si a apparaît dans toutes les séquences de L_1 (resp. L_2) et que $a \in Async$, alors a doit apparaître dans toutes les séquences de $L_1 \amalg L_2$. Et, si $a \in Sync$ et $\bar{a} \in F_2$ (resp. $\bar{a} \in F_1$), c-à-d., $\bar{a}$ n'apparaît pas dans les mots de L_2 (resp. de L_1), alors a doit apparaître dans toutes les séquences de $L_1 \amalg L_2$. Alors que si $\bar{a} \notin F_2$ (resp. $\bar{a} \notin F_1$), elle peut apparaître dans L_2 (resp. dans L_1) et mener à la simplification de a dans $L_1 \amalg L_2$. Donc, a n'est pas forcément présente dans les mots de $L_1 \amalg L_2$.
Comme expliqué au chapitre précédent, dans ce cas, le nombre d'itérations à calculer pour arriver au point fixe est au plus $|Act|$.

Exemple : Considérons l'exemple de la figure 7.3 où deux processus sont créés en parallèles pour exécuter les procédures π_1 et π_2. Nous pouvons utiliser cette abstraction pour conclure que le programme ne peut pas arriver à une configuration où π_1 est au point de contrôle n_1 et π_2 est au point de contrôle m_2 : L'action a est nécessaire pour atteindre n_1, mais $\bar{a}$ est interdite pour atteindre m_2. Intuitivement, nous avons

$$\begin{aligned}\alpha\Big(Paths\big(||(n_0, m_0), ||(n_1, m_2)\big)\Big) &= \alpha\big(Paths(n_0, n_1)\big) \otimes \alpha\big(Paths(m_0, m_2)\big)\\ &= [\emptyset, \{a\}] \otimes [\{\bar{a}\}, \emptyset]\\ &= [\emptyset, \{a\}]\end{aligned}$$

De plus, $\alpha\big((Async \cup \{\tau\})^*\big) = [\{\bar{a}, a\}, \emptyset]$. Par conséquent,

$$\alpha\Big(Paths\big(||(n_0, m_0), ||(n_1, m_2)\big)\Big) \sqcap \alpha\big((Async \cup \{\tau\})^*\big) = \bar{0}$$

puisque $\{\bar{a}, a\} \cap \{a\} \neq \emptyset$.

8.4.2.2 Label bitvectors :

Nous définissons $\otimes$ comme suit : $b_1 \otimes b_2 = b$ tel que :
– si $a \in Async$, alors $b(a) = b_1(a) \vee b_2(a)$;

– si $a \in Sync$ et $(b_1(a) \wedge b_2(\bar{a})) \vee (b_2(a) \wedge b_1(\bar{a})) = 0$, alors $b(a) = b_1(a) \vee b_2(a)$;
– si $a \in Sync$ et $(b_1(a) \wedge b_2(\bar{a})) \vee (b_2(a) \wedge b_1(\bar{a})) = 1$, alors $b(a) \in \{b_1(a) \vee b_2(a), 0\}$;
– s'il existe $a \in Sync$, $(b_1(a) \wedge b_2(\bar{a})) \vee (b_2(a) \wedge b_1(\bar{a})) = 1$, alors :
  – si $b(a) = 0$, alors $b(\tau) = 1$;
  – si $b(a) = 1$, alors $b(\tau) \in \{b_1(\tau) \vee b_2(\tau), 1\}$.
– si pour tout $a \in Sync$, $(b_1(a) \wedge b_2(\bar{a})) \vee (b_2(a) \wedge b_1(\bar{a})) = 0$, alors $b(\tau) = b_1(\tau) \vee b_2(\tau)$.

Cette définition exprime qu'étant données deux séquences w_1 et w_2, si $a \in w_1$ et $\bar{a} \notin w_2$, alors a appraît dans la séquence $w_1 \amalg w_2$, et si $a \in w_1$ et $\bar{a} \in w_2$, a peut ne pas apparaître dans $w_1 \amalg w_2$ (dans ce cas, nous aurons τ qui apparaît dans $w_1 \amalg w_2$).

Dans ce cas, le nombre d'itérations à calculer pour arriver au point fixe est en $2^{O(|Act|)}$.

Exemple : Pour le programme de la figure 7.1, nous pouvons conclure qu'aucune configuration globale où π_1 est au point n_3 et π_2 au point m_3 n'est accessible puisque

$$\alpha\Big(Paths\big(||(n_0, m_0), ||(n_3, m_3)\big)\Big) \sqcap \alpha\big((Async \cup \{\tau\})^*\big) = \emptyset.$$

En effet, $\alpha\big((Async \cup \{\tau\})^*\big) = \{00001\}$, où les composantes des vecteurs de bits correspondent aux actions $a, b, \bar{a}, \bar{b}$, et τ, respectivement. De plus,

$$\begin{aligned} \alpha\Big(Paths\big(||(n_0, m_0), ||(n_3, m_3)\big)\Big) &= \alpha\big(Paths(n_0, n_3)\big) \otimes \alpha\big(Paths(m_0, m_3)\big) \\ &= \{10000, 01000\} \otimes \{00110\} \end{aligned}$$

où $\{10000, 01000\} \otimes \{00110\}$ est égal à

$$\{10110, 10111, 00111, 10011, 00011, 00101, 01110, 01111, 01101\}$$

8.4.2.3 Ordre des premières occurrences :

Nous prenons dans ce cas $\otimes = \amalg$. Le nombre d'itérations à calculer pour arriver au point fixe est au plus $2^{O(|Act|!)}$.

Exemple : Nous déduisons que le programme de la figure 7.4 ne peut pas atteindre la configuration (n_3,m_3) puisque $\alpha\Big(Paths\big(||(n_0, m_0), ||(n_3, m_3)\big)\Big)$ qui est égal à $\{ab, b\} \otimes \{\bar{b}\bar{a}, \bar{c}\bar{b}\bar{a}\}$ ne contient aucune séquence dans $(Async \cup \{\tau\})^*$.

8.5 Un exemple avec création dynamique de processus

Considérons le PFG de la figure 8.1 qui correspond à un programme où une procédure π_1 s'appelle elle-même en parallèle avec une procédure π_2. Ces procédures π_1 et π_2 communiquent par des signaux a et b. Nous ne représentons pas ici la procédure `main`, nous supposons que le programme commence à partir du point n_0.

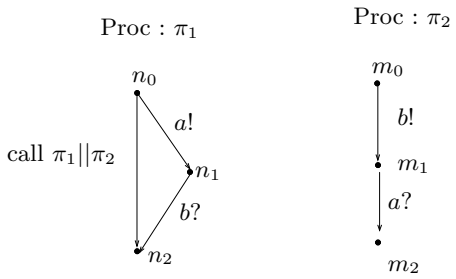


FIG. 8.1 – Un programme avec création dynamique de processus

Ce programme peut être modélisé par le SPA $\mathcal{R}$ contenant les règles suivantes[2] :

$$\begin{array}{lll} \mathcal{R}_1 : n_0 & \overset{a}{\hookrightarrow} & n_1 \\ \mathcal{R}_2 : n_1 & \overset{\bar{b}}{\hookrightarrow} & 0 \\ \mathcal{R}_3 : n_0 & \overset{\epsilon}{\hookrightarrow} & n_0||m_0 \end{array} \qquad \begin{array}{lll} \mathcal{R}_4 : m_0 & \overset{b}{\hookrightarrow} & m_1 \\ \mathcal{R}_5 : m_1 & \overset{\bar{a}}{\hookrightarrow} & 0 \end{array}$$

En utilisant l'abstraction des ordres des premières occurrences, nous pouvons montrer qu'en partant de n_0, il n'est pas possible d'atteindre une configuration où le point de contrôle m_1 est actif. Pour ce faire, nous posons $\mathcal{L} = \{n_0\}$ et $\mathcal{L}'$ l'ensemble des termes où les noeuds internes sont étiquetés par "||", et qui contiennent au moins une feuille étiquetée par m_1 (nous ne considérons pas "$\cdot$" dans $\mathcal{L}'$ puisque les règles du SPA ne font pas intervenir cet opérateur). Notre problème revient alors à vérifier que

$$\alpha\big(Paths(\mathcal{L},\mathcal{L}')\big) \sqcap \alpha\big((Async \cup \{\tau\})^*\big) = \emptyset.$$

C'est à dire, puisque dans ce cas $\alpha\big((Async \cup \{\tau\})^*\big) = \{\tau, \epsilon\}$, de vérifier si

$$\alpha\big(Paths(\mathcal{L},\mathcal{L}')\big) \cap \{\tau, \epsilon\} = \emptyset$$

Nous allons appliquer notre procédure pour calculer $\alpha\big(Paths(\mathcal{L},\mathcal{L}')\big)$. Soit alors $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini qui reconnaît $\mathcal{L}$, où $Q = F = \{p\}$ et $\delta = n_0 \to p$; et soit $\mathcal{A}' = (Q', \Sigma, F', \delta')$ un automate qui reconnaît $\mathcal{L}'$ tel que $Q = \{s, s_1\}$, $F = \{s_1\}$ et δ' est l'ensemble des règles :

- $n \to s$ pour tout $n \in E = \{0, n_0, n_1, n_2, m_0, m_1\}$;
- $m_1 \to s_1$;
- $||(s, s) \to s$, $||(s_1, s) \to s_1$, $||(s, s_1) \to s_1$.

s_1 reconnaît les termes ayant une feuille m_1, et s reconnaît tous les termes.

Notre but est de calculer $\alpha\big(\lambda(p, s_1)\big)$. Pour ce faire, nous allons poser le système de contraintes dont la solution caractérise cet ensemble. Soient alors $Q^{\mathcal{R}} = \{q_{n_0||m_0}, q_n, n \in E\}$, et $\delta^{\mathcal{R}}$ l'ensemble des règles de transitions suivantes :

[2]Ici, pour simplifier l'analyse en réduisant le nombre des variables, si $n \xrightarrow{inst} r$ est une arête du PFG où r est un point de retour, nous considérons directement la règle qui modélise la terminaison $n \overset{inst}{\hookrightarrow} 0$ au lieu de considérer les règles $n \overset{inst}{\hookrightarrow} r$ et $r \overset{\epsilon}{\hookrightarrow} 0$.

– $n \to q_n$ pour $n \in E$;
– $||(q_{n_0}, q_{m_0}) \to q_{n_0||m_0}$.

Soient $\mathcal{Q} = Q \cup Q^{\mathcal{R}}$, $\Delta = \delta \cup \delta^{\mathcal{R}}$, $\mathcal{Q}' = Q' \cup Q^{\mathcal{R}}$, et $\Delta' = \delta' \cup \delta^{\mathcal{R}}$. Nous appliquons les règles (β_1)–(β_5) et nous obtenons le système de contraintes suivant :

1. $l(q_{n_0}, s_1) \amalg l(q_{m_0}, s) \subseteq l(q_{n_0||m_0}, s_1)$;
2. $l(q_{n_0}, s) \amalg l(q_{m_0}, s_1) \subseteq l(q_{n_0||m_0}, s_1)$;
3. $l(q_{n_0}, s) \amalg l(q_{m_0}, s) \subseteq l(q_{n_0||m_0}, s)$;
4. $l(q, q_{n_0}) \cdot a \cdot l(q_{n_1}, q') \subseteq l(q, q')$, pour tout $q \in \mathcal{Q}$ et $q' \in \mathcal{Q}'$;
5. $l(q, q_{n_1}) \cdot \bar{b} \cdot l(q_0, q') \subseteq l(q, q')$, pour tout $q \in \mathcal{Q}$ et $q' \in \mathcal{Q}'$;
6. $l(q, q_{n_0}) \cdot \epsilon \cdot l(q_{n_0||m_0}, q') \subseteq l(q, q')$, pour tout $q \in \mathcal{Q}$ et $q' \in \mathcal{Q}'$;
7. $l(q, q_{m_0}) \cdot b \cdot l(q_{m_1}, q') \subseteq l(q, q')$, pour tout $q \in \mathcal{Q}$ et $q' \in \mathcal{Q}'$;
8. $l(q, q_{m_1}) \cdot \bar{a} \cdot l(q_0, q') \subseteq l(q, q')$, pour tout $q \in \mathcal{Q}$ et $q' \in \mathcal{Q}'$;
9. $\epsilon \in l(p, q_{n_0})$; $\epsilon \in l(q_n, s)$, $n \in E$; $\epsilon \in l(q_{n_0||m_0}, s)$; $\epsilon \in l(q_{m_1}, s_1)$; et $\epsilon \in l(q, q)$ pour tout $q \in Q^{\mathcal{R}}$.

Les contraintes (1), (2), et (3) sont obtenues par (β_4) puisque $||(q_{n_0}, q_{m_0}) \to q_{n_0||m_0} \in \Delta$ et $||(s, s) \to s$, $||(s_1, s) \to s_1$, et $||(s, s_1) \to s_1$ sont des règles de Δ'. Les contraintes (4), (5), (6), (7), et (8) sont obtenues en appliquant (β_5) aux règles $\mathcal{R}_1$, $\mathcal{R}_2$, $\mathcal{R}_3$, $\mathcal{R}_4$, et $\mathcal{R}_5$, respectivement. Les contraintes (9) sont obtenues à partir de (β_1).

En résolvant ce système de contraintes dans le domaine des "ordres des premières occurrences", nous obtenons que $\alpha\big(\lambda(p, s_1)\big)$ est égal à l'ensemble des mots suivants : $b, b\bar{a}, ab, ba, ab\bar{a}, ba\bar{a}, b\bar{a}a, b\tau, a\bar{b}b, ab\bar{b}, ba\bar{b}, a\tau, a\bar{b}b\bar{a}, ab\bar{b}\bar{a}, ab\bar{a}\bar{b}, ba\bar{b}\bar{a}, ba\bar{a}\bar{b}, b\bar{a}a\bar{b}, a\tau\bar{a}, b\tau\bar{b}, a\tau b, ab\tau, b\tau\bar{a}, b\bar{a}\tau, a\tau b\bar{a}, ab\tau\bar{a}, a\tau\bar{a}b, ab\bar{a}\tau, a\tau\bar{a}b, b\tau\bar{a}\bar{b}, b\tau\bar{b}\bar{a}, b\bar{a}\tau\bar{b}, b\bar{a}\bar{b}\tau$.

Comme l'intersection de cet ensemble avec $\{\tau, \epsilon\}$ est vide, nous déduisons que $\mathcal{L}'$ n'est pas accessible à partir de $\mathcal{L}$. Ce qui veut dire qu'il n'est pas possible d'atteindre le point m_1 à partir de n_0.

8.6 Conclusion

8.6.1 Bilan

Nous avons présenté une approche générique pour l'analyse statique des programmes concurrents avec récursion et création dynamique de processus. Notre méthode permet le calcul effectif d'abstractions des langages de chemins d'exécutions des programmes. Elle est basée sur (1) la caractérisation des langages de chemins comme la plus petite solution d'un système de contraintes, et (2) la résolution de ce système dans un domaine abstrait. Plus précisément, nous nous plaçons dans le cadre des abstractions à chaînes finies où un calcul itératif du plus petit point fixe termine. Nous pouvons instancier ce cadre par des classes d'abstractions différentes selon le coût et la précision désirés de l'analyse. Nous étendons les abstractions à chaînes finis que nous avons proposées dans le chapitre précédent pour qu'elles puissent être appliquées dans ce cadre où le dynamisme est permis.

La technique présentée dans ce chapitre et celle du chapitre précédent sont basées sur le même principe, à savoir, le calcul d'abstractions des langages de chemins d'exécutions des programmes en résolvant des systèmes de contraintes dans des domaines abstraits. Ces deux techniques se complètent. En effet, la méthode présentée au chapitre précédent n'autorise pas la création dynamique de processus, mais elle traite la récursivité de manière exacte ; alors que la technique de ce chapitre considère le dynamisme, permet de tenir compte de la récursivité, mais elle abstrait les valeurs des variables de retour que les procédures appelées doivent rendre aux procédures appelantes. Par conséquent, selon le type du programme que nous devons analyser, et selon l'information que nous voulons garder, l'une ou l'autre de ces deux techniques peuvent être utilisées.

L'avantage majeur de ces deux techniques est leur généricité. En effet, elles offrent un cadre général et uniforme pour calculer des abstractions des langages de chemins d'exécutions des programmes. Elles peuvent être instanciées par n'importe quelle classe d'abstraction qui soit à chaînes finies ou commutative (dans le cas de la technique du chapitre précédent) pour permettre des analyses graduelles des programmes, ayant différents coûts et différentes précisions. Nous pouvons également instancier nos techniques par n'importe quelle classe d'abstractions, ou essayer de résoudre les systèmes de contraintes directement dans le domaine concret en appliquant un calcul itératif et en utilisant les méthodes d'élargissement telles que définies dans le cadre de l'interprétation abstraite, pour accélérer le calcul de sur-approximations du plus petit point fixe [CC77].

Nous pouvons aussi considérer des séries d'abstractions finies qui peuvent être raffinées, telles que par exemple la série infinie des abstractions $(A_i)_{i>0}$ qui considérent l'ordre d'apparition dans le langage des $i^{ème}$ premières occurrences de chaque lettre (notre abstraction de "l'ordre des premières occurrences" correspond à la classe A_1), et considérer l'une de ces différentes abstractions selon la précision et le coût désirés de l'analyse. Si nous pensons que le résultat de l'analyse est trop grossier, nous pouvons toujours le raffiner en considérant une abstraction plus fine de cette série infinie.

8.6.2 Comparaison avec d'autres travaux

Dans [Mo02, SS01], les auteurs proposent d'autres techniques de calcul d'abstractions des chemins d'exécutions basées sur la résolution d'un système de contraintes dans un domaine abstrait. L'approche qu'ils proposent ne permet cependant pas de tenir compte de la synchronisation. De plus, leur technique de construction des systèmes de contraintes est différente des nôtres. En effet, ils posent leurs contraintes directement sur le PFG, alors que nos contraintes sont posées à partir d'automates correspondant aux ensembles de configurations du système.

Comme nous l'avons déjà dit, à notre connaissance, le seul travail qui considère récursivité, création dynamique de processus, et synchronisation, est celui de [DS91]. Nous avons montré qu'en considérant une abstraction finie particulière, la technique présentée au chapitre précédent permet de mener l'analyse proposée dans cet article. Il serait intéressant de voir si cette abstraction pourrait être étendue pour être utilisée dans le cadre présenté dans ce chapitre.

Quatrième partie

Analyse d'accessibilité par élargissement

Chapitre 9

Calcul des accessibles par élargissement

Dans les chapitres précédents, nous avons présenté des algorithmes de calcul de clôtures réflexives transitives spécifiques à des sous-classes de relations et de langages réguliers. Ces algorithmes ne peuvent s'appliquer que si le langage et/ou la relation considérés appartiennent à ces sous-classes qui sont certes assez significatives pour modéliser beaucoup de classes de systèmes, mais qui n'apparaissent pas forcément dans tous les contextes. Nous proposons dans ce chapitre une technique générale d'accélération appelée *élargissement régulier* qui peut être appliquée pour calculer les ensembles des accessibles indépendamment du type de langages ou de relations considérés. Plus précisément, étant donnés une relation et un langage réguliers de mots ou d'arbres, cette technique permet en général, lorsqu'elle termine, de calculer des sur-approximations de l'ensemble des accessibles, et peut dans certains cas calculer cet ensemble de manière *précise* grâce à un test d'invariance.

Etant donnés un langage régulier $\mathcal{L}$ représenté par un automate fini de mots ou d'arbres $\mathcal{A}$, et une relation régulière $\mathcal{R}$ donnée par un transducteur fini de mots ou un transducteur d'arbres linéaire $\mathcal{T}$, la technique que nous présentons est basée sur la détection de croissances pendant le calcul itératif de la séquence $\mathcal{A}, \mathcal{T}(\mathcal{A}), \mathcal{T}^2(\mathcal{A}), \ldots$ dans le but de deviner *automatiquement* un automate $\mathcal{A}'$ qui représente l'ensemble des accessibles $\mathcal{R}^*(\mathcal{L})$. Un test permet ensuite de savoir si l'automate deviné correspond à une sur-approximation de l'ensemble des accessibles. Ce même test permet dans certains cas de décider si l'automate construit reconnaît *exactement* l'ensemble des accessibles. L'idée intuitive est la suivante. Prenons par exemple le langage de mots a^*b représenté par l'automate $\mathcal{A}$ de la figure 9, et $\mathcal{R} = copy(a^*)(a, b)copy(b^*)$ la relation régulière de mots qui réécrit le a le plus à droite en b, et qui est donnée par le transducteur $\mathcal{T}$ de la figure 9. En comparant les structures des automates $\mathcal{A}$ et $\mathcal{T}(\mathcal{A})$, nous remarquons qu'il y a un "b" qui s'est rajouté à droite. L'idée est alors de deviner qu'à chaque application de $\mathcal{T}$, un "b" va se rajouter, et donc extrapoler en rajoutant la boucle b^* ce qui donne l'automate $\mathcal{A}'$ représenté à la figure 9 qui correspond précisément à $\mathcal{R}^*(a^*b) = a^*bb^*$.

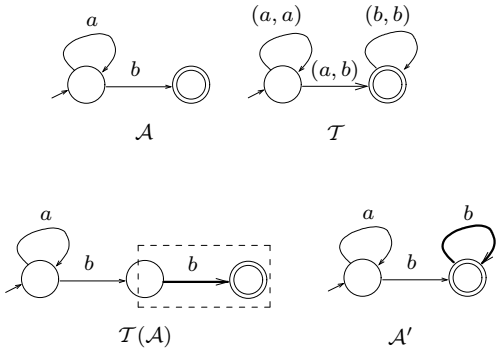


FIG. 9.1 – Exemple d'élargissement sur les automates de mots

Pour que tout ceci ait un sens, il nous faut définir selon quel critère les structures des automates sont-elles comparées. Pour ce faire, nous proposons de représenter un automate de mots ou d'arbres par un hypergraphe dirigé. En particulier, un automate de mots est représenté par un graphe de manière standard comme c'est le cas dans l'exemple présenté ci-dessus, puisque (comme nous allons le voir) un graphe est un hypergraphe dirigé particulier. Nous comparons ensuite les automates en comparant les structures des hypergraphes qui leur correspondent par rapport à la bisimulation observationnelle de Milner [Mil80]. Dans la première section de ce chapitre, nous montrons comment représenter un automate de mots ou d'arbres par un hypergraphe dirigé, et nous définissons la notion de bisimulation entre les hypergraphes. Ensuite, nous introduisons le principe général de notre méthode d'*élargissement* qui consiste à comparer les hypergraphes correspondants à $\mathcal{A}$ et $\mathcal{T}(\mathcal{A})$; si nous détectons dans $\mathcal{T}(\mathcal{A})$ un hypergraphe Δ qui s'est rajouté, c-à-d., un hypergraphe Δ tel que si nous l'enlevons nous obtenons un hypergraphe bisimilaire à celui de $\mathcal{A}$; alors nous itérons Δ en fusionnant les sommets d'entrée aux sommets de sortie qui leur correspondent. Dans l'exemple précédent, la croissance Δ est le sous-graphe représenté dans le rectangle en pointillés.

Dans la troisième partie de ce chapitre, nous montrons que notre technique peut être *exacte* si la relation considérée est *bien-fondée* et nous donnons certains exemples de relations de mots et d'arbres qui satisfont cette condition, et pour lesquelles donc notre technique permet, en cas de terminaison, de calculer l'ensemble exact des accessibles.

Nous montrons ensuite que notre technique est assez générale et qu'elle permet de calculer les clôtures transitives pour plusieurs classes de relations et de langages. Plus précisément, nous montrons que cette méthode permet de calculer la clôture transitive de toute *context-relation*, une classe de relations de mots introduite dans [ABJN99] pour représenter les systèmes paramétrés linéaires. Nous montrons également qu'elle est capable de calculer la fermeture d'un langage APC par semi-commutations, ce qui donne un autre algorithme pour le théorème 3.1.2. Elle peut aussi calculer la clôture transitive d'un langage régulier par une relation semi-monadique linéaire. Cette classe de relations introduite dans [CDGV94] définit une classe significative de systèmes de réécriture de termes. Notre technique est aussi capable de calculer $Post^*_{\mathcal{R},=}(\mathcal{L})$

pour tout système PRS bien-fondé $\mathcal{R}$ et tout langage régulier de termes $\mathcal{L}$. Nous en déduisons qu'elle peut calculer cet ensemble pour tout système PA $\mathcal{R}$. Finalement, nous montrons qu'elle permet de calculer la clôture réflexive-transitive de tout système WOS en simulant l'algorithme sous-jacent au théorème 3.2.1.

Nous montrons ensuite dans la section 9.6 comment appliquer notre technique d'élargissement à la vérification des systèmes paramétrés et des programmes récursifs parallèles. Avant de conclure, nous comparons notre méthode avec les techniques existantes.

Le contenu de ce chapitre a été publié dans [BJNT00, Tou01, BT02].

9.1 Hypergraphes dirigés et bisimulation

9.1.1 Représentation des automates par des hypergraphes

Définition 9.1.1 (Hypergraphes dirigés) *Soient V un ensemble de sommets et Σ un alphabet muni d'une fonction d'arité comprenant un symbole particulier ϵ d'arité 1. Soient $f \in \Sigma_n$ et $v, v_1, \ldots, v_n \in V$, le $n+2$-uplet $(v, f, v_1, \ldots, v_n)$ est un* **hyperarc dirigé** *étiqueté par f et connectant dans l'ordre v aux sommets $v_1, \ldots, v_n$. Si $f \in \Sigma_1$, (v, f, v_1) est appelé un* **arc dirigé**. *Nous écrivons $v \xrightarrow{f} v_1, \ldots, v_n$ pour chaque hyperarc dirigé $(v, f, v_1, \ldots, v_n)$, ou juste $v \xrightarrow{a}$ si $a \in \Sigma_0$.*

Un **hypergraphe dirigé** *est une paire $\mathcal{G} = (V, H)$ où V est un ensemble de sommets et H est un ensemble d'hyperarcs dirigés sur V. Un* **graphe dirigé** *est un hypergraphe dirigé dont tous les hyperarcs dirigés sont des arcs dirigés.*

Comme il n'y a pas de risque de confusion, nous utilisons dorénavant les appellations *(hyper)graphe* et *(hyper)arc* pour désigner respectivement un *(hyper)graphe dirigé* ou un *(hyper)arc dirigé*.

Il est bien connu qu'un automate fini de mots $\mathcal{A} = (Q, I, \Sigma, \delta, F)$ peut être représenté par un triplet $(\mathcal{G}, I, F)$, où $\mathcal{G} = (Q, \delta)$ est le graphe représentant la relation de transition δ, I est l'ensemble des états initiaux et F l'ensemble des états finaux. De manière analogue, nous proposons de représenter la relation de transition d'un automate d'arbres fini par un hypergraphe de la manière suivante : Soit $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini, la relation de transition δ peut être représentée par l'hypergraphe $\mathcal{G}_\delta = (Q, H_\delta)$, où H_δ est défini par :

- $q \xrightarrow{f} q_1, \ldots, q_n \in H_\delta$ pour chaque règle $f(q_1, \ldots, q_n) \to q$ dans δ.
- $q \xrightarrow{a} \in H_\delta$ pour chaque règle $a \to q$ dans δ.
- $q \xrightarrow{\epsilon} q' \in H_\delta$ pour chaque règle $q \to q'$ dans δ.

De cette façon, un automate d'arbres fini $\mathcal{A} = (Q, \Sigma, F, \delta)$ peut être représenté par une paire $(\mathcal{G}, F)$, où $\mathcal{G}$ est l'hypergraphe qui représente sa relation de transition δ et F est l'ensemble d'états finaux. Parfois, pour traiter les automates d'arbres et les automate de mots de manière uniforme, nous représentons l'automate d'arbres $\mathcal{A}$ par le triplet $(\mathcal{G}, \emptyset, F)$.

9.1.2 Bisimulation d'hypergraphes

Nous introduisons ci-dessous la notion de bisimulation d'hypergraphes. Comme les automates que nous manipulons ne sont pas forcément déterministes, les hypergraphes qui les représentent peuvent contenir des ϵ-transitions. Nous considérons donc la bisimulation observationnelle de Milner [Mil80] pour comparer deux hypergraphes. Pour définir cette relation d'équivalences, nous introduisons la notation $v \stackrel{f}{\Rightarrow}_H v_1, \ldots, v_n$ qui exprime quil y a des sommets $u, u_1, \ldots, u_n$ tels que dans l'ensemble d'hyperarcs H, il y a un chemin étiqueté par des ϵ-transitions qui mène de v à u, il y a un hyperarc de la forme $u \stackrel{f}{\rightarrow} u_1, \ldots, u_n$, et des chemins étiquetés par des ϵ-transitions qui mènent de u_i à v_i, pour chaque i, $1 \leq i \leq n$:

Notation 9.1.1 *Soit $\mathcal{G} = (V, H)$ un hypergraphe. Nous écrivons $v \stackrel{f}{\Rightarrow}_H v_1, \ldots, v_n$ s'il existe des sommets $v^0, \ldots, v^k, u_1^0, \ldots, u_1^{k_1}, u_n^0, \ldots, u_n^{k_n}$ tels que :*
- $v = v^0$,
- $\forall i, 0 \leq i < k, v^i \stackrel{\epsilon}{\longrightarrow} v^{i+1} \in H$;
- $v^k \stackrel{f}{\rightarrow} u_1^0, \ldots, u_n^0 \in H$;
- $\forall j, 1 \leq j \leq n; \exists k_j \mid u_j^{k_j} = v_j$ *et* $\forall i, 0 \leq i < k_j; u_j^i \stackrel{\epsilon}{\longrightarrow} u_j^{i+1} \in H$.

Nous pouvons maintenant définir la notion de bisimulation *"observationnelle"* sur les hypergraphes. Comme nous n'allons utiliser que cette notion de bisimulation, nous l'appelons simplement *bisimulation d'hypergraphes* :

Définition 9.1.2 *Soit $\mathcal{G} = (V, H)$ un hypergraphe et $F \subseteq V$ un ensemble de sommets acceptants. Une* **bisimulation d'hypergraphes** *par rapport à F est une relation binaire symétrique $\rho \subseteq V \times V$ telle que pour chaque $v, v' \in V$, $(v, v') \in \rho$ ssi*
- *$v \in F$ ssi $v' \in F$,*
- *Si $v \stackrel{f}{\Rightarrow}_H v_1, \ldots, v_n$, alors il existe des sommets $v'_1, \ldots, v'_n \in V$ tels que $v' \stackrel{f}{\Rightarrow}_H v'_1, \ldots, v'_n$, et pour chaque $i \in \{1, \ldots, n\}$, $(v_i, v'_i) \in \rho$. Nous écrivons $v \sim v'$ s'il existe une bisimulation d'hypergraphes liant v et v'.*

Etant donnés deux automates de mots $\mathcal{A} = (\mathcal{G}, I, F)$ et $\mathcal{A}' = (\mathcal{G}', I', F')$, nous écrivons $\mathcal{A} \sim \mathcal{A}'$ ssi chaque sommet dans I est bisimilaire à un sommet dans I' par rapport à $F \cup F'$ et vice versa. De même, étant donnés deux automates d'arbres $\mathcal{A} = (\mathcal{G}, F)$ et $\mathcal{A}' = (\mathcal{G}', F')$, nous écrivons $\mathcal{A} \sim \mathcal{A}'$ ssi chaque sommet dans F est bisimilaire à un sommet dans F' par rapport à $F \cup F'$ et vice versa. Par abus de langage, nous parlons parfois d'hypergraphes bisimilaires si I, I', F, et F' (resp. F et F') sont compris dans le contexte.

9.2 Techniques d'extrapolation sur les hypergraphes

9.2.1 Principe de l'élargissement régulier

Nous décrivons dans cette partie notre principe d'*élargissement régulier* : nous comparons un hypergraphe $\mathcal{G}'$ (qui est censé représenter un automate $\mathcal{A}$ correspondant

à un langage $\mathcal{L}$) à un hypergraphe $\mathcal{G}$ (qui est censé représenter un automate $\mathcal{T}(\mathcal{A})$ correspondant à $\mathcal{R}(\mathcal{L})$) dans le but de deviner un sous-hypergraphe Δ de $\mathcal{G}$, un sous-ensemble $\mathcal{I}_\Delta$ de sommets de Δ, et une partition φ des sommets de $\mathcal{I}_\Delta$ tels que si nous enlevons Δ de $\mathcal{G}$, et que nous fusionnons les sommets de $\mathcal{I}_\Delta$ selon la partition φ, nous obtenons un hypergraphe $\mathcal{G}\backslash_\varphi\Delta$ bisimilaire à $\mathcal{G}'$. L'*élargissement* consiste alors à calculer l'hypergraphe $\mathcal{G}[\Delta \leftarrow \Delta^+]$ obtenu à partir de $\mathcal{G}$ en fusionnant les états équivalents par rapport à φ (sans enlever les hyperarcs de Δ). Ceci revient à rajouter des boucles qui permettent d'itérer la croissance Δ. Intuitivement, deviner la partition φ revient à deviner les états de Δ qu'il faut fusionner pour obtenir les boucles qu'il faut rajouter à $\mathcal{G}$ pour avoir la limite qui doit correspondre à $\mathcal{R}^*(\mathcal{L})$. Ci-dessous, nous définissons formellement les hypergraphes $\mathcal{G}\backslash_\varphi\Delta$ et $\mathcal{G}[\Delta \leftarrow \Delta^+]$.

Définition 9.2.1 *Soit $\mathcal{G} = (V, H)$ un hypergraphe. Supposons donnés :*
- *un sous-hypergraphe de $\mathcal{G}$: $\Delta = (V_\Delta, H_\Delta)$ ($V_\Delta \subseteq V$ et $H_\Delta \subseteq H$) ;*
- *un sous ensemble $\mathcal{I}_\Delta$ de V_Δ ;*
- *φ : une partition de $\mathcal{I}_\Delta$.*

Soit $\sim_\varphi$ la relation d'équivalence induite par φ. Nous définissons deux hypergraphes $\mathcal{G}\backslash_\varphi\Delta$ et $\mathcal{G}[\Delta \leftarrow \Delta^+]$ comme suit :

- *$\mathcal{G}\backslash_\varphi\Delta$ est l'hypergraphe (V', H') tel que*
  - *$V' = V \setminus V_\Delta \cup \{s_{[v]_\varphi} \mid v \in \mathcal{I}_\Delta\}$ et*
  - *$H' = \{v'_0 \xrightarrow{f} v'_1, \ldots, v'_n \mid \exists v_0 \xrightarrow{f} v_1, \ldots, v_n \in H$ t.q. $v_i \in V \setminus V_\Delta$ et $v'_i = v_i$; ou $v_i \in \mathcal{I}_\Delta$ et $v'_i = s_{[v_i]_\varphi}\}$,*

  où $[v]_\varphi$ dénote la classe d'équivalence par rapport à $\sim_\varphi$ du sommet v. Intuitivement, $\mathcal{G}\backslash_\varphi\Delta$ est l'hypergraphe obtenu de $\mathcal{G}$ en enlevant tous les hyperarcs de Δ, et en collant les sommets $\sim_\varphi$-équivalents.
- *$\mathcal{G}[\Delta \leftarrow \Delta^+]$ est l'hypergraphe (V'', H'') où :*
  - *$V'' = (V \setminus \mathcal{I}_\Delta) \cup \{s_{[v]_\varphi} \mid v \in \mathcal{I}_\Delta\}$, et*
  - *$H'' = \{v'_0 \xrightarrow{f} v'_1, \ldots, v'_n \mid v_0 \xrightarrow{f} v_1, \ldots, v_n \in H$ t.q. $v'_i = v_i$ si $v_i \notin \mathcal{I}_\Delta$, et $v'_i = s_{[v_i]_\varphi}$ sinon $\}$.*

  Intuitivement, $\mathcal{G}[\Delta \leftarrow \Delta^+]$ est obtenu en fusionnant les états $\sim_\varphi$-équivalents de Δ, ce qui rajoute des boucles permettant d'itérer Δ.

Maintenant, nous pouvons définir l'opération d'*élargissement régulier* sur les automates de mots ou d'arbres :

Définition 9.2.2 (Elargissement régulier) *Soient deux automates $\mathcal{A} = (\mathcal{G}, I, F)$ et $\mathcal{A}' = (\mathcal{G}', I', F')$. Etant donnés un sous hypergraphe Δ de $\mathcal{G}'$, un ensemble de sommets $\mathcal{I}_\Delta$ de Δ, et une partition φ de $\mathcal{I}_\Delta$, si*

$$(\mathcal{G}, I, F) \sim \big(\mathcal{G}' \backslash_\varphi \Delta, (I' \setminus V_\Delta) \cup \{s_{[v]_\varphi} \mid v \in I' \cap \mathcal{I}_\Delta\}, (F' \setminus V_\Delta) \cup \{s_{[v]_\varphi} \mid v \in F' \cap \mathcal{I}_\Delta\}\big) \tag{9.1}$$

*alors nous définissons l'opérateur d'***élargissement**

$$\nabla(\mathcal{A}, \mathcal{A}', \Delta, \varphi) = \big(\mathcal{G}'[\Delta \leftarrow \Delta^+], (I' \backslash \mathcal{I}_\Delta) \cup \{s_{[v]_\varphi} \mid v \in I' \cap \mathcal{I}_\Delta\}, (F' \backslash \mathcal{I}_\Delta) \cup \{s_{[v]_\varphi} \mid v \in F' \cap \mathcal{I}_\Delta\}\big)$$

Observons que dans le cas des automates d'arbres où I et I' sont vides, l'élargissement revient à tester si

$$(\mathcal{G}, F) \sim \big(\mathcal{G}' \setminus_\varphi \Delta, (F' \setminus V_\Delta) \cup \{s_{[v]_\varphi} \mid v \in F' \cap \mathcal{I}_\Delta\}\big) \tag{9.2}$$

Si cette condition est satisfaite, nous définissons l'opérateur d'**élargissement d'arbres**

$$\nabla(\mathcal{A}, \mathcal{A}', \Delta, \varphi) = \big(\mathcal{G}'[\Delta \leftarrow \Delta^+], (F' \setminus \mathcal{I}_\Delta) \cup \{s_{[v]_\varphi} \mid v \in F' \cap \mathcal{I}_\Delta\}\big)$$

Remarque 9.2.1 *Observons que si nous voulions comparer les automates $\mathcal{A}$ et $\mathcal{A}'$ non pas par leur structure mais par les langages qu'ils reconnaissent, nous pouvons utiliser le même principe donné ci-dessus en considérant des structures déterministes.*

9.2.2 Calcul de $\mathcal{R}^*(\mathcal{L})$ par élargissement

Soit $\mathcal{R}$ une relation régulière de mots ou d'arbres représentée par un transducteur de mots ou un transducteur linéaire d'arbres $\mathcal{T}$; et soit $\mathcal{L}$ un langage régulier de mots ou d'arbres représenté par un automate fini de mots ou d'arbres $\mathcal{A}$. Nous proposons d'appliquer l'élargissement pour deviner $\mathcal{R}^*(\mathcal{L})$ de la manière suivante : calculer la séquence $\mathcal{A}, \mathcal{T}(\mathcal{A}), \mathcal{T}^2(\mathcal{A}), \ldots$ [1] et comparer à chaque étape l'hypergraphe du nouvel automate calculé $\mathcal{T}^i(\mathcal{A})$ aux hypergraphes des automates précédents $\mathcal{T}^j(\mathcal{A})$, $j < i$ dans l'ordre décroissant des indices j. Dès qu'une croissance est détectée entre $\mathcal{T}^i(\mathcal{A})$ et $\mathcal{T}^k(\mathcal{A})$ pour un certain indice k, nous extrapolons et calculons un automate $\mathcal{A}'$ par élargissement comme décrit ci-dessus. Par cette opération, nous devinons que $\mathcal{A}'$ représente l'ensemble des accessibles, ou une sur-approximation de cet ensemble. Pour nous assurer que nous avons bien deviné, nous vérifions que le langage $\mathcal{L}'$ reconnu par $\mathcal{A}'$ est invariant par $\mathcal{R}$ en testant si :

$$\mathcal{L}' = \mathcal{R}(\mathcal{L}') \cup \mathcal{L}$$

Si c'est le cas, nous arrêtons le calcul et déduisons que $\mathcal{L}'$ est une sur-approximation de l'ensemble des accessibles, sinons nous n'extrapolons pas et nous continuons à comparer $\mathcal{T}^i(\mathcal{A})$ aux automates $\mathcal{T}^j(\mathcal{A})$, pour $j < k$ jusqu'à atteindre l'indice $j = 1$, auquel cas, nous passons à l'itération suivante, nous calculons $\mathcal{T}^{i+1}(\mathcal{A})$ et le comparons à ses prédécesseurs, etc. Nous pouvons simplifier cette procédure en ne comparant le nouvel automate calculé qu'à son prédécesseur immédiat, c-à-d., en ne comparant à chaque fois $\mathcal{T}^i(\mathcal{A})$ qu'à $\mathcal{T}^{i-1}(\mathcal{A})$, si l'élargissement ne peut pas s'appliquer ou si l'ensemble calculé ne satisfait pas le test, nous passons à l'itération suivante et calculons $\mathcal{T}^{i+1}(\mathcal{A})$, etc. Ou encore, nous pouvons fixer une profondeur de comparaison k et ne comparer $\mathcal{T}^i(\mathcal{A})$ qu'à $\mathcal{T}^j(\mathcal{A})$, pour $i - k \leq j \leq i - 1$. Si la limite n'a pas été devinée à cette profondeur, nous passons à l'itération suivante et calculons $\mathcal{T}^{i+1}(\mathcal{A})$, etc.

[1] Cette séquence peut être calculée en composant à chaque fois par le transducteur $\mathcal{T}$ comme décrit dans la définition 2.1.17 si $\mathcal{T}$ est un transducteur de mots, ou dans la définition 2.1.24 si $\mathcal{T}$ est un transducteur linéaire d'arbres. En effet, comme l'opération de produit des automates est associative, $\mathcal{T}^n(\mathcal{A})$ est égal à $\underbrace{\mathcal{T}\big(\mathcal{T}(\cdots\mathcal{T}(\mathcal{A})\cdots)\big)}_{n \text{ fois}}$.

Dans la section suivante nous montrons que pour certaines relations, tester $\mathcal{L}' = \mathcal{R}(\mathcal{L}') \cup \mathcal{L}$ revient à s'assurer que l'ensemble calculé est *exactement* égal à l'ensemble des accessibles.

Exemple 9.2.1 *Soit $\mathcal{L} = (a + c)^*(b + d)^*$ un langage représenté par l'automate $\mathcal{A}$ de la figure 9.2, et $\mathcal{R}$ la relation de semi-commutation $\mathcal{R} = copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$ représentée par le transducteur $\mathcal{T}$ de la figure 9.3. Sur les figures, les états initiaux sont marqués par des flêches entrantes et les états finaux par des doubles cercles. Alors $\mathcal{T}(\mathcal{A})$ qui reconnaît le langage $\mathcal{R}(\mathcal{L}) = (a + c)^*ba(b + d)^*$ et $\mathcal{T}^2(\mathcal{A})$ qui reconnaît le langage $\mathcal{R}^2(\mathcal{L}) = (a + c)^*b(a + b)a(b + d)^*$ sont représentés sur les figures 9.4 et 9.5. Soient $\mathcal{G}$ et $\mathcal{G}'$ les graphes sous-jacents à ces deux automates.*

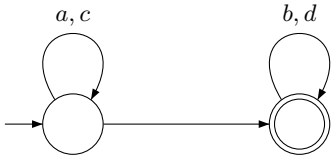


FIG. *9.2 – L'automate $\mathcal{A}$*

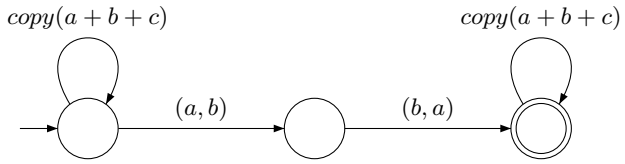


FIG. *9.3 – Le transducteur $\mathcal{T}$*

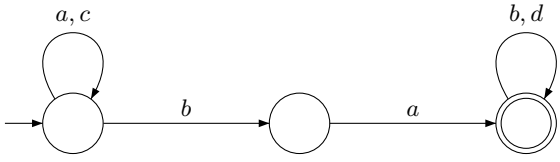


FIG. *9.4 – L'automate $\mathcal{T}(\mathcal{A})$*

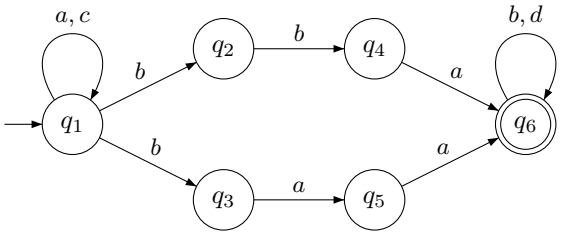


FIG. *9.5 – L'automate* $\mathcal{T}^2(\mathcal{A})$

En comparant $\mathcal{G}$ *et* $\mathcal{G}'$ *nous détectons le graphe* Δ *défini par les arcs* (q_2, b, q_4) *et* (q_3, a, q_5) *comme croissance. Si nous prenons* $\mathcal{I}_\Delta = \{q_2, q_3, q_4, q_5\}$ *et* φ *la partition* $\{q_2, q_3, q_4, q_5\}$*, nous obtenons un graphe identique à* $\mathcal{G}$*. Nous appliquons alors l'élargissement en fusionnant* q_2, q_3, q_4*, et* q_5 *et nous obtenons un automate qui reconnaît* $(a+c)^*b(a+b)^*a(b+d)^*$*. En rajoutant* $\mathcal{L}$ *nous obtenons* $\mathcal{R}^*(\mathcal{L}) = (a+c)^*(a+b)^*(b+d)^*$*.*

Exemple 9.2.2 (Elargissement d'arbres) *Considérons la règle de réécriture suivante :* $\mathcal{R} = a \to f(a,b)$ *correspondant à la relation qui réécrit une feuille étiquetée par* a *en* $f(a,b)$*. Cette relation est reconnue par le transducteur d'arbres linéaire* $\mathcal{T} = (Q, \Sigma, \Sigma, F, \delta)$ *tel que* $Q = \{q, q_a\}$*,* $F = \{q_a\}$*,* $\Sigma_1 = \{a, b\}$*,* $\Sigma_2 = \{f\}$*, et* δ *est l'ensemble des règles suivantes :*

1. $a \to q(a)$*,* $b \to q(b)$ *;*
2. $a \to q_a\big(f(a,b)\big)$ *;*
3. $f\big(q(x_1), q_a(x_2)\big) \to q_a\big(f(x_1,x_2)\big)$*,* $f\big(q_a(x_1), q(x_2)\big) \to q_a\big(f(x_1,x_2)\big)$ *;*
4. $f\big(q(x_1), q(x_2)\big) \to q\big(f(x_1,x_2)\big)$*.*

Intuitivement, les règles (1) et (4) expriment que l'état q *reconnaît les termes où aucune réécriture n'a eu lieu. Alors que l'état* q_a *reconnaît les termes où exactement une feuille étiquetée par* a *s'est réécrite en* $f(a,b)$*. En effet, la règle (2) annote le terme* $f(a,b)$ *qui a remplacé une feuille* a *par* q_a*. Les règles (3) garantissent qu'une seule feuille s'est réécrite.*

Supposons que nous voulions calculer $\mathcal{R}^*(a)$*. Soient* $\mathcal{A} = (\mathcal{G}_0, \{q_0\})$*,* $\mathcal{T}(\mathcal{A}) = (\mathcal{G}_1, \{q_2\})$*, et* $\mathcal{T}^2(\mathcal{A}) = (\mathcal{G}_2, \{q_3\})$ *les automates d'arbres reconnaissants* a*,* $\mathcal{R}(a)$*, et* $\mathcal{R}^2(a)$*. Leurs hypergraphes correspondants sont représentés dans la figure 9.6. En comparant* $\mathcal{G}_1$ *et* $\mathcal{G}_2$*, nous détectons une situation d'élargissement où* Δ *est l'hypergraphe* $(\{q_1, q_2, q_3\}, \{q_3 \xrightarrow{f} q_2, q_1\})$*,* $\mathcal{I}_\Delta = \{q_3, q_1, q_2\}$*, et* $\varphi = \{\{q_1\}, \{q_2, q_3\}\}$*. Par conséquent, l'opérateur d'élargissement* ∇ *produit l'automate* $(\mathcal{G}, \{q_3\})$ *obtenu en fusionnant les états* q_2 *et* q_3*, ce qui permet de rajouter la boucle représentée en gras. Cet automate définit précisément* $\mathcal{R}^+(a)$*. Son union avec* $(\mathcal{G}_0, \{q_0\})$ *correspond à* $\mathcal{R}^*(a)$*.*

9.2.3 Elargissement et relations transitives

Ce principe peut également être appliqué pour calculer les clôtures transitives des relations régulières de mots et des relations de réétiquetages d'arbres, puisque ces relations peuvent être vues comme des langages réguliers sur l'alphabet produit $\Sigma \times \Sigma$ (voir la section 2.1.2.3), et peuvent donc être représentées par des hypergraphes.

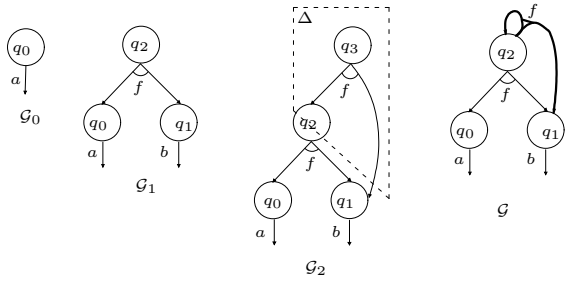


FIG. 9.6 – Illustration du mécanisme d'élargissement sur les automates d'arbres

Prenons par exemple la relation $\mathcal{R} = copy(\Sigma^*)(C, \bot)copy(\Sigma^*)$. Cette relation fait partie de la modélisation du " Bakery algorithm" que nous allons décrire plus tard à la section 9.6. Soit $\mathcal{T}$ le transduteur de la figure 9.7 qui reconnaît $\mathcal{R}$. $\mathcal{T}^2 = (\mathcal{G}, I, F)$ et $\mathcal{T}^3 = (\mathcal{G}', I', F')$ sont représentés sur les figures 9.8 et 9.9 [2]. En comparant $\mathcal{G}$ et $\mathcal{G}'$, nous détectons la croissance Δ définie par l'arc $\big(q_2, (C, \bot), q_3\big)$ telle que $\mathcal{I}_\Delta = \{q_2, q_3\}$ et φ est la partition $\{q_2, q_3\}$. Il est facile de voir que

$$(\mathcal{G}, I, F) \sim \big(\mathcal{G}' \setminus_\varphi \Delta, I', F'\big)$$

et donc nous pouvons appliquer l'élargissement en fusionnant les états q_2 et q_3. Nous obtenons un transducteur qui reconnaît $copy(\Sigma^*)(C, \bot)\big(copy(\Sigma)+(C, \bot)\big)^*(C, \bot)copy(\Sigma^*)$. En rajoutant cette relation à $\mathcal{R}$, nous obtenons $\mathcal{R}^+ = copy(\Sigma^*)(C, \bot)\big(copy(\Sigma) + (C, \bot)\big)^*$.

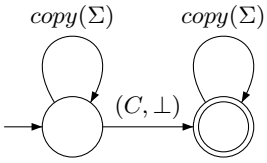


FIG. 9.7 – Transducteur $\mathcal{T}$

[2] Les "vrais" transducteurs $\mathcal{T}^2$ et $\mathcal{T}^3$ obtenus en calculant un produit avec $\mathcal{T}$ comprennent d'autres branches qui sont identiques, à un renommage d'états près, aux branches représentées sur les figures. Nous ne représentons pas ces branches pour des raisons de lisibilité et de simplicité puisque ceci ne change rien à l'analyse.

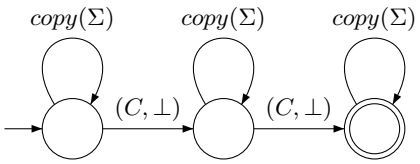


FIG. 9.8 – Transducteur $\mathcal{T}^2$

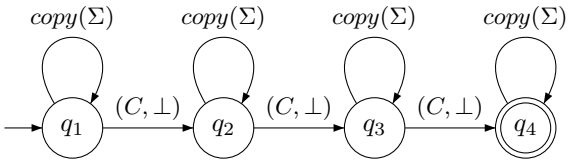


FIG. 9.9 – Transducteur $\mathcal{T}^3$

9.3 Elargissement exact

Nous définissons dans cette section une classe de relations pour laquelle le test $\mathcal{L}' = \mathcal{R}(\mathcal{L}') \cup \mathcal{L}$ permet de décider si $\mathcal{L}' = \mathcal{R}^*(\mathcal{L})$.

9.3.1 Relations bien-fondées

Définition 9.3.1 *Une relation $\mathcal{R}$ est nœthérienne (resp. bien-fondée) s'il n'existe pas de séquence infinie $t_0, t_1, \ldots$ telle que pour chaque $i \geq 0$, $(t_i, t_{i+1}) \in \mathcal{R}$ (resp. $(t_{i+1}, t_i) \in \mathcal{R}$).*

Notons que si $\mathcal{R}$ est bien-fondée, alors $\mathcal{R}^{-1}$ est nœthérienne.

Proposition 9.3.1 *[FO97] Si $\mathcal{R}$ est bien-fondée alors $\mathcal{L}' = \mathcal{R}^*(\mathcal{L})$ ssi*

$$\mathcal{L}' = \mathcal{R}(\mathcal{L}') \cup \mathcal{L} \tag{9.3}$$

Par conséquent, quand $\mathcal{R}$ est bien-fondée, nous pouvons utiliser nos techniques d'élargissement pour calculer automatiquement des ensembles susceptibles d'être l'ensemble des accessibles, et utiliser le test (9.3) pour vérifier automatiquement qu'un ensemble calculé est vraiment égal à $\mathcal{R}^*(\mathcal{L})$.

Notons que ce test permet aussi de calculer de manière exacte la relation $\mathcal{R}^*$ si $\mathcal{R}$ est une relation de mots ou d'arbres bien-fondée :

Proposition 9.3.2 *Si $\mathcal{R}$ est une relation de mots ou d'arbres bien-fondée, alors $\mathcal{R}' = \mathcal{R}^*$ ssi*

$$\mathcal{R}' = (\mathcal{R} \circ \mathcal{R}') \cup \mathcal{R} \tag{9.4}$$

Preuve : Ceci est dû à la proposition 9.3.1 ; au fait que $\mathcal{R} \circ \mathcal{R}' = \mathcal{R}_1(\mathcal{R}')$, où $\mathcal{R}_1$ est la relation définie par $\mathcal{R}_1 = \{\big((t, t'), (t, t'')\big) \mid (t', t'') \in \mathcal{R}\}$; et au fait que $\mathcal{R}^* = \mathcal{R}_1^*(\mathcal{R})$. En effet, il est clair que $\mathcal{R}_1$ est bien fondée ssi $\mathcal{R}$ l'est. □

9.3.2 Exemples de relations bien-fondées

Nous donnons dans ce qui suit quelques exemples de relations bien-fondées. Pour ces relations, nous sommes sûrs qu'en cas de treminaison, nos techniques d'élargissement nous calcule *exactement* l'ensemble des accessibles.

9.3.2.1 Des relations de mots bien-fondées

Proposition 9.3.3 *Soit $\mathcal{R}$ une relation de la forme :*

$$\mathcal{R} = \mathcal{R}_1(a_1, b_1)\mathcal{R}_2(a_2, b_2).....(a_n, b_n)\mathcal{R}_{n+1}; \mathcal{R}_i \subseteq copy(\Sigma^*), \forall i, j,\ a_i \neq b_j. \tag{9.5}$$

alors $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont bien-fondées.

Preuve : Soit $\mathcal{R} = \mathcal{R}_1(a_1, b_1)\mathcal{R}_2(a_2, b_2).....(a_n, b_n)\mathcal{R}_{n+1}$ tels que pour tout i, $\mathcal{R}_i \subseteq copy(\Sigma^*)$ et pour tout i, j, $a_i \neq b_j$. Soit $w \in \Sigma^*$, $(w, w) \notin \mathcal{R}^+$ puisqu'on ne peut écrire qu'une seule fois dans un certain endroit, et donc après un nombre fini de réécritures, on ne peut jamais retrouver le mot de départ. Donc, $\mathcal{R}$ est nœthérienne. Il en est de même pour $\mathcal{R}^{-1}$ puisque $\mathcal{R}^{-1} = \mathcal{R}_1(b_1, a_1)\mathcal{R}_2(b_2, a_2).....(b_n, a_n)\mathcal{R}_{n+1}$. □

Définition 9.3.2 (Relation unaire simple) *Soit $\mathcal{R}$ une relation de la forme :*

$$\bigcup_{i=1}^{n} \mathcal{R}_i^1 \cdot \mathcal{R}_i \cdot \mathcal{R}_i^2 \tag{9.6}$$

où pour tout i, la relation $\mathcal{R}_i$ est un sous-ensemble de $\Sigma \times \Sigma$. Soit $\mathcal{R}_0 = \bigcup_{i=1}^{n} \mathcal{R}_i$. $\mathcal{R}$ est dite relation unaire simple *si $\mathcal{R}_0$ est acyclique (c-à-d., il n'existe pas $a \in \Sigma$ tel que $(a, a) \in \mathcal{R}_0^+$).*

Définition 9.3.3 (Semi-commutation simple) *Soit $\mathcal{R}$ une relation de la forme :*

$$\bigcup_{i=1}^{n} \mathcal{R}_i^1 \cdot \mathcal{R}_i \cdot \mathcal{R}_i^2 \tag{9.7}$$

où pour tout i, la relation $\mathcal{R}_i$ est un ensemble de paires de la forme (ab, ba) où $a, b \in \Sigma$ et $a \neq b$. Soit $\mathcal{R}_0 = \bigcup_{i=1}^{n} \mathcal{R}_i$. $\mathcal{R}$ est dite semi-commutation simple *si $\mathcal{R}_0$ est anti-symétrique.*

Définition 9.3.4 (Relation simple) *Une relation $\mathcal{R}$ est* simple *si elle est l'union d'une relation unaire simple et d'une semi-commutation simple.*

Lemme 9.3.1 *Si $\mathcal{R}$ est une relation unaire simple, alors $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont bien-fondées.*

Preuve : Soit $w \in \Sigma^*$, nous montrons que $(w, w) \notin \mathcal{R}^+$. En effet, si à une certaine position du mot, un symbole $a \in \Sigma$ est remplacé par un autre symbole b, on ne peut retrouver a dans cette position que si b a pu être réécrit en a (peut être en plusieurs étapes), ce qui contredit le fait que $\mathcal{R}$ est simple. De la même manière, nous pouvons montrer que $(w, w) \notin (\mathcal{R}^{-1})^+$. □

Lemme 9.3.2 *Si $\mathcal{R}$ est une semi-commutation simple, alors $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont bien-fondées.*

Preuve : Soit $\mathcal{R} = \bigcup_{i=1}^{n} \mathcal{R}_i^1 \cdot \mathcal{R}_i \cdot \mathcal{R}_i^2$, et soit $\mathcal{R}_0 = \bigcup_{i=1}^{n} \mathcal{R}_i$. Nous montrons que si $\mathcal{R}_0$ est antisymétrique, alors $\forall w \in \Sigma^*$, $(w, w) \notin \mathcal{R}^+$. Nous raisonnons par induction sur la longueur du mot w.

Soit w un mot non vide (si $w = \varepsilon$, la propriété est triviale puisqu'on ne peut pas appliquer la semi-commutation à w). Soit $w = a.w'$. Supposons que $(w, w) \in \mathcal{R}^+$, alors il existe une séquence $\sigma = w_0, w_1, \ldots, w_n$ telle que $w_0 = w_n = w$ et $\forall i \in \{0, \ldots, n-1\}$, $(w_i, w_{i+1}) \in \mathcal{R}$. Deux cas se présentent :

- Le symbole a à la première position de w n'est jamais déplacé dans la séquence σ. Ceci veut dire que $(w', w') \in \mathcal{R}^+$, ce qui contredit l'hypothèse d'induction (puisque $|w'| < |w|$).
- Il existe un indice $i < n$ tel que $w_i = a \cdot b \cdot u$, $w_{i+1} = b \cdot a \cdot u$, et $(a \cdot b, b \cdot a) \in \mathcal{R}$. Alors, il est facile de voir que a ne peut pas retourner à sa position initiale puisque $(b \cdot a, a \cdot b) \notin \mathcal{R}$ ($\mathcal{R}$ est antisymétrique). De ce fait, il est impossible de retrouver le mot w à partir de w_i.

$\mathcal{R}$ antisymétrique implique que $\mathcal{R}^{-1}$ l'est aussi, nous pouvons alors montré de la même manière que $\mathcal{R}^{-1}$ est nœthérienne. □

Théorème 9.3.1 *Si $\mathcal{R}$ est une relation simple, alors $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont bien-fondées.*

Preuve : Soient $\mathcal{R}_c$ la relation de semi-commutation de $\mathcal{R}$ et $\mathcal{R}_s$ sa relation unaire. Soit $(w, w') \in \mathcal{R}^+$. Nous montrons que $w \neq w'$. Il y a deux cas :

1. si $(w, w') \in \mathcal{R}_s^+$ ou $(w, w') \in \mathcal{R}_c^+$, alors les lemmes 9.3.1 et 9.3.2 impliquent que $w \neq w'$.
2. si $(w, w') \in (\mathcal{R}_s^+ \mathcal{R}_c^+)^+ \cup (\mathcal{R}_c^+ \mathcal{R}_s^+)^+$ alors $w' \neq w$ puisqu'après avoir appliqué la relation $\mathcal{R}_s$ qui remplace certaines lettres par d'autres ; le nombre d'occurrences des lettres présentes dans le mot va changer puisque $\mathcal{R}_s$ est acyclique. Ensuite, il n'est pas possible de retrouver un mot ayant le nombre d'occurrences initial de chaque lettre puisque $\mathcal{R}_s$ est acyclique.

□

9.3.2.2 Des relations d'arbres bien-fondées

Théorème 9.3.2 (WOS) *Soit $\mathcal{R}$ un well-oriented system, alors $\mathcal{R}$ et $\mathcal{R}^{-1}$ sont bien-fondées.*

Preuve : Ceci est dû au fait qu'après chaque application d'une règle de $\mathcal{R}_i$, le nombre de nœuds étiquetés par une lettre de $\mathcal{S}_i$ décroit strictement. □

Théorème 9.3.3 (Systèmes PA modulo =) *Soit $\mathcal{R}$ un système PA qui ne comprend pas de séquence de règles de la forme $(X_i \to X_{i+1})_{0 \leq i < n}$ telle que $X_n = X_0$; alors $\Rightarrow_{=,\mathcal{R}}$ est bien-fondée.*

Preuve : Ceci est dû au fait que si $t \Rightarrow_{=,\mathcal{R}} t'$, alors nous ne pouvons pas retrouver t à partir de t'. En effet, si $t \Rightarrow_{=,\mathcal{R}} t'$, alors :

- soit la taille de t est strictement inférieure à celle de t' (si t' est obtenu en appliquant une règle de la forme $X \rightarrow u$ où $u \notin Var \cup \{0\}$) et donc comme les règles d'un système PA ne réduisent pas la taille des termes, nous ne pouvons pas retrouver t ;
- soit $t = C[X]$ et $t' = C[0]$ (si t' est obtenu en appliquant une règle de la forme $X \rightarrow 0$). Dans ce cas, il n'est pas possible de retrouver t puisque dans un système PA il n'y a pas de règles de la forme $0 \rightarrow t$;
- soit $t = C[X]$ et $t' = C[Y]$ (si t' est obtenu en appliquant une règle de la forme $X \rightarrow Y$). Dans ce cas aussi, nous ne pouvons jamais retrouver t à partir de t' puisqu'il n'existe pas de séquences $(X_i \rightarrow X_{i+1})_{0 \leq i < n}$ telles que $X_n = X_0 = X$ et $X_1 = Y$.

□

9.4 Résultats de complétude

Dans cette section, nous montrons comment l'élargissement permet de calculer, pour certaines classes de relations et certaines classes de langages réguliers l'ensemble des accessibles ou, plus généralement, la clôture réflexive-transitive de la relation en question. Nous commençons la section par les langages de mots. Nous montrons d'abord que l'élargissement permet de calculer un transducteur de mots qui reconnaît la relation $\mathcal{R}^*$ pour toute *context-relation* $\mathcal{R}$, une classe de relations définie dans [ABJN99] dans le but de modéliser les systèmes paramétrés linéaires. Dans la deuxième partie, nous montrons comment l'élargissement permet de calculer la fermeture d'une expression APC par semi-commutations. Nous passons ensuite aux langages d'arbres. Nous montrons d'abord que notre méthode calcule $\mathcal{R}^*(\mathcal{L})$ pour tout langage régulier d'arbres $\mathcal{L}$ et toute relation semi-monadique linéaire bien-fondée $\mathcal{R}$ (les relations semi-monadiques linéaires ont été définies dans [CDGV94]). Ensuite, nous montrons que l'élargissement permet de calculer $Post^*_{\mathcal{R},=}(\mathcal{L})$ pour tout langage régulier d'arbres $\mathcal{L}$, et tout système PRS $\mathcal{R}$ tel que $\Rightarrow_{=,\mathcal{R}}$ est bien-fondé. Nous en déduisons que notre technique permet de calculer cet ensemble pour tout système PA $\mathcal{R}$. Finalement, nous prouvons que cette méthode calcule le réétiquetage $\mathcal{R}^*$ pour toute relation WOS $\mathcal{R}$.

9.4.1 Les context-relations

Dans [ABJN99], Abdulla et al. ont introduit une classe de relations régulières unaires simples, que nous appelons ici *context-relations*, pour laquelle ils caractérisent $\mathcal{R}^*$ par un transducteur fini de mots. Dans cette section, nous montrons que nos techniques d'élargissement permettent de calculer un transducteur reconnaissant $\mathcal{R}^*$ pour chaque context-relation $\mathcal{R}$. Ce résultat permet d'analyser les protocoles d'exclusion mutuelle de Burns, de Dijkstra, de Szymanski, et le "Bakery algorithm" par élargissement puisque tous ces protocoles sont décrits par des context-relations (voir la section 9.6 et l'annexe A). Nous commençons par donner la définition d'une context-relation.

Définition 9.4.1 *Une context-relation est une relation de la forme*

$$\mathcal{R} = copy(L_L)\mathcal{R}_0 copy(L_R)$$

telle que :

- *L_L est un langage régulier qui peut être accepté par un automate déterministe de mots ayant un seul état final, et tel que toutes les transitions sortant de l'état final sont des boucles sur le même état. La* fin *de L_L est l'ensemble de symboles qui étiquettent les boucles de l'état final.*
- *L_R est un langage régulier tel que son langage miroir L'_R satisfait les conditions ci-dessus. La* fin *de L_R est la fin de L'_R.*
- *$\mathcal{R}_0 = \{(a_i, b_i) \mid \forall i, j, a_i \neq b_j\}$.*

Théorème 9.4.1 *Les techniques d'élargissement permettent de calculer $\mathcal{R}^*$ pour toute context-relation $\mathcal{R}$.*

Preuve : Soit $\mathcal{R} = copy(L_L)\mathcal{R}_0 copy(L_R)$ une context-relation. Soit Σ_L (resp. Σ_R) la fin de L_L (resp. la fin de L_R). Comme dans [ABJN99], nous utilisons les abbréviations c_L pour $c \in \Sigma_L$, c'_L pour $c' \in \Sigma_L$, c_R pour $c \in \Sigma_R$ et c'_R pour $c' \in \Sigma_R$. Dans les transducteurs des figures ci-dessous, nous ne montrons que la partie qui est entre l'état final q_L d'un transducteur qui recopie les mots de L_L, et l'état initial q_R d'un transducteur qui recopie les mots de L_R. En plus des transitions représentées ci-dessous, il y a des self-loops étiquetées par $c = c' \in \Sigma_L \cap \Sigma_R$ à tous les états. Nous écrivons $\mathcal{R}_0(c, c')$ à la place de $(c, c') \in \mathcal{R}_0$.

Nous allons montrer que l'élargissement permet de calculer $\mathcal{R}^*$. Comme $\mathcal{R}$ est bien-fondée (lemme 9.3.1), nous ne nous préoccupons pas des cas où l'élargissement peut calculer des mauvais ensembles puisque ces cas seront écartés par le test 9.4 (Proposition 9.3.2).

Soit $\mathcal{T}$ le transducteur de $\mathcal{R}$ représenté sur la figure 9.10. Nous calculons $\mathcal{T}^2$, $\mathcal{T}^3$, $\mathcal{T}^4$, et $\mathcal{T}^5$ en composant à chaque fois par le transducteur $\mathcal{T}$ comme décrit à la section 2.1.2.3. Par exemple, les transducteurs $\mathcal{T}^2$ et $\mathcal{T}^3$ sont représentés sur les figures 9.11 et 9.12. En comparant le transducteur $\mathcal{T}^3$ à $\mathcal{T}^5$, nous pouvons deviner une croissance, extrapoler et obtenir un transducteur $\mathcal{T}'$. En rajoutant $\mathcal{T}$ et $\mathcal{T}^2$ à $\mathcal{T}'$, nous obtenons excatement le transducteur de la figure 9.13 qui correspond à $\mathcal{R}^+$ [ABJN99].

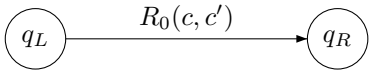


FIG. 9.10 – Transducteur $\mathcal{T}$

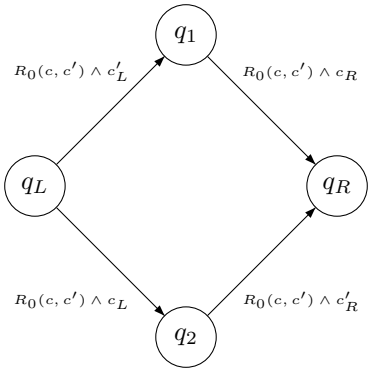


FIG. 9.11 – Transducteur $\mathcal{T}^2$

□

9.4.2 Calcul de la clôture d'une expression APC par semi-commutations

Rappelons qu'une expression APC est une somme de produits, où un produit est une concaténation de lettres de Σ et d'expressions étoiles de Σ, c-à-d. d'expressions de la forme $(a_1 + \cdots + a_n)^*$, où les a_i sont dans Σ (voir la section 3.1.2.1). Rappelons également qu'une relation de semi-commutations est une union finie de relations de la forme $copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$.

Dans ce qui suit, nous montrons que les techniques d'élargissement permettent de calculer la clôture d'une expression APC par semi-commutations. Pour obtenir des résultats exacts, nous utilisons le test (9.3).

Théorème 9.4.2 *$\mathcal{R}^*(\mathcal{L})$ peut être calculé par élargissement pour tout langage APC $\mathcal{L}$ et toute relation de semi-commutations $\mathcal{R}$.*

Preuve : Soit une relation de semi-commutations $\mathcal{R} = \cup_i \mathcal{R}_i$, où les $\mathcal{R}_i$ sont des relations de semi-commutations de la forme

$$\mathcal{R}_i = copy(\Sigma^*)(ab, ba)copy(\Sigma^*),$$

et soit $\mathcal{L}$ un langage APC donné sous forme d'une expression APC. Nous supposons sans perte de généralité que $\mathcal{R}$ est bien-fondée. En effet, si ce n'est pas le cas, nous construisons itérativement un langage APC $\mathcal{L}'$ et une relation de semi-commutations $\mathcal{R}'$ telles que $\mathcal{R}'$ soit bien-fondée et $\mathcal{R}^*(\mathcal{L})$ se déduit de $\mathcal{R}'^*(\mathcal{L})$ de la manière suivante : $\mathcal{L}'$ est la limite d'une séquence de langages APC $\mathcal{L}'_0, \mathcal{L}'_1, \ldots$ et $\mathcal{R}'$ est la limite d'une séquence de relations de semi-commutations $\mathcal{R}'_0, \mathcal{R}'_1, \ldots$ définies comme suit :

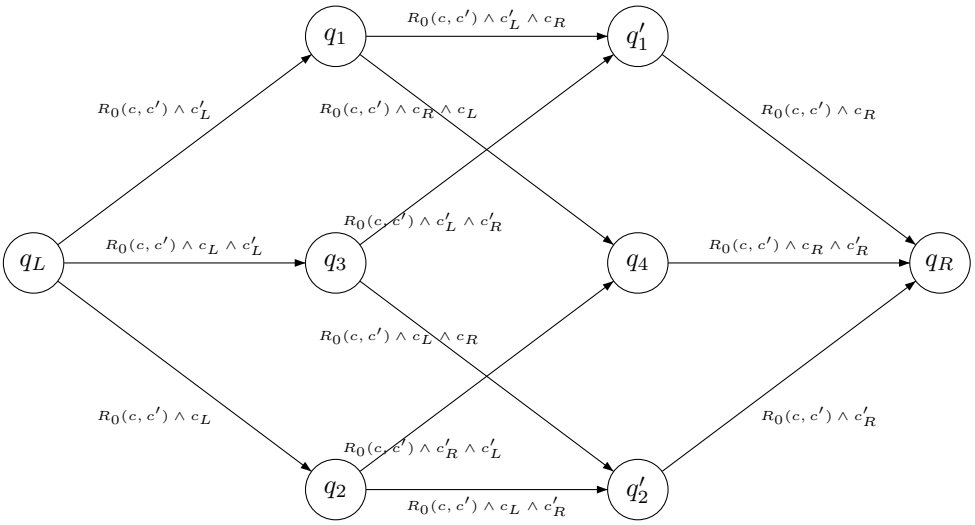


FIG. 9.12 – Transducteur $\mathcal{T}^3$

- $\mathcal{R}'_0 = \mathcal{R}$ et $\mathcal{L}'_0 = \mathcal{L}$.
- si la relation de semi-commutations $\mathcal{R}'_k = \cup_i \mathcal{R}^k_i$ n'est pas bien-fondée, alors il existe forcément deux indices i et j tels que $\mathcal{R}^k_i = copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$ et $\mathcal{R}^k_j = copy(\Sigma^*)(ba, ab)copy(\Sigma^*)$ (lemme 9.3.2). Dans ce cas, nous rajoutons deux lettres a_k et b_k à l'alphabet et $\mathcal{R}'_{k+1}$ et $\mathcal{L}'_{k+1}$ sont obtenus comme suit :
  - $\mathcal{R}'_{k+1}$ est obtenu à partir de $\mathcal{R}'_k$ en remplaçant $\mathcal{R}^k_j$ par la relation

$$copy(\Sigma^*)(b_k a_k, a_k b_k)copy(\Sigma^*),$$

    et en substituant dans toutes les autres relations $\mathcal{R}^k_l$, $l \neq i$, la lettre a par $a + a_k$ et la lettre b par $b + b_k$.
  - $\mathcal{L}'_{k+1}$ est obtenu en substituant dans $\mathcal{L}'_k$ toute occurrence de la lettre a par $a + a_k$ et toute occurrence de la lettre b par $b + b_k$.

$\mathcal{R}'$ (resp. $\mathcal{L}'$) est la limite de la suite $(\mathcal{R}'_k)$ (resp. $\mathcal{L}'_k$). Il est clair que ces suites sont finies. Finalement, $\mathcal{R}^*(\mathcal{L})$ est obtenu en substituant dans $\mathcal{R}'^*(\mathcal{L}')$, pour chaque k, toute occurrence de la lettre a_k par a et toute occurrence de la lettre b_k par b. Cette transformation est basée sur le fait qu'une relation de semi-commutations $\mathcal{R}$ peut s'appliquer une infinité de fois à un mot w, mais ne modifie pas sa longueur. Comme il y a un nombre fini de mots de longueur $|w|$, l'idée est de considérer à la place de $\mathcal{R}$ une relation de semi-commutations qui permet de générer à partir de w tous les mots qui sont aussi produits par $\mathcal{R}$, mais qui ne boucle pas.

Soit alors $\mathcal{R} = \bigcup_{(a,b)\in\rho_{\mathcal{R}}} copy(\Sigma^*)(ab, ba)copy(\Sigma^*)$ une relation de semi-commutations bien-fondée donnée sous la forme du transducteur de mots $\mathcal{T} = (Q, I, \Sigma, \delta, F)$ tel que

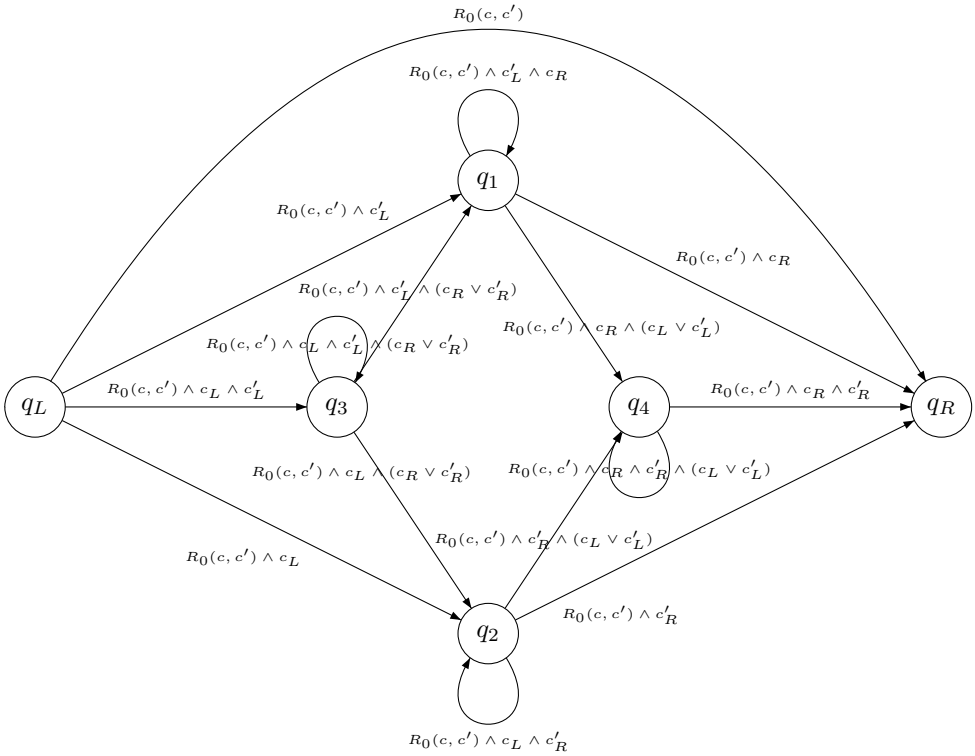


FIG. 9.13 – Transducteur de $\mathcal{R}^+$

$Q = \{q_I, q_F\} \cup \{q_{a,b} \mid (a,b) \in \rho_{\mathcal{R}}\}$, $I = \{q_I\}$, $F = \{q_F\}$, et δ est l'ensemble des règles suivantes :

- $\big(q_I, copy(\Sigma), q_I\big)$;
- $\big(q_F, copy(\Sigma), q_F\big)$;
- $\big(q_I, (a,b), q_{a,b}\big)$, et $\big(q_{a,b}, (b,a), q_F\big)$ pour chaque $(a,b) \in \rho_{\mathcal{R}}$.

Soit e une expression APC. Comme e est une somme finie de produits, pour montrer que l'élargissement permet de calculer $\mathcal{R}^*(e)$, il suffir de montrer qu'il permet de calculer $\mathcal{R}^*(p)$ pour un produit p. Soit alors un produit $p = e_1 e_2 \cdots e_n$ donné sous la forme d'un automate $\mathcal{A}$ qui préserve la structure du produit, c-à-d., qui comprend un seul état initial, un seul état final, et une seule "branche" menant de l'état initial à l'état final et dont l'étiquette correspond exactement au produit p. Plus précisément, p peut s'écrire sous la forme $A_1^* a_1 A_2^* a_2 \cdots a_{n-1} A_n^*$, où les A_i sont des ensembles de lettres de Σ qui peuvent être vides (dans ce cas, $A_i^* = \epsilon$) ; et les $a_i \in \Sigma \cup \{\epsilon\}$. Dans ce cas, $\mathcal{A} = (Q, I, \Sigma, \delta, F)$, où $Q = \{q_i \mid i \leq n+1\}$, $I = \{q_1\}$, $F = \{q_n\}$, et δ contient les transitions suivantes : (q_i, a, q_i) si $a \in A_i$, $i \leq n$; et (q_i, a_i, q_{i+1}) pour $i < n$.

Nous montrons que pendant le calcul itératif de $\mathcal{T}(\mathcal{A}), \mathcal{T}^2(\mathcal{A}), \ldots$, notre principe d'élargissement peut détecter une situation où l'extrapolation calcule un automate qui reconnaît exactement la clôture $\mathcal{R}^*(p)$. Nous ne nous préoccupons pas des cas où de mauvaises extrapolations peuvent avoir lieu puisque $\mathcal{R}$ est bien-fondée, et donc toutes les mauvaises extrapolations seront rejetées par le test (9.3).

Nous savons d'après la section 3.1.2.2 que $\mathcal{R}^*(p)$ est une APC. Soient alors des expressions atomique l_i^j telles que

$$\mathcal{R}^*(p) = \sum_i l_1^i l_2^i \cdots l_{k_i}^i$$

où les l_i^j sont soit des expressions étoiles ou des lettres qui figurent déjà parmi les e_i (il existe un indice k tel que $l_i^j = e_k$), soit des expressions étoiles qui se sont rajoutées (voir l'algorithme décrit dans la section 3.1.2.2).

Il est alors facile de voir qu'après un nombre fini m d'itérations, nous obtenons un automate $\mathcal{T}^m(\mathcal{A})$ qui contient des branches menant de l'état initial à l'état final étiquetées par ${l'}^i_{1,g} {l'}^i_{2,g} \cdots {l'}^i_{k_i,g}$, c'est à dire qui reconnaît $\mathcal{R}^m(p) = \sum_i \sum_g {l'}^i_{1,g} {l'}^i_{2,g} \cdots {l'}^i_{k_i,g}$, où

- si $l_i^j = e_k$, alors ${l'}^j_{i,g} = e_k$;
- sinon, si $l_i^j = (a_1 + \cdots + a_h)^*$, ${l'}^j_{i,g} = b_1 \cdots b_{h'_g}$, où les b_i sont des lettres parmi $\{a_1, \ldots, a_h\}$.

Après un nombre supplémentaire fini d'itérations, nous obtenons un automate $\mathcal{T}^{m'}(\mathcal{A})$ qui contient des branches menant de l'état initial à l'état final étiquetées par ${l''}^i_{1,g,s} {l''}^i_{2,g,s} \cdots {l''}^i_{k_i,g,s}$, c'est à dire qui reconnaît $\mathcal{R}^{m'}(p) = \sum_i {l''}^i_{1,g,s} {l''}^i_{2,g,s} \cdots {l''}^i_{k_i,g,s}$, où

- si $l_i^j = e_k$, alors ${l''}^j_{i,g,s} = e_k$;
- sinon, si $l_i^j = (a_1 + \cdots + a_h)^*$ et que ${l'}^j_{i,g} = b_1 \cdots b_{h'}$, il existe une séquence d'indices $q_1, \ldots, q_k$ et une suite de séquences de lettres $(c_1^r, \ldots, c_{h_s^r}^r)_{1 \le r \le k}$ tels que [3] pour tout r, $\{c_1^r, \ldots, c_{h_s^r}^r\} = \{a_1, \ldots, a_h\}$ et

$${l''}^j_{i,g,s} = b_1 \cdots b_{q_1} c_1^1 \cdots c_{h_s^1}^1 b_{q_1+1} \cdots b_{q_2} c_1^2 \cdots c_{h_s^2}^2 \cdots\cdots\cdots b_{q_k} c_1^k \cdots c_{h_s^k}^k b_{q_k+1} \cdots b_{h'}.$$

En comparant $\mathcal{T}^{m'}(\mathcal{A})$ à $\mathcal{T}^m(\mathcal{A})$, nous détectons des croissances qui correspondent aux séquences "$c_1^r \cdots c_{h_s^r}^r$". L'élargissement peut alors calculer un automate $\mathcal{A}'$ dont les branches sont étiquetées par $A^i_{1,g} A^i_{2,g} \cdots A^i_{k_i,g}$, c-à-d. un automate qui reconnaît le langage APC $\sum_i \sum_g \sum_s A^i_{1,g,s} A^i_{2,g,s} \cdots A^i_{k_i,g,s}$, où

- si $l_i^j = e_k$, alors $A^j_{i,g} = e_k$;
- sinon, si $l_i^j = (a_1 + \cdots + a_h)^*$, ${l'}^j_{i,g} = b_1 \cdots b_{h'}$, et

$${l''}^j_{i,g,s} = b_1 \cdots b_{q_1} c_1^1 \cdots c_{h_s^1}^1 b_{q_1+1} \cdots b_{q_2} c_1^2 \cdots c_{h_s^2}^2 \cdots\cdots\cdots b_{q_k} c_1^k \cdots c_{h_s^k}^k b_{q_k+1} \cdots b_{h'}$$

alors $A^j_{i,g,s} = b_1 \cdots b_{q_1} (a_1 + \cdots + a_h)^* b_{q_1+1} \cdots b_{q_2} (a_1 + \cdots + a_h)^* \cdots\cdots\cdots b_{q_k} (a_1 + \cdots + a_h)^* b_{q_k+1} \cdots b_{h'}$.

[3] on aurait dû écrire $q_{1,s}, \ldots, q_{k_s,s}$ puisque ces indices dépendent de s ; et $(c^r_{1,s}, \ldots, c^r_{h^r_s,s})_{1 \le r \le k_s}$ puisque ces lettres dépendent aussi de s. Ici, nous omettons cet indexage par s pour des raisons de lisibilité.

En effet, considérons par exemple le cas où $k = 1$; c-à-d., ${l'}^{j}_{i,g} = b_1 \cdots b_{h'}$, et ${l''}^{j}_{i,g,s} = b_1 \cdots b_q c_1 \cdots c_{h''} b_{q+1} \cdots b_{h'}$, alors dans le graphe correspondant à $\mathcal{T}^m(\mathcal{A})$, il y a une partie qui comprend des noeuds v_i, $0 \leq i \leq h'$, et les arcs (v_i, b_{i+1}, v_{i+1}). C'est la partie qui correspond à ${l'_{i,g}}^{j} = b_1 \cdots b_{h'}$. De même, dans le graphe correspondant à $\mathcal{T}^{m'}(\mathcal{A})$, la partie correspondant à ${l''}^{j}_{i,g,s} = b_1 \cdots b_q c_1 \cdots c_{h''} b_{q+1} \cdots b_{h'}$ comprend des noeuds v_i, $0 \leq i \leq h'$, v'_i, $1 \leq i \leq h''$, et les arcs (v_i, b_{i+1}, v_{i+1}), (v_q, c_1, v'_1), $(v'_i, c_{i+1}, v'_{i+1})$, $(v'_{h''}, b_{q+1}, v_{q+1})$. En comparant ces deux parties, nous pouvons détecter comme croissance le graphe Δ ayant v_q et v'_i, $1 \leq i \leq h''$ comme noeuds et (v_q, c_1, v'_1), $(v'_i, c_{i+1}, v'_{i+1})$ comme arcs. Si nous prenons $\mathcal{I}_\Delta = \{v_q\} \cup \{v'_i \mid 1 \leq i \leq h''\}$, et φ la partition $\{v_q, v'_i \mid 1 \leq i \leq h''\}$ qui permet de fusionner tous ces états, en enlevant ce Δ, nous obtenons exactement $b_1 \cdots b_{h'}$, et en extrapolant, c-à-d., en fusionnant tous ces états, nous obtenons

$$b_1 \cdots b_q (a_1 + \cdots + a_h)^* b_{q+1} \cdots b_{h'}.$$

Il est alors clair qu'il existe une itération m telle que $\mathcal{R}^*(p)$ est obtenu en calculant l'union de cet automate $\mathcal{A}'$ avec les automates $\mathcal{T}^i(\mathcal{A})$, $0 \leq i \leq m'$.

□

9.4.3 Relations semi-monadiques linéaires

Nous montrons dans cette section que l'élargissement permet de calculer $\mathcal{R}^*(\mathcal{L})$ pour tout langage régulier d'arbres $\mathcal{L}$ et toute relation semi-monadique linéaire bien-fondée $\mathcal{R}$. Les relations semi-monadiques linéaires sont définies par :

Définition 9.4.2 (Relations semi-monadiques linéaires [CDGV94]) *Un système de réécriture S sur un alphabet Σ muni d'une fonction d'arité est* semi-monadique linéaire *si, pour chaque règle $l(x_1, \ldots, x_k) \to r$ de S, le membre droit r est soit une variable, soit un terme linéaire $f(r_1, \ldots, r_n)$ où $f \in \Sigma_n$, et pour tout i, $1 \leq i \leq n$, r_i est une variable parmi $\{x_1, \ldots, x_k\}$ ou un terme clos. Un système de réécriture est clos si toutes ses règles sont de la forme $l \to r$ où l et r sont des termes clos.*

Un système de réécriture semi-monadique linéaire S induit une relation semi-monadique linéaire $\mathcal{R}_S$ *sur les termes de T_Σ définie par l'ensemble des paires de termes (t, t') telles qu'il existe un contexte C, des termes $u_1, \ldots, u_k$ dans T_Σ tels que $t = C\big[l[x_i \leftarrow u_i]\big]$ et $t' = C\big[r[x_i \leftarrow u_i]\big]$; où $l[x_i \leftarrow u_i]$ (resp. $r[x_i \leftarrow u_i]$) est le terme obtenu en substituant dans l (resp. dans r) chaque occurrence des variables x_i par le terme u_i, et ce pour chaque i, $1 \leq i \leq k$.*

Il est bien connu que les systèmes de réécritures clos préservent effectivement la régularité [Bra69, DT90]. Ce résultat a été étendu aux relations semi-monadiques linéaires par Coquidé, Dauchet, Gilleron, et Vágvölgyi :

Théorème 9.4.3 *[CDGV94] Soit S un système de réécriture semi-monadique linéaire sur Σ, $\mathcal{R}_S$ la relation qu'il induit, et $\mathcal{L}$ un langage régulier d'arbres sur T_Σ, alors $\mathcal{R}_S^*(\mathcal{L})$ est un langage régulier d'arbres et est effectivement constructible.*

Preuve : Nous donnons la construction de [CDGV94]. Soit $Sub(S)$ l'ensemble de tous les sous termes clos des membres droits des règles de S. Soit $Q_S = \{q_t \mid t \in Sub(S)\}$, et soit δ_S l'ensemble suivant de règles de transition :

- $a \to q_a$ pour chaque $a \in Sub(S) \cap \Sigma_0$,
- $f(q_{t_1}, \ldots, q_{t_n}) \to q_{f(t_1,\ldots,t_n)}$ si $f(t_1, \ldots, t_n) \in Sub(S)$.

Il est clair qu'avec δ_S, pour chaque terme $t \in Sub(S)$, l'état q_t accepte $\{t\}$ ($L_{q_t} = \{t\}$). Soit $\mathcal{A} = (Q, \Sigma, F, \delta)$ un automate d'arbres fini. Nous définissons l'automate $\mathcal{A}^* = (Q', \Sigma, F', \delta')$ comme suit :

- $Q' = Q \cup Q_S$.
- $F' = F$,
- δ' est le plus petit ensemble de règles contenant $\delta \cup \delta_S$ et tel que pour chaque états $q, q_1, \ldots, q_n \in Q \cup Q_S$:

  (α_1) si $l(x_1, \ldots, x_k) \to f(r_1, \ldots, r_{k'}) \in S$ et $l(q_1, \ldots, q_k) \xrightarrow{*}_{\delta'} q$, alors $f(q'_1, \ldots, q'_{k'}) \to q$ est dans δ' avec $q'_j = q_t$ si $r_j = t \in T_\Sigma$, et $q'_j = q_i$ si $r_j = x_i$ pour $j \in \{1, \ldots, k'\}$,

  (α_2) si $l(x_1, \ldots, x_k) \to x_j \in S$ et $l(q_1, \ldots, q_k) \xrightarrow{*}_{\delta'} q$, alors $q_j \to q$ est dans δ'.

Alors, $\mathcal{L}(\mathcal{A}^*) = \mathcal{R}_S^*(\mathcal{L})$. Plus précisément, nous pouvons montrer que pour chaque état q, $L'_q = \mathcal{R}_S^*(L_q)$ où L_q est le langage reconnu par l'état q dans l'automate $\mathcal{A}$, et L'_q est le langage reconnu par l'état q dans le nouvel automate construit $\mathcal{A}^*$. La preuve est très similaire à celle du théorème 7.6.1.

□

Nous montrons dans ce qui suit que l'élargissement permet de calculer cet automate.

Théorème 9.4.4 *Soit S un système de réécriture semi-monadique linéaire sur Σ, $\mathcal{R}_S$ la relation qu'il induit, et $\mathcal{L}$ un langage régulier d'arbres sur T_Σ. S'il existe un test qui permet de décider si un langage donné $\mathcal{L}'$ est égal à $\mathcal{R}_S^*(\mathcal{L})$, alors $\mathcal{R}_S^*(\mathcal{L})$ peut être calculé par élargissement.*

Preuve : Comme nous avons un test qui permet de décider si un langage donné $\mathcal{L}'$ est égal à $\mathcal{R}_S^*(\mathcal{L})$, nous allons montrer que l'élargissement permet de calculer l'automate $\mathcal{A}^*$ décrit dans la preuve ci-dessus sans nous préoccuper des cas où de mauvaises extrapolations peuvent être faites puisque cettes dernières seront rejetées par le test. Nous reprenons donc les notations considérées dans la preuve précédente. Comme il n'y a pas de confusion possible, nous notons la relations $\mathcal{R}_S$ simplement par $\mathcal{R}$. Il est facile de voir que $\mathcal{R}^{\leq j}(\mathcal{L})$ est reconnu par l'automate $\mathcal{A}^j = (Q^j, \Sigma, F^j, \delta^j)$ défini comme suit (ces automates sont obtenus en composant à chaque fois par un transducteur $\mathcal{T}$ représentant $\mathcal{R}$ qu'il est facile de décrire, m mais que nous ne représentons pas ici) :

- $Q^0 = Q \cup Q_S$, $\delta^0 = \delta \cup \delta_S$;
- $Q^j = Q^{j-1} \cup \{q^T \mid q \in Q^{j-1}\}$. Nous introduisons la notation q^{T^j} définie par : $q^{T^0} = q$ et $q^{T^{j+1}} = (q^{T^j})^T$;
- $F^j = \{q^T \mid q \in F^{j-1}\}$;
- δ^j est le plus petit ensemble de règles contenant δ^{j-1} et tel que pour chaque états $q \in Q^0$ et $q_1, \ldots, q_n \in Q^{j-1}$:

  1. si $q_1 \to q_2 \in \delta^{j-1}$ alors $q_1^T \to q_2^T \in \delta^j$;

2. si $f(q_1, \ldots, q_n) \to q^{T^{j-1}} \in \delta^{j-1}$ alors pour chaque i, $1 \leq i \leq n$,

$$f(q_1, \ldots, q_{i-1}, q_i^T, q_{i+1}, \ldots, q_n) \to q^{T^j} \in \delta^j,$$

3. si $l(x_1, \ldots, x_k) \to f(r_1, \ldots, r_{k'}) \in S$ et $l(q_1, \ldots, q_k) \xrightarrow{*}_{\delta^{j-1}} q^{T^{j-1}}$, alors $f(q'_1, \ldots, q'_{k'}) \to q^{T^j}$ est dans δ^j avec $q'_h = q_t$ si $r_h = t \in T_\Sigma$, et $q'_h = q_i$ si $r_h = x_i$ pour $h \in \{1, \ldots, k'\}$,
4. si $l(x_1, \ldots, x_k) \to x_h \in S$ et $l(q_1, \ldots, q_k) \xrightarrow{*}_{\delta^{j-1}} q^{T^{j-1}}$, alors $q_h \to q^{T^j}$ est dans δ^j.

Il est alors facile de voir que pour tout $q \in Q \cup Q_S$, $L'_{q^{T^j}} = \mathcal{R}^j(L_q)$ où L_q est le langage reconnu par l'état q dans l'automate $\mathcal{A}$, et $L'_{q^{T^j}}$ est le langage reconnu par l'état q^{T^j} dans l'automate construit $\mathcal{A}^j$. Ceci peut se montrer facilement comme fait dans la preuve du théorème 7.6.1. Nous expliquons dans ce qui suit l'intuition exprimée par les règles (1)–(4) :

- Les règles (1) expriment que si $L_{q_1} \subseteq L_{q_2}$, alors $\mathcal{R}^j(L_{q_1}) \subseteq \mathcal{R}^j(L_{q_2})$;
- Les règles (2) expriment que si $f(t_1, \ldots, t_n) \in \mathcal{R}^{j-1}(L_q)$, et $t'_i \in \mathcal{R}(t_i)$, alors $f(t_1, \ldots, t'_i, \ldots, t_n) \in \mathcal{R}^j(L_q)$;
- Les règles (3) expriment que si $l(x_1, \ldots, x_k) \to f(r_1, \ldots, r_{k'}) \in S$ et $l(t_1, \ldots, t_k) \in \mathcal{R}^{j-1}(L_q)$, alors $r(u_1, \ldots, u_{k'}) \in \mathcal{R}^j(L_q)$ où $u_h = t$ si $r_h = t \in T_\Sigma$, et $u_h = t_i$ si $r_h = x_i$ pour $h \in \{1, \ldots, k'\}$;
- De la même manière, les règles (4) expriment que si $l(x_1, \ldots, x_k) \to x_h \in S$ et $l(t_1, \ldots, t_k) \in \mathcal{R}^{j-1}(L_q)$, alors $t_h \in \mathcal{R}^j(L_q)$.

De par les constructions des règles (3) et (α_1), pour chaque règle α de la forme $f(q_1, \ldots, q_n) \to q$ rajoutée par (α_1), il existe un indice j et des indices $i_1, \ldots, i_n$ tels que $f(q_1^{T^{i_1}}, \ldots, q_n^{T^{i_n}}) \to q^{T^j}$ est une règle de δ_j. Soit $m(\alpha)$ le minimum de tels indices j.

De même chaque règle α de la forme $q_i \to q$ rajoutée par (α_2), il existe des indices j et $k \leq j$ tels que $q_i^{T^k} \to q^{T^j}$ est une règle de δ_j. De la même manière, soit $m(\alpha)$ le minimum de tels indices j.

Soit $m = \max m(\alpha)$, le maximum étant pris sur toutes les règles α de δ' qui ont été rajoutées par les règles d'inférence (α_1) et (α_2). Il est clair que pour chaque règle de la forme $f(q_1, \ldots, q_n) \to q$ rajoutée par (α_1), il existe une règle de la forme $f(q_1^{T^{i_1}}, \ldots, q_n^{T^{i_n}}) \to q^{T^j}$ dans δ^m. De même, pour chaque règle de la forme $q_i \to q$ rajoutée par (α_2), il existe une règle de la forme $q_i^{T^k} \to q^{T^j}$ dans δ^m.

Si nous comparons les hypergraphes $\mathcal{G}$ de $\mathcal{A}^0$ et $\mathcal{G}'$ celui de $\mathcal{A}^m$, nous remarquons des croissances définies par l'hypergraphe Δ défini par :

- $V_\Delta = Q^m$;
- H_Δ est l'ensemble des hyperarcs définis par les règles de $\delta^m \setminus \delta^0$;
- $\mathcal{I}_\Delta = \{q^{T^i} \mid q \in Q^0, 0 \leq i \leq m\}$.

Considérons la partition φ suivante : $\big\{\{q^{T^i} \mid i \geq 0\} \mid q \in Q \cup Q_S\big\}$. Il y a autant de classes que d'états dans $Q \cup Q_S$.

Il est facile de voir que

$$(\mathcal{G}, F) = \big(\mathcal{G}' \setminus_\varphi \Delta, F\big)$$

Donc, l'élargissement nous calcule

$$\nabla(\mathcal{A}^0, \mathcal{A}^m, \Delta, \varphi) = (\mathcal{G}'[\Delta \leftarrow \Delta^+], F)$$

et il est facile de voir que le nouvel hypergraphe représente exactement l'automate $\mathcal{A}^*$. En effet, dans le nouvel hypergraphe, pour chaque état q de Q, les états de la forme q^{T^i} sont remplacés par un état $s_{[q^{T^i}]}$, c-à-d. par un état unique que l'on peut appeler s_q puisque pour tout i, $[q] = [q^{T^i}]$. Et donc, puisqu'à chaque règle de la forme $f(q_1, \ldots, q_n) \rightarrow q$ rajoutée par (α_1), il existe une règle de la forme $f(q_1^{T^{i_1}}, \ldots, q_n^{T^{i_n}}) \rightarrow q^{T^j}$ dans δ^m, nous aurons après fusion des états une règle de la forme $f(s_{q_1}, \ldots, s_{q_n}) \rightarrow s_q$ et qu'à chaque règle de la forme $q_i \rightarrow q$ rajoutée par (α_2), il existe une règle de la forme $q_i^{T^k} \rightarrow q^{T^j}$ dans δ^m, nous aurons après fusion des états une règle de la forme $s_{q_i} \rightarrow s_q$. De même, les règles de la forme $f(q_1, \ldots, q_n) \rightarrow q$ de Q^0 sont elle aussi remplacées par les règles $f(s_{q_1}, \ldots, s_{q_n}) \rightarrow s_q$. Il y a clairement un isomorphisme immédiat entre cet hypergraphe et celui de $\mathcal{A}^*$: cet isomorphisme associe à chaque état s_q l'état q. □

Nous obtenons le résultat suivant comme conséquence immédiate de ce théorème :

Corollaire 9.4.1 *Soit S un système de réécriture semi-monadique linéaire sur Σ tel que la relation qu'il induit $\mathcal{R}_S$ est bien-fondée, et $\mathcal{L}$ un langage régulier d'arbres sur T_Σ. Alors $\mathcal{R}_S^*(\mathcal{L})$ peut être calculé par élargissement.*

Puisque les systèmes de réécritures clos sont semi-monadiques linéaires, le corollaire ci-dessus implique que l'élargissement permet de retrouver le résultat de [Bra69, DT90] pour les systèmes clos s'ils sont bien-fondés :

Corollaire 9.4.2 *Soit S un système de réécriture clos sur Σ tel que la relation qu'il induit $\mathcal{R}_S$ est bien-fondée, et $\mathcal{L}$ un langage régulier d'arbres sur T_Σ. Alors $\mathcal{R}_S^*(\mathcal{L})$ peut être calculé par élargissement.*

9.4.4 Systèmes PRS

Nous montrons dans cette section que l'élargissement permet de calculer $Post^*_{\mathcal{R},=}(\mathcal{L})$ pour tout langage régulier $\mathcal{L}$ et tout système PRS $\mathcal{R}$ tel que $\Rightarrow_{=,\mathcal{R}}$ est bien-fondée.

Théorème 9.4.5 *Soit $\mathcal{R}$ un PRS sur Var et $\mathcal{L}$ un langage régulier d'arbres sur Var. S'il existe un test qui permet de décider si un langage donné $\mathcal{L}'$ est égal à $Post^*_{\mathcal{R},=}(\mathcal{L})$, alors $Post^*_{\mathcal{R},=}(\mathcal{L})$ peut être calculé par élargissement.*

Preuve : Nous reprenons la construction de $\mathcal{A}^*_{\mathcal{R}}$ donnée dans la section 5.3. Comme précédemment, puisque nous avons un test qui permet de décider si un langage donné $\mathcal{L}'$ est égal à $Post^*_{\mathcal{R},=}(\mathcal{L})$, nous allons montrer que l'élargissement permet de calculer l'automate $\mathcal{A}^*_{\mathcal{R}}$ sans nous préoccuper des cas où de mauvaises extrapolations peuvent être faites puisque cettes dernières seront rejetées par le test. Nous reprenons donc les notations considérées dans la section 5.3. De la même manière que précédemment, il est facile de voir que $Post^{\leq j}_{\mathcal{R},=}(\mathcal{L})$ est reconnu par l'automate $\mathcal{A}^j = (Q^j, \Sigma, F^j, \delta^j)$ défini comme suit :

– $Q^0 = Q \cup Q_{\mathcal{R}}$, $\delta^0 = \delta \cup \delta_{\mathcal{R}} \cup \{0 \to q^{nil} \mid 0 \xrightarrow{*}_{\delta} q\}$;
– $Q^j = Q^{j-1} \cup \{q^T \mid q \in Q^{j-1}\} \cup \{q^{nil} \mid q \in Q^{j-1}\}$. Comme précédemment, nous introduisons les notations q^{T^j} définie par : $q^{T^0} = q$ et $q^{T^{j+1}} = (q^{T^j})^T$; et q^{nil^j} définie par : $q^{nil^1} = q^{nil}$ et $q^{nil^{j+1}} = \{(q^{nil^j})^T, (q^{nil^j})^{nil}, (q^{T^j})^{nil}\}$.
– $F^j = \{q^T \mid q \in F^{j-1}\}$;
– δ^j est le plus petit ensemble de règles contenant δ^{j-1} et tel que pour chaque états $q \in Q^0$ et $q_1, \ldots, q_n \in Q^{j-1}$:

1. si $t_1 \to t_2 \in \mathcal{R}$, et il existe un état $q \in Q \cup Q_{\mathcal{R}}$ tel que $t_1 \xrightarrow{*}_{\delta^{j-1}} q^{T^k}, k < j$, alors si $l + k + 1 = j$ nous avons :
   (a) $q_{t_2}^{T^l} \to q^{T^j} \in \delta^j$,
   (b) $q_{t_2}^{nil^l} \to q^{nil^j} \in \delta^j$, et
   (c) $q_{t_2}^{T^l} \to q^{nil^j} \in \delta^j$ si $t_1 = 0$,
2. si $\cdot(q_1, q_2) \to q \in \delta \cup \delta_{\mathcal{R}}$, alors :
   (a) $\cdot(q_1^{nil^k}, q_2^{T^l}) \to q^{T^j} \in \delta^j$, si $k + l = j$;
   (b) $\cdot(q_1^{nil^k}, q_2^{nil^l}) \to q^{nil^j} \in \delta^j$, si $k + l = j$
   (c) $\cdot(q_1^{T^j}, q_2) \to q^{T^j} \in \delta'$;
3. si $||(q_1, q_2) \to q \in \delta \cup \delta_{\mathcal{R}}$, alors :
   (a) $||(q_1^{nil^k}, q_2^{nil^l}) \to q^{nil^j} \in \delta^j$, si $k + l = j$;
   (b) $||(q_1^{T^k}, q_2^{T^l}) \to q^{T^j} \in \delta^j$, si $k + l = j$;
4. si $q \to q' \in \delta^{j-1}$, alors $q^T \to q'^T \in \delta^j$, et $q^{nil} \to q'^{nil} \in \delta^j$.

Il est alors facile de voir que pour tout $q \in Q \cup Q_{\mathcal{R}}$, $L'_{q^{T^j}} = Post^j_{\mathcal{R},=}(L_q)$ et que

$$L'_{q^{nil^j}} = \bigcup_{l+k=j} Post^k_{\mathcal{R},=}\big(Post^l_{\mathcal{R},=}(L_q) \cap \{u \in \mathcal{T} \mid u \sim_0 0\}\big)$$

où L_q est le langage reconnu par l'état q dans l'automate $\mathcal{A}$ et $L'_{q'}$ est le langage reconnu par l'état q' dans l'automate construit $\mathcal{A}^j$. Ceci peut se montrer facilement comme fait dans la preuve du théorème 7.6.1.

Il est clair que pour chaque règle α de la forme $q_t^T \to q^T$, $q_t^{nil} \to q^{nil}$, ou $q_t^T \to q^{nil}$ rajoutée par les règles (3) de la construction de $\mathcal{A}^*_{\mathcal{R}}$ donnée dans la section 5.3, il y a un indice j tel que δ^j contient des règles de la forme $q_t^{T^l} \to q^{T^j} \in \delta^j$, $q_t^{nil^l} \to q^{nil^j} \in \delta^j$, ou $q_t^{T^l} \to q^{T^{j-1}nil} \in \delta^j$ si $t_1 = 0$. Soit $m(\alpha)$ le minimum de tels indices j. Soit $m = \max m(\alpha)$, le maximum étant pris sur toutes les règles α de δ' qui ont été rajoutées par les règles d'inférence (3) de $\mathcal{A}^*_{\mathcal{R}}$.

Si nous comparons les hypergraphes $\mathcal{G}$ de $\mathcal{A}^0$ et $\mathcal{G}'$ celui de $\mathcal{A}^m$, nous remarquons des croissances définies par l'hypergraphe Δ défini par :
– $V_\Delta = Q^m$;
– H_Δ est l'ensemble des hyperarcs définis par les règles de $\delta^m \setminus \delta^0$;
– $\mathcal{I}_\Delta = \{q^{T^i} \mid q \in Q^0, 0 \le i \le m\} \cup \{q^{nil^i} \mid q \in Q^0, 0 \le i \le m\}$.

Considérons la partition φ suivante : $\big\{\{q^{T^i} \mid i > 0\}, \{q^{nil^i} \mid i > 0\}, \{q\} \mid q \in Q \cup Q_{\mathcal{R}}\big\}$; c-à-d. pour chaque état q dans $Q \cup Q_{\mathcal{R}}$ il y a trois classes $[q]$, $[q^T] = [q^{T^i}]$, et $[q^{nil}] = [q^{nil^i}]$ pour tout $i > 0$.

Il est facile de voir comme précédemment que l'hypergraphe calculé par l'élargissement $\nabla(\mathcal{A}^0, \mathcal{A}^m, \Delta, \varphi)$ représente exactement l'automate $\mathcal{A}^*_{\mathcal{R}}$. □

Nous obtenons le résultat suivant comme conséquence immédiate de ce théorème :

Corollaire 9.4.3 *Soit $\mathcal{R}$ un PRS sur Var et $\mathcal{L}$ un langage régulier d'arbres sur Var. si $\Rightarrow_{=,\mathcal{R}}$ est bien-fondée, alors $Post^*_{\mathcal{R},=}(\mathcal{L})$ peut être calculé par élargissement.*

Nous déduisons alors que l'élargissement peut calculer $Post^*_{\mathcal{R},=}(\mathcal{L})$ pour les systèmes PA :

Théorème 9.4.6 *Soit $\mathcal{R}$ un PA sur Var et $\mathcal{L}$ un langage régulier d'arbres sur Var, alors $Post^*_{\mathcal{R},=}(\mathcal{L})$ peut être calculé par élargissement.*

Preuve : Soit $\mathcal{R}$ un système PA. Si $\Rightarrow_{=,\mathcal{R}}$ n'est pas bien-fondée, nous calculons un langage régulier $\mathcal{L}'$ et un système PA $\mathcal{R}'$ tels que $\Rightarrow_{=,\mathcal{R}'}$ est bien-fondée et $Post^*_{\mathcal{R},=}(\mathcal{L})$ est obtenu à partir de $Post^*_{\mathcal{R}',=}(\mathcal{L}')$ par une simple substitution S. Dans ce cas, puisque $\Rightarrow_{=,\mathcal{R}'}$ est bien-fondé, le corollaire 9.4.3 implique que l'élargissement permet de calculer $Post^*_{\mathcal{R}',=}(\mathcal{L}')$. Il ne reste donc qu'à calculer la fermeture de cet ensemble par la substitution S pour avoir $Post^*_{\mathcal{R},=}(\mathcal{L})$.

Si $\Rightarrow_{=,\mathcal{R}}$ n'est pas bien-fondée, alors il existe nécessairement des variables $X_1, \ldots, X_k$ qui ne sont pas forcément tous différents tels que pour tout $1 \leq l < k$, $\mathcal{R}$ contient les règles $R_l = X_l \rightarrow X_{l+1}$ et $X_k \rightarrow X_1$ (d'après le théorème 9.3.3). Dans ce cas, nous considérons de nouvelles variables de processus X'_l, $1 \leq l < k$, et nous calculons $\mathcal{R}'$ et $\mathcal{L}'$ de la manière suivante :

- nous remplaçons la règle $X_k \rightarrow X_1$ par la règle $X_k \rightarrow X'_1$;
- nous rajoutons les règles $X'_l \rightarrow X'_{l+1}$, $1 \leq l < k$;
- nous remplaçons dans toutes les règles de $\mathcal{R}$, à part les règles R_l, ainsi que dans $\mathcal{L}$ les variables X_l par $\{X_l, X'_l\}$. Ceci exprime que dans chaque terme apparaissant dans les membres gauches ou droits des règles de $\mathcal{R}$, une ou plusieurs (ou aucune) occurrences de la variable X_l est remplacée par la variable X'_l.

Finalement, $Post^*_{\mathcal{R},=}(\mathcal{L})$ est obtenu à partir de $Post^*_{\mathcal{R}',=}(\mathcal{L}')$ en remplaçant toutes les occurrences de la variable X'_l par X_l.

□

9.4.5 Well-oriented systems

Nous montrons dans cette section que l'élargissement permet de calculer $\mathcal{R}^*$ pour tout WOS $\mathcal{R}$:

Théorème 9.4.7 *Soit $\mathcal{R}$ un well-oriented system, alors $\mathcal{R}^*$ peut être calculé par élargissement.*

Preuve : Nous montrons que l'élargissement permet de calculer les relations $\mathcal{R}_i^*$ pour chaque $1 \leq i \leq n$ et $\mathcal{R}_{i\to i+1}^*$ pour chaque $1 \leq i < n$. Nous donnons dans ce qui suit les détails du calcul de $\mathcal{R}_i^*$, l'autre construction est similaire. Comme $\mathcal{R}_i$ est bien-fondée (théorème 9.3.2), nous montrons que lors du calcul itératif des réétiquetages des relations $\mathcal{R}_i^j$, il existe une situation où l'élargissement peut s'appliquer et calculer un réétiquetage correspondant à $\mathcal{R}^*$. Les mauvaises extrapolations seront rejetées par le test 9.4. Pour simplifier la présentation, nous supposons que $\mathcal{R}_i = \mathcal{R}_i^{\uparrow}$. Le cas où $\mathcal{R}_i^{\downarrow}$ n'est pas vide se traite de la même manière.

$\mathcal{R}_i$ est donnée par le réétiquetage suivant : $\mathcal{A} = (Q, \mathcal{S}, \mathcal{S}, F, \delta)$ où $Q = \{q, q'\} \cup \{q_c \mid c \in \mathcal{S}_i\} \cup \{q_{b'}^a \mid a, b' \in \mathcal{S}_i\}$, $F = \{q'\}$, et δ est l'ensemble des règles de transition suivantes, où à chaque règles de la forme $f/g(q_1, q_2) \to q_3$ correspond une règle symétrique de la forme $f/g(q_2, q_1) \to q_3$ qui n'est pas représentée :

(γ_1) $a/a \to q$ et $a/a(q, q) \to q$, pour tout $a \in \mathcal{S}$,

(γ_2) $c/c \to q_c$, $c/c(q, q) \to q_c$, et $q_c \to q$ pour tout $c \in \mathcal{S}_i$,

et tel que si $\mathcal{R}_i^{\uparrow}$ contient la règle $b(a(x_1, x_2), c(x_3, x_4)) \to a(b'(x_1, x_2), c(x_3, x_4))$, où $a, b', c \in \mathcal{S}_i$, et $b \in \mathcal{S}_{i-1}$, alors la relation de transition δ contient :

(γ_3) $a/b'(q, q) \to q_{b'}^a$,

(γ_4) $a/b' \to q_{b'}^a$,

(γ_5) $b/a(q_{b'}^a, q_c) \to q'$,

(γ_6) $a/a(q', q) \to q'$, pour tout $a \in \mathcal{S}$,

L'état q reconnaît les termes de la forme t/t où aucune réécriture ne s'est produite. Les états q_a reconnaissent les termes de la forme t/t où $racine(t) = a$ avec $a \in \mathcal{S}_i$. L'état q' reconnaît les termes de la forme t/t' tels que $t' \in \mathcal{R}_i(t)$.

Si nous composons le réétiquetage ci-dessus avec lui-même comme décrit dans la section 2.1.2.3, nous obtenons le réétiquetage $\mathcal{A}^2$ suivant : $\mathcal{A}^2 = (Q', \mathcal{S}, \mathcal{S}, F', \delta')$ où $Q' = Q \times Q$, $F' = F \times F$, et δ' est l'ensemble des règles de transition obtenues en appliquant la procédure décrite dans la section 2.1.2.3. Si $\mathcal{R}_i^{\uparrow}$ contient les règles $b(a(x_1, x_2), c(x_3, x_4)) \to a(b'(x_1, x_2), c(x_3, x_4))$ et $d(a(x_1, x_2), e(x_3, x_4)) \to a(d'(x_1, x_2), e(x_3, x_4))$, où $a, b', c, d', e \in \mathcal{S}_i$, et $b, d \in \mathcal{S}_{i-1}$, alors δ' contient :

(β_1) $f/f \to [q_0, q_1]$ et $f/f\big([q, q], [q, q]\big) \to [q_0, q_1]$ pour tout $f \in \mathcal{S}$ avec $q_0, q_1 \in \{q, q_f\}$;

(β_2) $a/b'\big([q, q], [q, q]\big) \to [q_{b'}^a, q]$, $a/b' \to [q_{b'}^a, q]$;

(β_3) $a/b'\big([q, q], [q, q]\big) \to [q, q_{b'}^a]$, $a/b' \to [q, q_{b'}^a]$;

(β_4) $b/a\big([q_{b'}^a, q], [q_c, q]\big) \to [q', q]$,

(β_5) $b/a\big([q_{b'}^a, q], [q_c, q]\big) \to [q', q_a]$,

(β_6) $b/a\big([q, q_{b'}^a], [q, q_c]\big) \to [q, q']$,

(β_7) $b/a\big([q, q_{b'}^a], [q', q_c]\big) \to [q', q']$,

(β_8) $b/d'\big([q_{b'}^a, q], [q_c, q]\big) \to [q', q_{d'}^a]$,

(β_9) $b/a\big([q', q_{b'}^a], [q, q_c]\big) \to [q', q']$,

(β_{10}) $f/f\big([q', q], [q, q']\big) \to [q', q']$, $f/f\big([q, q], [q', q']\big) \to [q', q']$, $f/f\big([q, q], [q', q]\big) \to [q', q]$, $f/f\big([q, q], [q, q']\big) \to [q, q']$, pour tout $f \in \mathcal{S}$,

(β_{11}) $[q_1, q_a] \to [q_1, q]$ et $[q_a, q_1] \to [q, q_1]$, avec $q_1 \in \{q, q_f, q'\}$.

En comparant les hypergraphes $\mathcal{G}$ de $\mathcal{A}$ à $\mathcal{G}'$ celui de $\mathcal{A}^2$ nous détectons l'hypergraphe Δ comme croissance, où Δ est l'hypergraphe défini par les règles (β_4), (β_5), (β_6), (β_8), (β_{11}), et une partie des règles (β_{10}). Si nous prenons la partition φ suivante : $[q_a] = \{[q', q_a], [q, q_a], [q_a, q]\}$ pour chaque $a \in \mathcal{S}_i$, $[q^a_{b'}] = \{[q', q^a_{b'}], [q^a_{b'}, q], [q, q^a_{b'}]\}$, et $[q] = \{[q', q], [q, q'], [q', q']\}$, nous vérifions que

$$(\mathcal{G}, F) = \left(\mathcal{G}' \setminus_\varphi \Delta, F\right)$$

et qu'en fusionnant les états de Δ selon la partition φ nous obtenons un réétiquetage équivalent à celui de $\mathcal{R}^*_i$ donné à la section 3.2.3. Intuitivement, en reprenant les règles (α_1)–(α_8) de la section 3.2.3, les états $s_{[q]}$, $s_{[q_a]}$, et $s_{[q^a_{b'}]}$ correspondent respectivement aux états q, q_a, et $q^a_{b'}$ de la construction de la section 3.2.3. Donc après fusion des états, les règles (β_1) vont donner les règles (α_1) et (α_2) ; les règles (β_2) et (β_3) vont donner les règles (α_4) et (α_5) ; les règles (β_5) vont donner les règles (α_6) ; les règles (β_8) vont donner les règles (α_7) et (α_8) ; les règles (β_{10}) vont donner les règles (α_1) ; et les règles (β_{11}) vont donner les règles (α_3).

□

Remarque 9.4.1 *Les résultats de complétude donnés dans cette section sont tous basés sur un test qui permet de déterminer si l'ensemble calculé par notre technique d'élargissement est égal à l'ensemble des accessibles. Observons qu'avoir un tel test permet toujours de calculer cet ensemble s'il est régulier : il suffit d'énumérer les automates finis réguliers jusqu'à ce que le test réussisse. Cette méthode n'est cependant pas applicable en pratique. De plus, les résultats de cette section montrent que nos techniques d'élargissement sont assez puissantes et générales pour simuler plusieurs algorithmes directs de calcul de clôtures transitives spécifiques à des classes de relations et de langages particulières. Ceci montre que tous ces algorithmes appliquent de manière implicite ce principe d'élargissement. D'ailleurs, nous pensons que ce principe est aussi présent dans beaucoup d'autres algorithmes tels que les algorithmes de calcul des accessibles des automates à pile [BEM97, EHRS00], ou des systèmes à file [BH97, AAB99].*

9.5 Calcul d'un graphe d'accessibilité symbolique par élargissement

Etant donné un système défini par un ensemble de configurations initiales $\mathcal{L}_0$ et n relations $\mathcal{R}_1, \ldots, \mathcal{R}_n$ décrivant chacune une action du système ; l'élargissement permet, en cas de terminaison, de calculer un graphe d'accessibilité symbolique qui simule le système et en constitue donc une abstraction finie. Les nœuds de ce graphe correspondent aux ensembles des accessibles calculés aux différentes étapes, et les arcs sont étiquetés par le nom des relations appliquées. L'idée comme décrite au chapitre 2 consiste à décomposer $\mathcal{R}$ en $m+1$ relations $\mathcal{R} = \mathcal{R}' \cup \mathcal{R}'_1 \cup \cdots \cup \mathcal{R}'_m$, où $\mathcal{R}'$ est elle-même une union de relations $\mathcal{R}_{i_1}, \ldots, \mathcal{R}_{i_k}$; telles que l'on sache calculer les clôtures réflexives-transitives par les $\mathcal{R}'_i$ pour tout $1 \leq i \leq m$ (soit par des algorithmes spécifiques, soit

par élargissement). Il s'agit ensuite d'utiliser ces résultats partiels pour essayer de calculer $\mathcal{R}^*(\mathcal{L}_0)$. La procédure consiste à partir de $\mathcal{L}_0$ et à appliquer à chaque fois une des relations $\mathcal{R}_{i_1}, \ldots, \mathcal{R}_{i_k}$, ou ${\mathcal{R}'_1}^*, \ldots, {\mathcal{R}'_m}^*$. Au cours du calcul, nous pouvons également utiliser l'élargissement pour accélérer le calcul de l'ensemble des accessibles de la manière suivante : nous comparons chaque nouvel ensemble calculé à ses ancêtres dans le graphe pour essayer de détecter des croissances et extrapoler. Si nous trouvons dans le graphe un chemin de la forme $L_1 \xrightarrow{r_1} L_2 \cdots \xrightarrow{r_h} L_h$ tel que nous détectons une croissance entre L_1 et L_h, nous extrapolons et vérifions que l'ensemble L'_h ainsi calculé est *exactement* égal à $(r_h \circ \cdots \circ r_1)^*(L_1)$. Si c'est le cas, nous rajoutons au graphe un nœud étiqueté par L'_h, et relié au nœud correspondant à L_1 par un arc étiqueté par $(r_h \circ \cdots \circ r_1)^*$.

Observons qu'avec cette procédure, chaque relation r_i peut elle-même être la clôture transitive de compositions d'autres relations. Le problème qui se pose alors, c'est comment vérifier que $L'_h = (r_h \circ \cdots \circ r_1)^*(L_1)$. Pour ce faire, si $r_h \circ \cdots \circ r_1$ est bien-fondée, nous utilisons le test (9.3). Seulement, le problème qui se pose est que pour appliquer ce test, nous devons être capables de calculer $r_h \circ \cdots \circ r_1(L'_h)$, ce qui n'est pas toujours possible si les r_i n'ont pas été caractérisées. En effet, il se pourrait que $L_h = r_h \circ \cdots \circ r_1(L_1)$ ait été calculé en utilisant l'élargissement à plusieurs reprises ; et le fait que l'élargissement ait permis de calculer $r_h \circ \cdots \circ r_1(L_1)$ ne garantie pas qu'il saura calculer $r_h \circ \cdots \circ r_1(L'_h)$. Nous proposons une solution à ce problème dans le cas où pour tout i, $r_i = {r'_i}^*$ pour une certaine relation r'_i que l'on sait caractériser. Dans ce cas, il y a un autre problème qui se présente, puisque même si nous savons caractériser les r_i, tester si $L'_h = r_h \circ \cdots \circ r_1(L'_h) \cup L_1$ ne nous permet pas de conclure quant à l'exactitude de notre extrapolation. En effet, la relation $r_h \circ \cdots \circ r_1 = {r'}_h^* \circ \cdots \circ {r'}_1^*$ n'est jamais bien-fondée puisque $Id_{\Sigma^*} \subseteq {r'}_h^* \circ \cdots \circ {r'}_1^*$ et Id_{Σ^*} n'est pas bien-fondée (pour tout $w \in \Sigma^*, (w, w) \in Id_{\Sigma^*}$). Pour contourner ce problème, nous remarquons que $({r'}_h^* \circ \cdots \circ {r'}_1^*)^* = ({r'}_h + \cdots + {r'}_1)^*$, où la relation ${r'}_h + \cdots + {r'}_1$ est caractérisable puisque les r'_i le sont. Donc, vérifier que $L'_h = (r_h \circ \cdots \circ r_1)^*(L_1)$ revient à vérifier que $L'_h = ({r'}_h + \cdots + {r'}_1)^*(L_1)$ en appliquant le test (9.3) avec la relation ${r'}_h + \cdots + {r'}_1$ si elle est bien-fondée (elle a des chances de l'être, contrairement à ${r'}_h^* \circ \cdots \circ {r'}_1^*$)

Si nous n'avons pas de moyens pour vérifier l'exactitude de notre extrapolation (si la séquence de relations considérée n'est pas bien-fondée, ou si elle n'est pas caractérisable), nous pouvons abandonner l'idée de calculer exactement l'ensemble des accessibles et se contenter d'en calculer une sur-approximation. Dans ce cas, nous extrapolons à chaque fois que ceci est possible.

9.6 Exemples

Nous avons appliqué notre méthode d'élargissement pour analyser plusieurs protocoles d'exclusion mutuelle définis sur des systèmes paramétrés linéaires ou arborescents. Pour le cas linéaire, nous avons considéré l'algorithme du "Bakery" et les protocoles de Dijkstra, et de Szymanski. Nous montrons dans la première partie de cette section comment traiter l'algorithme du "Bakery". L'analyse des autres protocoles est décrite en annexes. Pour le cas arborescent, notre méthode permet de traiter les algorithmes PERCOLATE et "parité" décrits dans la section 3.2 puisqu'ils sont modélisés par des

WOS, et l'élargissement permet de calculer la clôture réflexive-transitive de chaque WOS (théorème 9.4.7). Nous considérons également un protocole d'exclusion mutuelle défini sur une architecture arborescente. Nous décrivons ce protocole dans la deuxième partie de cette section.

9.6.1 Le "Bakery algorithm"

9.6.1.1 Description et modélisation de l'algorithme

Processus i :
$\perp$: init
W : $ticket(i)$:=1+max$_j$ $ticket(j)$
pour tout $j \neq i$ faire
    SI ($ticket(i) < ticket(j)$ ou $ticket(j) = 0$) faire
C :   entrer dans C
    fSI
fpour
$ticket(i)$:=0
$\perp$: init

FIG. 9.14 – Le "Bakery algorithm"

Le fonctionnement de cet algorithme est similaire à celui d'une boulangerie "bakery" : quand un nouveau client arrive, il prend un ticket (avec un numéro $ticket(i)$). Il est servi (il entre dans la section critique) quand tous les clients dont le numéro de ticket est plus petit que $ticket(i)$ (qui étaient arrivés avant lui) le sont déjà. C'est-à-dire qu'à chaque fois, c'est le processus ayant le plus petit indice (qui s'est présenté le premier) qui entre dans la section critique. Nous représentons les processus par une séquence dans laquelle les processus en attente sont ordonnés de façon que le processus le plus à gauche dans la séquence soit celui qui porte le plus petit indice.

Cet algorithme peut donc être modélisé selon la figure 9.15, où un processus est à l'état $\perp$ s'il est inactif. Il est à l'état W s'il est en attente (Wait) et est à l'état C s'il est dans la section critique.

$\mathcal{R}_1$: $(\perp, W)copy(\perp^*)$,
$\mathcal{R}_2$: $copy(\Sigma^*(C + W))(\perp, W)copy(\perp^*)$,
$\mathcal{R}_3$: $copy(\perp^*)(W, C)copy(\Sigma^*)$,
$\mathcal{R}_4$: $copy(\Sigma^*)(C, \perp)copy(\Sigma^*)$.

FIG. 9.15 – Relations représentant le "Bakery algorithm"

Notons que la première ligne de l'algorithme est représentée par deux relations :

1. la première relation $\mathcal{R}_1$ traite le cas où i est le premier processus qui s'est manifesté pour entrer dans la section critique. C'est donc le processus le plus à gauche qui passe de l'état inactif $\perp$ à l'état d'attente W.
2. la seconde relation $\mathcal{R}_2$ exprime le fait que s'il existe des processus en attente ou dans la section critique, alors le prochain processus qui se manifeste aura le plus grand indice (*ticket*) dans la séquence.

La relation $\mathcal{R}_3$ exprime que si la section critique est vide, alors le processus qui y entrerait serait celui qui aurait attendu le plus, c'est-à-dire, celui en attente ayant le plus petit indice dans la séquence. La relation $\mathcal{R}_4$ libère la section critique.

L'ensemble des configurations initiales est représenté par $\perp^*$ et celui des mauvaises configurations par $\Sigma^* C \Sigma^* C \Sigma^*$, c-à-d. les configurations où au moins deux processus sont dans la section critique en même temps.

9.6.1.2 Analyse d'accessibilité du "Bakery algorithm"

L'élargissement permet de calculer les clôtures transitives des différentes relations de ce programme de manière exacte. Nous procédons comme dans la section 9.2.3 où nous avons calculé $\mathcal{R}_4^+$ pour calculer les autres relations. En fait, ces relations peuvent être calculées en suivant la preuve du théorème 9.4.1 puisqu'elles sont toutes des context-relations. Nous obtenons :

- $\mathcal{R}_1^+ = \mathcal{R}_1 = (\perp, W) copy(\perp^*)$,
- $\mathcal{R}_2^+ = copy(\Sigma^*(C + W))(\perp, W)(\perp, W)^* copy(\perp^*)$,
- $\mathcal{R}_3^+ = \mathcal{R}_3 = copy(\perp^*)(W, C) copy(\Sigma^*)$,
- $\mathcal{R}_4^+ = copy(\Sigma^*)(C, \perp)\big(copy(\Sigma) + (C, \perp)\big)^*$.

Essayons maintenant d'appliquer l'algorithme d'exploration du graphe d'accessibilité symbolique en utilisant ces relations accélérées, mais sans appliquer l'élargissement. L'analyse d'accessibilité figure au tableau (9.1), où les deux premières colonnes représentent l'ensemble de configurations auquels nous appliquons la relation donnée dans la troisième colonne. L'ensemble obtenu est représenté dans la colonne 4. La colonne 5 indique quand un ensemble calculé est inclus dans un autre déjà calculé.

Nous remarquons que le calcul ne s'arrêtera pas. En effet, la séquence $\mathcal{R}_4 \circ \mathcal{R}_3^* \circ \mathcal{R}_2^*$ ajoute toujours un $\perp$ à gauche du langage considéré puisqu'en partant de $L_4 = \perp \mathrm{W}^* \perp^*$, et en appliquant la séquence $\mathcal{R}_4 \circ \mathcal{R}_3^* \circ \mathcal{R}_2^*$ deux fois de suite, nous trouvons $L_7 = \perp\perp \mathrm{W}^* \perp^*$ et $L_{10} = \perp\perp\perp \mathrm{W}^* \perp^*$, etc. Nous voyons bien que l'exécution de cette séquence est infinie. Nous devons alors appliquer l'élargissement entre L_4 et L_7, ce qui donne $L_7' = \perp^* \perp W^* \perp^*$ [4]. De plus, le test 9.3 est valide puisqu'il est facile de vérifier que

$$\mathcal{R}_4 \circ \mathcal{R}_3^* \circ \mathcal{R}_2^*(L_7') \cup L_4 = L_7'$$

et donc comme $\mathcal{R}_4 \circ \mathcal{R}_3^* \circ \mathcal{R}_2^*$ est bien-fondée, nous sommes sûrs que l'élargissement appliqué est exact. En remplaçant L_7 par L_7', nous voyons que l'analyse de l'accessibilité termine (tableau 9.2).

[4] Il nous faut bien sûr considérer les automates correspondant à ces langages pour pouvoir appliquer l'élargissement. Mais nous décrivons ici cette opération sur les langages de manière informelle.

L_0	$\bot^*$	$\mathcal{R}_1^*$	L_1
L_1	$\mathrm{W}\bot^*$	$\mathcal{R}_2^*$	L_2
L_2	$\mathrm{WW}^*\bot^*$	$\mathcal{R}_3^*$	L_3
L_3	$\mathrm{CW}^*\bot^*$	$\mathcal{R}_4$	L_4
L_4	$\bot\mathrm{W}^*\bot^*$	$\mathcal{R}_2^*$	L_5
L_5	$\bot\mathrm{WW}^*\bot^*$	$\mathcal{R}_3^*$	L_6
L_6	$\bot\mathrm{CW}^*\bot^*$	$\mathcal{R}_4$	L_7
L_7	$\bot\bot\mathrm{W}^*\bot^*$	$\mathcal{R}_2^*$	L_8
L_8	$\bot\bot\mathrm{WW}^*\bot^*$	$\mathcal{R}_3^*$	L_9
L_9	$\bot\bot\mathrm{CW}^*\bot^*$	$\mathcal{R}_4$	L_{10}
L_{10}	$\bot\bot\bot\mathrm{W}^*\bot^*$	$\mathcal{R}_2^*$	L_{11}
L_{11}	$\cdots$	$\cdots$	L_∞

TAB. 9.1 – Analyse de l'accessibilité du "Bakery algorithm" sans élargissement

A présent, nous pouvons valider le "Bakery algorithm" en vérifiant que les mauvaises configurations représentées par $\Sigma^* C \Sigma^* C \Sigma^*$ ne sont pas accessibles. Nous en déduisons qur le "Bakery algorithm" satisfait bien la propriété d'exclusion mutuelle.

L_0	$\bot^*$	$\mathcal{R}_1^*$	L_1	
L_1	$\mathrm{W}\bot^*$	$\mathcal{R}_2^*$	L_2	
L_2	$\mathrm{WW}^*\bot^*$	$\mathcal{R}_3^*$	L_3	
L_3	$\mathrm{CW}^*\bot^*$	$\mathcal{R}_4 + \mathcal{R}_2^* \circ \mathcal{R}_3^* \circ \mathcal{R}_4^*$	L_4	
L_4	$\bot^*\mathrm{W}^*\bot^*$	$\mathcal{R}_3^*$	L_5	
L_5	$\bot^*\mathrm{CW}^*\bot^*$	$\mathcal{R}_4$	L_6	
L_6	$\bot^*\bot\mathrm{W}^*\bot^*$			$\subset L_4$

TAB. 9.2 – Analyse d'accessibilité du "Bakery algorithm" avec élargissement

9.6.2 L'arbre arbitre : un protocole d'exclusion mutuelle sur des architectures arborescentes [ABH+97]

9.6.2.1 Description et modélisation

Le système consiste en un nombre arbitraire de processus qui se partagent une ressource commune. Les processus sont disposés sous forme d'un arbre binaire. Seuls les processus aux feuilles peuvent avoir la ressource. Le rôle des autres processus étant de propager les demandes de leurs fils pour qu'elles atteignent la racine. Les processus internes arbitrent entre leur deux fils : si les deux demandent la ressource, le père choisit l'un des deux de manière non déterministe. Les gagnants sont ensuite arbitrés au niveau suivant, et ainsi de suite. Les demandes sont propagées le long de l'arbre jusqu'à ce que la racine est atteinte. Cette dernière alloue la ressource à (au plus) une

feuille. Cette permission d'accéder à la ressource se propage le long de l'arbre jusqu'à une des feuilles (celle qui va accéder à la ressource critique). Quand la ressource est libérée par le processus qui la détient, le système est réinitialisé.

Ce programme peut être modélisé comme suit : Chaque processus peut être dans l'un des états suivants :

- $\bot$ s'il ne demande pas l'accès à la ressource critique,
- D s'il détient une demande d'accès à la ressource critique,
- D' s'il détient une demande d'accès à la ressource critique, et qu'il a été choisi par son père,
- R s'il reçoit l'accord pour accéder à la ressource (s'il est une feuille), ou pour permettre à l'un de ses fils d'accéder à la ressource.

Initialement, tous les processus sont dans l'état $\bot$, à l'exception de quelques feuilles qui demandent l'accés à la ressource, et qui sont donc dans l'état D. L'ensemble des configurations initiales $\mathcal{L}_0$ est donc le langage régulier d'arbres reconnu par l'automate $\mathcal{A} = (Q, \Sigma, F, \delta)$ où $Q = F = \{q\}$, $\Sigma_0 = \Sigma_2 = \{\bot, D, D', R\}$, et δ contient les règles $\bot \to q$, $D \to q$, et $\bot(q, q) \to q$.

Les actions du système sont représentées par les relations de réétiquetage suivantes.

- $\mathcal{R}_1$ est l'ensemble des paires de termes $(t, t') \in T_\Sigma$ tels qu'il existe un contexte C, et trois termes t_1, t_2, et t_3 tels que :
  - soit $t = C\Big[\bot\big(D(t_1, t_2), t_3\big)\Big]$ et $t' = C\Big[D\big(D'(t_1, t_2), t_3\big)\Big]$,
  - soit $t = C\Big[\bot\big(t_3, D(t_1, t_2)\big)\Big]$ et $t' = C\Big[D\big(t_3, D'(t_1, t_2)\big)\Big]$,
  - soit $t = C\big[\bot(D, t_3)\big]$ et $t' = C\big[D(D', t_3)\big]$,
  - soit $t = C\big[\bot(t_3, D)\big]$ et $t' = C\big[D(t_3, D')\big]$.

  Cette relation modélise la propagation ascendante de la demande d'accés à la ressource : Si un processus est dans l'état D (c-à-d., s'il demande la ressource), alors son père va demander la ressource pour lui : il passe à l'état D, et le processus lui-même passe à l'état D' pour mémoriser qu'il a fait une demande, et qu'il a été choisi par son père pour recevoir la ressource. Si les deux fils d'un même processus font une demande simultanée, le processus père doit choisir de manière non déterministe l'un des deux et demander la ressource pour lui. C'est ce processus qui passe à l'état D'.

  Observons que cette relation n'est pas un WOS puisque pour que ça soit le cas, il faut que $D \in \mathcal{S}_i$, $\bot \in \mathcal{S}_{i-1}$, et que la racine de t_3 soit dans $\mathcal{S}_i$. Or ceci n'est pas possible puisque la racine de t_3 peut être égale à $\bot$.

- $\mathcal{R}_2$ est l'ensemble des paires de termes $(t, t') \in T_\Sigma$ tels qu'il existe deux sous termes t_1 et t_2 tels que $t = D(t_1, t_2)$ et $t' = R(t_1, t_2)$. Cette relation exprime que quand la demande atteint la racine, cette dernière donne la permission d'accès à la ressource.

- $\mathcal{R}_3$ est l'ensemble des paires de termes $(t, t') \in T_\Sigma$ tels qu'il existe un contexte C et trois sous termes t_1, t_2, et t_3 tels que :
  - $t = C\Big[R\big(D'(t_1, t_2), t_3\big)\Big]$ et $t' = C\Big[R\big(R(t_1, t_2), t_3\big)\Big]$, ou
  - $t = C\Big[R\big(t_1, D'(t_2, t_3)\big)\Big]$ et $t' = C\Big[R\big(t_1, R(t_2, t_3)\big)\Big]$, ou

– $t = C\big[R(D', t_1)\big]$ et $t' = C\big[R(R, t_1)\big]$, ou
– $t = C\big[R(t_1, D')\big]$ et $t' = C\big[R(t_1, R)\big]$.
Cette relation exprime que la réponse concernant l'allocation de la ressouce se propage de la racine jusqu'à l'une des feuilles.

9.6.2.2 Analyse d'accessibilité de l'algorithme

Pour montrer que cet algorithme satisfait la propriété d'exclusion mutuelle, il faut montrer que

$$\mathcal{R}_3^* \circ \mathcal{R}_2^* \circ \mathcal{R}_1^*(\mathcal{L}_0) \subseteq \mathcal{L}_{mutex}$$

où $\mathcal{L}_{mutex}$ est l'ensemble régulier des termes dont au plus une seule feuille est à l'état R. $\mathcal{L}_{mutex}$ est représenté par l'automate $\mathcal{A} = (Q_{mutex}, \Sigma, F_{mutex}, \delta_{mutex})$ où $Q_{mutex} = \{q_1, q_2, q_3\}$, $F_{mutex} = \{q_2, q_3\}$, $\Sigma_0 = \Sigma_2 = \{\bot, D, D', R\}$, et δ_{mutex} contient les règles $R \to q_1$; $g \to q_3$, $f(q_3, q_3) \to q_3$, $f(q_1, q_3) \to q_2$, et $f(q_2, q_3) \to q_2$, pour tout $f \in \Sigma$ et $g \in \Sigma \setminus \{R\}$. L'état q_1 annote la feuille R, l'état q_3 reconnaît les termes dont aucune feuille n'est étiquetée par R, et l'état q_2 reconnaît les termes ayant une seule feuille étiquetée par R.

Il est clair que $\mathcal{R}_1$, $\mathcal{R}_2$ et $\mathcal{R}_3$ sont bien-fondées. Nous montrons dans ce qui suit que l'élargissement permet de construire les clôtures réflexives-transitives $\mathcal{R}_1^*$, $\mathcal{R}_2^*$ et $\mathcal{R}_3^*$. Dans le réétiquetages ci-dessous, à chaque règle de la forme $f/g(q_1, q_2) \to q_3$ correspond une règle $f/g(q_2, q_1) \to q_3$ qui n'est pas représentée pour simplifier la présentation.

Calcul de $\mathcal{R}_1^*$.
$\mathcal{R}_1$ est donné par le réétiquetage $\mathcal{T}_1 = (Q_1, \Sigma \times \Sigma, F_1, \delta_1)$, où $Q_1 = \{q, q_{D'}, q'\}$, $F_1 = \{q'\}$, et δ_1 est l'ensemble des règles suivantes, où $f \in \Sigma$:
– $f/f \to q$, $f/f(q, q) \to q$;
– $D/D' \to q_{D'}$, $D/D'(q, q) \to q_{D'}$;
– $\bot/D(q_{D'}, q) \to q'$;
– $f/f(q', q) \to q'$.

Le noeud étiqueté par D/D' où D est réécrit en D' est annoté par $q_{D'}$. Le père de ce noeud doit être annoté par q' puisqu'il doit être étiqueté par $\bot/D$.

En composant $\mathcal{T}_1$ avec lui-même, nous obtenons le réétiquetage $\mathcal{T}_1^2$ qui reconnaît $\mathcal{R}_1^2$ suivant : $\mathcal{T}_1^2 = (Q_1 \times Q_1, \Sigma \times \Sigma, F_1 \times F_1, \delta_1')$, où δ_1' est l'ensemble des règles suivantes, où $f \in \Sigma$:

1. $f/f \to [q, q]$, $f/f\big([q, q], [q, q]\big) \to [q, q]$, $f/f\big([q, q'], [q, q]\big) \to [q, q']$, $f/f\big([q', q], [q, q]\big) \to [q', q]$;
2. $D/D' \to [q_{D'}, q]$, $D/D' \to [q, q_{D'}]$;
3. $D/D'\big([q, q], [q, q]\big) \to [q_{D'}, q]$, $D/D'\big([q, q], [q, q]\big) \to [q, q_{D'}]$;
4. $\bot/D\big([q_{D'}, q], [q, q]\big) \to [q', q]$, $\bot/D\big([q, q_{D'}], [q, q]\big) \to [q, q']$;
5. $\bot/D'\big([q_{D'}, q], [q, q]\big) \to [q', q_{D'}]$;
6. $D/D'\big([q', q], [q, q]\big) \to [q', q_{D'}]$, $D/D'\big([q, q], [q, q']\big) \to [q_{D'}, q']$;
7. $\bot/D\big([q_{D'}, q'], [q, q]\big) \to [q', q']$, $\bot/D\big([q, q], [q', q_{D'}]\big) \to [q', q']$;
8. $f/f\big([q', q], [q, q']\big) \to [q', q']$, $f/f\big([q, q], [q', q']\big) \to [q', q']$.

En comparant l'hypergraphe de ce réétiquetage à celui de $\mathcal{T}_1$, nous pouvons considérer la croissance Δ définie par les règles 4, 5, et 6. Si nous considérons $\mathcal{I}_\Delta = Q_1 \times Q_1$, et la partition $[q_{D'}] = \{[q, q_{D'}], [q_{D'}, q], [q', q_{D'}], [q_{D'}, q']\}$ et $[q] = \{[q, q], [q', q], [q, q'], [q', q']\}$; nous obtenons que l'hypergraphe quotient est bisimilaire à $\mathcal{T}_1$, l'élargissement nous donnne alors le réétiquetage $\mathcal{T} = (Q, \Sigma \times \Sigma, F, \delta)$, où $Q = \{[q_{D'}], [q]\}$, $F = \{[q]\}$, et δ est l'ensemble des règles suivantes, où $f \in \Sigma$:

- $f/f \to [q]$, $f/f([q], [q]) \to [q]$;
- $D/D' \to [q_{D'}]$, $D/D'([q], [q]) \to [q_{D'}]$;
- $\bot/D([q_{D'}], [q]) \to [q]$;
- $\bot/D'([q_{D'}], [q]) \to [q_{D'}]$.

Ce réétiquetage reconnaît *exactement* $\mathcal{R}_1^*$ puisque le test 9.4 réussit et que $\mathcal{R}_1$ est bien-fondée.

Calcul de $\mathcal{R}_2^*$.
Il est facile de voir que $\mathcal{R}_2^* = \mathcal{R}_2$ est reconnu par le réétiquetage $\mathcal{T}_2 = (Q_2, \Sigma \times \Sigma, F_2, \delta_2)$, où $Q_2 = \{p, p'\}$, $F_1 = \{p'\}$, et δ_2 est l'ensemble des règles suivantes, où $f \in \Sigma$:

- $f/f \to p$, $f/f(p, p) \to p$;
- $D/R \to p'$, $D/R(p, p) \to p'$.

Calcul de $\mathcal{R}_3^*$.
L'élargissement permet aussi de calculer de manière précise $\mathcal{R}_3^*$. $\mathcal{R}_3$ est donné par le réétiquetage $\mathcal{T}_3 = (Q_3, \Sigma \times \Sigma, F_3, \delta_3)$, où $Q_3 = \{s, s_R, s'\}$, $F_3 = \{s'\}$, et δ_3 est l'ensemble des règles suivantes, où $f \in \Sigma$:

- $f/f \to s$, $f/f(s, s) \to s$;
- $D'/R \to s_R$, $D'/R(s, s) \to s_R$;
- $R/R(s_R, s) \to s'$;
- $f/f(s', s) \to s'$.

Le noeud étiqueté par D'/R où D' est réécrit en R est annoté par s_R. Le père de ce noeud doit être annoté par s' puisqu'il doit être étiqueté par R/R.

En composant $\mathcal{T}_3$ avec lui-même, nous obtenons le réétiquetage $\mathcal{T}_3^2$ qui reconnaît $\mathcal{R}_3^2$ suivant : $\mathcal{T}_3^2 = (Q_3 \times Q_3, \Sigma \times \Sigma, F_3 \times F_3, \delta_3')$, où δ_3' est l'ensemble des règles suivantes, où $f \in \Sigma$:

1. $f/f \to [s, s]$, $f/f\big([s, s], [s, s]\big) \to [s, s]$, $f/f\big([s, s'], [s, s]\big) \to [s, s']$, $f/f\big([s', s], [s, s]\big) \to [s', s]$;
2. $D'/R \to [s_R, s]$, $D'/R \to [s, s_R]$;
3. $D'/R\big([s, s], [s, s]\big) \to [s_R, s]$;
4. $D'/R\big([s, s], [s, s]\big) \to [s, s_R]$;
5. $D'/R\big([s, s], [s, s_R]\big) \to [s_R, s']$;
6. $R/R\big([s_R, s], [s, s]\big) \to [s', s]$;
7. $R/R\big([s, s_R], [s, s]\big) \to [s, s']$;
8. $R/R\big([s_R, s'], [s, s]\big) \to [s', s']$;
9. $f/f\big([s', s], [s, s']\big) \to [s', s']$, $f/f\big([s, s], [s', s']\big) \to [s', s']$.

En comparant l'hypergraphe de ce réétiquetage à celui de $\mathcal{T}_3$, nous pouvons considérer la croissance Δ définie par les règles 4, 5, 7, et 9. Si nous considérons $\mathcal{I}_\Delta = Q_3 \times Q_3$, et la partition $[s_R] = \{[s, s_R], [s_R, s], [s_R, s']\}$ et $[s] = \{[s, s], [s', s], [s, s'], [s', s']\}$; nous obtenons que l'hypergraphe quotient est bisimilaire à $\mathcal{T}_3$, l'élargissement nous donnne alors le réétiquetage $\mathcal{T}' = (Q, \Sigma \times \Sigma, F, \delta)$, où $Q = \{[s_R], [s]\}$, $F = \{[s]\}$, et δ est l'ensemble des règles suivantes, où $f \in \Sigma$:

- $f/f \to [s]$, $f/f([s], [s]) \to [s]$;
- $D'/R \to [s_R]$, $D'/R([s], [s]) \to [s_R]$, $D'/R([s_R], [s]) \to [s_R]$;
- $R/R([s_R], [s]) \to [s]$.

Ce réétiquetage reconnaît *exactement* $\mathcal{R}_3^*$ puisque le test 9.4 réussit et que $\mathcal{R}_3$ est bien-fondée.

Calcul des accessibles.

En appliquant $\mathcal{R}_3^* \circ \mathcal{R}_2^* \circ \mathcal{R}_1^*$ à $\mathcal{L}_0$ en composant les automates comme décrit dans la section 2.1.2.3, nous obtenons un automate $\mathcal{A}'$ qui reconnaît l'ensemble des accessibles. Cet automate est défini par : $\mathcal{A}' = (Q', \Sigma, F', \delta')$ où $Q' = \{q_1, q_2, q_3, q_4\}$, $F' = \{q_3\}$, et δ' contient les règles suivantes, où $f \in \Sigma \setminus \{R\}$:

- $f \to q_2$, $f(q_2, q_2) \to q_2$;
- $D' \to q_1$, $D'(q_2, q_1) \to q_1$;
- $D(q_2, q_1) \to q_2$;
- $R(q_2, q_1) \to q_4$, $R(q_2, q_4) \to q_4$;
- $R(q_2, q_4) \to q_3$;
- $R \to q_4$.

Il est facile de vérifier que le langage reconnu par cet automate est inclus dans $\mathcal{L}_{mutex}$, ce qui veut dire que tous les termes accessibles ont au plus une feuille dans l'état R. Donc, le protocole satisfait la propriété d'exclusion mutuelle.

9.7 Comparaison avec d'autres travaux

9.7.1 Travaux basés sur le principe d'élargissement

9.7.1.1 L'élargissement dans le cadre de l'interprétation abstraite

L'idée de l'élargissement a été proposée dans [CC77] pour sur-approximer des plus petits points fixes dans un treillis complet. Elle a été principalement utilisée pour les domaines manipulant des entiers ou des réels [CH78]. Les techniques considérées ne se préoccupent pas de l'exactitude du point fixe calculé, et leur convergence est garantie. Contrairement à la méthode que nous proposons, où nous appliquons l'idée de l'élargissement à des langages réguliers, nous considérons un test d'exactitude, et la terminaison n'est pas assurée.

9.7.1.2 Comparaison avec la technique d'élargissement de [HLR97]

Dans [HLR97], Halbwacks et al. définissent des techniques d'élargissement sur les automates de mots pour calculer des sous-approximations de plus grands points fixes, dans le but de calculer des invariants de réseaux pour les systèmes paramétrés linéaires.

Leur méthode d'élargissement est différente de la nôtre dans plusieurs sens. D'abord, elle n'est définie que sur les automates de mots alors que nous traitons aussi le cas des arbres. La deuxième différence réside dans le fait que notre méthode s'applique à une séquence croissante d'automates pour en calculer une sur-approximation du plus petit point fixe, alors que leur technique s'applique à des séquences décroissantes pour en deviner une sous-approximation du plus grand point fixe. De plus, l'esprit de leur principe d'extrapolation est différent du nôtre : étant donnés deux automates de mots $\mathcal{A}$ et $\mathcal{A}'$, notre technique est basée sur la détection de croissances afin de rajouter des boucles qui permettent de rajouter des comportements supplémentaires aux automates de départ, alors que leur méthode consiste à détecter les états de $\mathcal{A}'$ qui se sont rajoutés et qui font que certains mots acceptés par $\mathcal{A}$ ne le sont plus par $\mathcal{A}'$. Leur opération d'extrapolation consiste alors à enlever ces états et à rajouter des boucles qui font que l'automate obtenu a moins de comportements que l'automate initial.

9.7.2 Travaux sur le calcul de clôtures transitives

9.7.2.1 Comparaison avec les techniques d'extrapolation de [BLW03]

Dans un travail récent [BLW03], une technique d'extrapolation basée sur la comparaison des différentes puissances d'un transducteur de mots $\mathcal{T}$ afin de détecter les croissances et deviner la limite de l'itération de $\mathcal{T}$ a été considérée. Cette technique a été appliquée aux transducteurs représentant des opérations arithmétiques. Cette méthode, qui a le même esprit que la nôtre, présente certaines différences. Une première différence réside dans le fait qu'au lieu de comparer à chaque fois chaque transducteur avec son successeur immédiat, la technique exposée dans cet article consiste à comparer les transducteurs en suivant une certaine période. En effet, ils ont observé que pour les transducteurs représentant des opérations arithmétiques, comparer les puissances successives ne permet pas de détecter des croissances ; et que dans ce cas, il est plus judicieux de comparer les puissances de $\mathcal{T}$ qui sont puissances de 2, c-à-d., comparer $\mathcal{T}^{2^i}$ à $\mathcal{T}^{2^{i+1}}$. Notre technique permet aussi de comparer les automates obtenus après une certaine période, mais ceci se fait après avoir comparé l'automate initial à tous les automates intermédiaires. Donc, dans ce cas des transducteurs arithmétiques, la méthode de [BLW03] serait moins coûteuse.

La deuxième différence réside dans la manière de détecter les croissances. Dans [BLW03], les auteurs manipulent des automates minimaux, et comparent donc les langages des automates pour détecter les incréments ; alors que notre méthode consiste à comparer les structures des automates en utilisant le critère de bisimulation. De plus, même si nous considérons des automates minimaux, nos techniques de comparaison ne sont pas les mêmes. En effet, pour comparer $\mathcal{T}$ à $\mathcal{T}'$, la méthode de [BLW03] consiste à détecter les croissances en déterminant les états de $\mathcal{T}$ et $\mathcal{T}'$ qui sont bisimilaires en arrière (resp. en avant) : ce sont les états d'entrée (resp. de sortie) de l'incrément. Quant à notre technique, elle consiste à deviner une croissance dans $\mathcal{T}'$, l'enlever, fusionner les états d'entrée aux états de sortie correspondants, et ensuite vérifier que l'automate obtenu est bisimilaire à $\mathcal{T}$. Comme nous fusionnons les états avant de tester la bisimilarité, nous ne garantissons pas que les états d'entrée ou de sortie équivalents de l'incrément sont bisimilaires dans $\mathcal{T}'$. La condition de [BLW03] est donc plus exigente

que la nôtre. Nous pouvons cependant l'utiliser dans notre cadre comme stratégie de détection de croissances.

De plus, notre méthode permet d'extrapoler dès qu'une croissance est détectée, alors que dans [BLW03], il n'est possible d'extrapoler qu'après avoir détecté le même incrément un nombre fixé k de fois (ce qui revient à comparer k puissances de $\mathcal{T}$). Ceci écarte les possibilités d'extrapolation lorsque l'incrément détecté ne peut être répété qu'un nombre fixé, inférieur à k, de fois. Il serait intéressant de voir si en appliquant cette idée, nous pouvons retrouver nos résultats de complétude sans utiliser le test 9.3.

D'un autre côté, les deux techniques d'extrapolation sont différentes : nous extrapolons en fusionnant les états équivalents d'entrée/sortie ; alors que dans [BLW03], pour pouvoir extrapoler et permettre à l'incrément d'être répété un nombre arbitraire de fois, l'idée est d'ajouter des boucles en rajoutant des arcs entre les états "équivalents" des différents incréments.

Pour tester l'exactitude de cette extroplation, les auteurs proposent un critère suffisant basé sur des automates à compteurs. L'idée est de vérifier que tout comportement du transducteur avec un nombre k d'incréments peut être obtenu en composant les transducteurs ayant moins de k incréments. Il serait intéressant de comparer ce critère avec notre test 9.3 valable dans le cas des relations bien-fondées. Une question intéressante serait par exemple de voir si pour toute relation bien-fondée, le résultat du test de [BLW03] est toujours exact.

Il serait intéressant de comparer plus en détails cette technique avec notre méthode d'élargissement, et de voir s'il est possible d'intégrer les deux techniques. Une possibilité serait par exemple de pouvoir utiliser le critère suffisant d'exactitude qu'ils proposent avec notre technique d'élargissement.

9.7.2.2 D'autres techniques basées sur la fusion d'états

D'autres techniques semi-algorithmiques (dont la terminaison n'est pas garantie) basées sur le calcul du transducteur de la clôture transitive en calculant les premières itérations d'un transducteur de mots ou d'arbres $\mathcal{T}$ ont été considérées dans [BJNT00, JN00, DLS01, AJMd02, AJNd02]. Ces techniques ne cherchent pas à détecter des croissances, mais à essayer de calculer le transducteur limite en fusionnant les états "équivalents" selon certains critères. Dans [BJNT00, JN00, AJMd02, AJNd02], il s'agit de commencer à calculer les différentes itérations en composant avec le transducteur à itérer $\mathcal{T}$ tout en gardant trace des séquences d'états obtenues en faisant les produits. Par exemple, les états du transducteur $\mathcal{T}^2$ sont des paires (q, q') d'états de $\mathcal{T}$ qui peuvent être considérées comme des mots de longueur deux. Les états de $\mathcal{T}^3$, sont des triplets d'états de $\mathcal{T}$, et peuvent donc être vus comme des mots de taille trois, etc. Les auteurs donnent un critère syntaxique sur ces séquences d'états qui permet de collapser les états de manière exacte, c-à-d. sans introduire de mauvais comportements. La méthode a été introduite dans [BJNT00, JN00, AJNd02] pour le cas de transducteurs de mots, et a été étendue pour le cas des réétiquetages d'arbres dans [AJMd02]. Elle ne peut cependant pas s'appliquer aux relations d'arbres qui changent la structure. Le semi-algorithme proposé est exact pour toutes les relations régulières (de mots ou de réétiquetages d'arbres). Sa terminaison est assurée pour une sous-classe appelée *local-depth* qui apparaît lors de la modélisation des systèmes paramétrés. Par exemple,

les context-relations sont dans cette classe de relations. Il serait intéressant de voir si notre méthode d'élargissement permet aussi de calculer la clôture transitive pour cette classe de relations.

Dans [DLS01] aussi, il s'agit de commencer par calculer les différentes itérations en composant avec le transducteur à itérer $\mathcal{T}$ et de comparer les états en utilisant la notion de bisimulation en avant et en arrière : Une des conditions nécessaires pour que deux états q_1 et q_2 soient confondus est que l'un soit bisimilaire en avant à un état q, et que l'autre soit bisimilaire en arrière à ce même état q. Cette notion de bisimilarité en avant et en arrière est plus forte que notre test de bisimulation et garantie l'exactitude de l'opération de fusion. En effet, dans notre cas, nous enlevons la croissance, nous fusionnons les états, et après nous testons la bisimilarité, donc deux états qui n'étaient pas bisimilaires (en avant ou en arrière), peuvent le devenir. La technique exposée est générale et en cas de terminaison, donne la clôture transitive *exacte* de $\mathcal{T}$. Cependant, nous ne pouvons pas cerner la puissance de cette méthode puisqu'aucun résultat de complétude n'est démontré.

9.7.2.3 Comparaison avec la méthode de [GK00]

Genet et Klay [GK00] ont défini une technique semi-automatique qui peut s'appliquer pour calculer une sur-approximation de l'ensemble des accessibles $\mathcal{R}^*(\mathcal{L})$ pour tout langage régulier d'arbres $\mathcal{L}$ et tout système de réécriture de termes $\mathcal{R}$ linéaire à gauche, c-à-d. un système contenant des règles de la forme $l[x_1, \ldots, x_n] \rightarrow r[x_1, \ldots, x_n]$, où l est un contexte linéaire. La technique utilisée dans cet article est différente de la nôtre. Elle est basée sur la saturation des règles de l'automate $\mathcal{A}$ qui reconnaît $\mathcal{L}$ comme suit : si $l[x_1, \ldots, x_n] \rightarrow r[x_1, \ldots, x_n]$ est une règle de réécriture de $\mathcal{R}$, et s'il existe une dérivation de $\mathcal{A}$ qui reconnaît $l[q_1, \ldots, q_n]$ dans l'état q, c-à-d. si $l[q_1, \ldots, q_n] \xrightarrow{*}_{\delta} q$ où δ est l'ensemble des règles de transition de $\mathcal{A}$; alors il faut rajouter à δ des règles qui permettent d'avoir la dérivation $r[q_1, \ldots, q_n] \xrightarrow{*}_{\delta} q$. Ceci introduit de nouveaux états, et donc la procédure ne termine presque jamais dans les cas non triviaux. Pour forcer la terminaison, les auteurs considèrent une fonction d'approximation, qui est donnée par l'utilisateur, et qui amène parfois le calcul à terminer. Cette fonction a pour effet de réduire le nombre d'états rajoutés à chaque étape. L'ensemble calculé est une sur-approximation de l'ensemble des accessibles. La précision de cette approximation dépend de la fonction considérée. La limite principale de cette technique réside dans le fait qu'elle n'est pas complétement automatique puisqu'elle nécessite l'intervention de l'utilisateur. Dans [OCKS02], Kouchnarenko et al. ont proposé une version automatique de cette approche en générant des fonctions d'approximation automatiquement. Ces techniques ont été appliquées pour la vérification de protocoles cryptographiques.

9.8 Conclusion

Nous avons défini dans ce chapitre une technique d'accélération générale basée sur la détection de croissances au cours du calcul itératif de l'ensemble des accessibles afin de deviner automatiquement la limite. Cette technique peut être appliquée de manière uniforme indépendamment des classes des relations ou des langages considérées. En

particulier, elle peut être utilisée aussi bien pour les relations qui préservent la structure (dans le cas des systèmes paramétrés) que pour celles qui ne la préservent pas (dans le cas des programmes récursifs parallèles). Nous avons montré que cette technique est assez puissante pour permettre le calcul des clôtures transitives dans le cas de plusieurs sous-classes significatives de relations régulières. De même, nous avons appliqué notre méthode pour analyser plusieurs systèmes tels que des protocoles d'exclusion mutuelle définis sur des structures linéaires et arborescentes.

Du point de vue théorique, notre technique semble donc être très prometteuse pour la vérification automatique des systèmes infinis dont les configurations peuvent être représentées par des mots ou des arbres. Il reste à voir si pratiquement elle peut être implantée de manière peu coûteuse. Il faut donc trouver des structures de données qui permettent de représenter de manière compacte les hypergraphes, et qui permettent surtout de détecter les croissances et de tester la bisimulation entre les hypergraphes de manière efficace. Par exemple, il serait intéressant de voir si les structures introduites dans [Mau99] peuvent être utilisées dans ce cadre. De plus, il nous faut trouver des stratégies efficaces qui permettent de réduire de manière intelligente le nombre de croissances qui peuvent être détectées.

Une autre direction de recherche serait d'étendre ces méthodes d'élargissement au-delà du cadre régulier en combinant les hypergraphes avec des contraintes arithmétiques. L'idée serait par exemple d'extrapoler en rajoutant des boucles munies de formules de Presburger exprimant des contraintes arithmétiques sur le nombre de passages par chaque boucle rajoutée. Intuitivement, si nous partons de a^* et que nous obtenons $a^*ba^*ca^*$, notre méthode actuelle nous calcule $a^*b^*a^*c^*a^*$. Pour être plus précis, nous pouvons rajouter l'information qu'à chaque fois, le nombre de b est égal au nombre de c.

Chapitre 10

Conclusion

10.1 Bilan

Dans cette thèse, nous nous sommes intéressés au problème de model-checking des systèmes infinis. Les contributions principales de ce travail peuvent être résumées comme suit :

Accélération et élargissement dans le cadre du Regular Model Checking. Nous avons défini un cadre général et uniforme dans lequel il est possible de décrire et d'analyser plusieurs types de systèmes infinis. Ce cadre est basé sur la représentation des ensembles de configurations des systèmes par des classes d'automates de mots ou d'arbres, et de leurs relations de transitions par des systèmes de réécriture de mots ou de termes. Le problème de la vérification est ensuite réduit à une analyse d'accessibilité qui consiste à calculer les ensembles des accessibles en avant et en arrière. Ce problème étant en général indécidable, nous avons proposé une technique d'accélération générale appelée *élargissement régulier* qui permet, en cas de terminaison, de calculer (une sur-approximation de) l'ensemble des accessibles. Cette méthode est basée sur la comparaison des différents ensembles obtenus durant le calcul itératif de l'ensemble des accessibles, et sur l'itération des croissances détectées. Un test permet ensuite de vérifier que l'ensemble calculé est une sur-approximation de l'ensemble des accessibles. Ce même test permet dans certains cas de s'assurer que cet ensemble est exactement égal à l'ensemble des accessibles. Nous avons montré que ce principe est assez puissant puisqu'il permet de construire les ensembles des accessibles et les clôtures transitives pour plusieurs classes significatives de relations et de langages pour lesquelles des algorithmes spécifiques ont déjà été proposés dans la littérature.

L'avantage principal de cette technique est qu'elle peut être appliquée aveuglément à toutes les classes de systèmes dont les ensembles de configurations et les relations de transition peuvent être représentés par des langages et des relations réguliers de mots ou d'arbres. Elle peut également être combinée avec des algorithmes de calculs de clôtures transitives spécifiques à des classes bien particulières de langages et de relations.

Nous avons appliqué ce cadre général à la vérification (1) des systèmes paramétrés, et (2) des programmes récursifs parallèles. Nous avons montré que notre technique d'élargissement peut s'appliquer dans ces cas. Nous avons également considéré des classes de langages et de règles de réécritures significatives pour la modélisation de ces systèmes, et nous avons proposé des algorithmes de calcul des accessibles spécifiques à ces classes de systèmes :

Vérification des systèmes paramétrés. Dans le cas des systèmes paramétrés linéaires, nous avons considéré les relations de semi-commutations. Ces relations sont présentes dans la modélisation des systèmes paramétrés, puisqu'elles permettent de modéliser, par exemple, la communication entre les processus voisins. Comme ces relations ne préservent pas la régularité, nous avons considéré une classe d'expressions régulières : les expressions APC. Ces expressions apparaissent naturellement dans la modélisation des systèmes paramétrés linéaires. Nous avons montré que cette classe est effectivement fermée par semi-commutations.

Nous nous sommes ensuite intéressés aux systèmes paramétrés arborescents. Nous avons mis en évidence une classe de systèmes de réécriture de termes (les WOS) qui permet de coder la communication entre père/fils dans un arbre. Nous avons donné un algorithme qui calcule les clôtures transitives des systèmes de cette classe.

Ces deux résultats étendent l'applicabilité du regular model-checking aux systèmes qui peuvent être représentés par des semi-commutations ou des WOSs.

Vérification des programmes récursifs parallèles. Pour l'analyse de ces programmes, nous avons proposé deux approches différentes, basées toutes les deux sur notre cadre de vérification général :

• Dans un premier temps, nous avons considéré les systèmes PRS. Nous avons montré que ce formalisme permet de modéliser les programmes récursifs parallèles. Le modèle obtenu sur-approxime en général les comportements du vrai programme. Nous avons identifié une classe intéressante de programmes pour laquelle notre traduction est précise. Nous avons donné des algorithmes exacts d'analyse de ces systèmes. Pour ce faire, nous avons adopté deux approches. La première consiste à résoudre le problème d'accessibilité de ces systèmes en oubliant certaines équivalences structurelles entre les termes de processus. Nous avons alors proposé des constructions polynômiales de représentants réguliers des ensembles des accessibles pour toute la classe PRS lorsque le "||" n'est pas considéré associatif/commutatif. Ces résultats permettent d'analyser une classe importante de programmes qui comprend la récursivité, la création dynamique de processus, et la synchronisation. Lorsque toutes le équivalences sont considérées, nous nous sommes restreints à la classe PAD qui est plus générale que les systèmes à pile et les systèmes PA, et qui permet de modéliser les programmes en tenant compte de la récursivité, du parallélisme, et des résultats de retour des procédures appelées. Nous avons montré que nos constructions précédentes permettent de calculer en temps polynômial un représentant régulier de l'ensemble des successeurs si l'ensemble initial est régulier. Elles permettent aussi de calculer un représentant régulier des prédécesseurs d'un ensemble de termes régulier et fermé par commutativité du "||". Dans le cas général où l'ensemble de termes ne satisfait pas cette propriété de fermeture, nous

avons réussi à calculer un 0-CTA qui correspond à un représentant de l'ensemble des prédécesseurs. Nous avons également réussi à montré que le problème du vide de l'intersection d'un 0-CTA et d'un automate fini d'arbres est décidable. Ceci permet à notre résultat d'être utilisé dans l'analyse symbolique. Ces résultats permettent d'étendre et d'unifier les techniques d'analyse existantes des systèmes PA et des systèmes à pile vers les systèmes PRS.

La deuxième approche consiste à exploiter le fait qu'un PRS peut être vu comme la combinaison d'un système à pile et d'un réseau de Petri, pour proposer une procédure générique de calcul de l'ensemble des accessibles. Pour définir cette procédure, nous avons représenté les ensembles de termes de processus par des ensembles d'arbres à largeurs non bornées qui sont fermés par commutation des fils des nœuds étiquetés par "||". Nous avons introduit la classe des CH-automates qui permet de reconnaître de tels ensembles, et nous avons montré que cette classe satisfait toutes les bonnes propriétés de décidabilité et de fermeture par les opérations booléennes. Ceci permet de l'utiliser comme structure de représentation symbolique dans notre cadre. Etant donné un système PRS dont le réseau de Petri sous-jacent est effectivement semilinéaire, et un ensemble de termes de processus représenté par un CH-automate ; notre procédure permet de calculer un CH-automate reconnaissant exactement l'ensemble des successeurs. En particulier, cette technique permet de calculer les ensembles exacts des successeurs et des prédécesseurs dans le cas des systèmes PAD. Elle permet également le model-checking des formules EF pour ces systèmes qui, rappelons-le, sont plus généraux que les systèmes à pile et les systèmes PA et permettent de modéliser les programmes en tenant compte du parallélisme (qui n'est pas considéré par les automates à pile) et du retour de résultats aux procédures appelantes (qui n'est pas pris en compte par PA). Dans le cas général, notre procédure permet de calculer des sur-approximations des ensembles des accessibles pour toute la classe PRS.

Les deux méthodes que nous avons adoptées ont chacune ses propres avantages. L'avantage principal de cette deuxième méthode est qu'elle offre un cadre générique et uniforme où toutes les techniques existantes d'analyse des systèmes à pile, des réseaux de Petri et des automates à compteurs peuvent être exploitées pour calculer des sur-approximations des ensembles des accessibles pour toute la classe PRS. Concernant les résultats d'analyse des systèmes PAD, l'avantage principal de la première méthode est qu'elle est souvent polynômiale, alors que la deuxième méthode est exponentielle. De plus, un autre avantage de la première approche est qu'elle permet d'analyser de manière exacte et polynômiale toute la classe PRS si le "||" n'est pas considéré comme associatif/commutatif. Comme nous l'avons déjà mentionné, ceci permet l'analyse exacte d'une classe importante de programmes où récursivité, dynamisme, et synchronisation sont considérés.

• La deuxième approche consiste à considérer des modèles plus riches que PRS qui permettent de modéliser de manière précise les comportements des programmes. Nous avons considéré deux types de programmes : (1) les programmes comprenant un nombre fixe de processus séquentiels qui s'exécutent en parallèle et qui peuvent se synchroniser. Nous avons modélisé ces programmes par des automates à pile communicants. (2) Les programmes avec création dynamique de processus et récursivité, mais où les procédures appelées ne peuvent pas rendre de résultats aux procédures

appelantes. Nous avons modélisé ces programmes par des SPA.

Ces deux modèles étant indécidables, nous avons proposé de résoudre le problème d'accessibilité en calculant des abstractions des langages de chemins d'exécutions. Pour ce faire, nous avons proposé un cadre générique et uniforme de calcul de ces abstractions qui est basé sur (1) la représentation des ensembles de configurations par des automates de mots ou d'arbres, (2) la caractérisation des langages de chemins par un système de contraintes défini à partir de ces automates, et (3) la résolution de ces contraintes dans un domaine abstrait. L'avantage de ce cadre est qu'il peut être instancié par n'importe quelle classe d'abstraction pour donner des techniques d'analyse ayant différents coûts et différentes précisions.

Avec ces deux approches, nous avons contribué à étendre les techniques de model-checking pour la vérification automatique des programmes récursifs parallèles. Les techniques que nous avons proposé étendent les techniques d'analyse existantes pour pouvoir traiter des classes plus importantes de programmes.

10.2 Perspectives

A la fin de chaque chapitre, nous avons mentionné quelques perspectives spécifiques au contenu du chapitre. Nous allons maintenant proposer d'autres directions de recherche qui sont plus générales et qui concernent tout le contenu de la thèse.

Etendre et implanter les techniques d'élargissement. La méthode d'élargissement que nous avons introduit est, de point de vue théorique, très prometteuse pour la vérification des systèmes infinis. En effet, elle a permis de calculer les ensembles des accessibles pour le cas de plusieurs classes significatives de langages et de relations. De plus, nous l'avons appliquée avec succès à la vérification des systèmes paramétrés, et des programmes récursifs parallèles. Il serait alors intéressant d'implanter cette technique dans un outil performant qui pourrait alors s'appliquer de manière uniforme à tous les systèmes dont les ensembles de configurations sont représentés par des automates de mots ou d'arbres. Pour ce faire, il nous faut, comme nous l'avons déjà mentionné, trouver des structures de représentation optimales des hypergraphes qui permettent de tester efficacement la bisimulation, ainsi que des stratégies de détection de croissances qui réduisent le nombre d'incréments détectées, tout en restant complètes dans les cas intéressants.

De plus, nous aimerions étendre ces techniques d'élargissement au-delà du cadre régulier, et définir pour ce faire une sorte d'hypergraphes combinés avec des contraintes linéaires comme fait dans [BH97]. Ceci permettrait d'appliquer cette technique de manière exacte dans le cas où l'ensemble des accessibles n'est pas régulier.

Egalement, dans [Mon02, GL00, CCM01, GK00], les automates d'arbres sont utilisés pour modéliser et analyser les protocoles cryptographiques. Il serait intéressant de voir si nos techniques d'élargissement peuvent s'appliquer dans ce cadre.

Etendre l'applicabilité du cadre général de vérification. Le cadre de vérification présenté dans cette thèse peut s'appliquer de manière uniforme à tous les systèmes

dont les configurations peuvent être représentées par des mots ou des arbres. Il serait intéressant de l'étendre à d'autres topologies telles que les grilles ou les graphes en général. Ceci permettrait de vérifier des classes plus importantes de systèmes, notamment les systèmes paramétrés dont les processus sont disposés dans une architecture quelconque, ou les programmes récursifs parallèles manipulant des pointeurs, etc. Pour ce faire, il faut définir des classes d'automates ou de grammaires (1) qui reconnaissent des grilles, des DAG, ou d'autres classes de graphes en général ; (2) qui vérifient toutes les bonnes propriétés de décidabilité et de fermeture nécessaires pour l'analyse d'accessibilité symbolique ; et (3) il faut définir des techniques d'accélération pour ces nouvelles classes, en essayant par exemple d'étendre le principe de l'élargissement à ces topologies, ou de trouver des algorithmes de fermeture spécifiques à des classes de systèmes particulières.

De manière générale, pour pouvoir appliquer ce cadre de vérification à plus de systèmes, il serait intéressant d'identifier d'autres classes de règles de réécriture de mots, d'arbres, ou de graphes $\mathcal{C}_R$ qui sont significatives dans la modélisation de systèmes infinis ; et de mettre en évidence d'autres classes de langages $\mathcal{C}_L$ de mots, d'arbres, ou de graphes qui satisfont les propriétés de fermeture et de décidabilité exigées pour les structures de représentation symboliques ; et pour lesquelles on sait proposer des algorithmes efficaces de calcul de $\mathcal{R}^*$ ou de $\mathcal{R}^*(\mathcal{L})$ pour toute relation $\mathcal{R}$ de la classe $\mathcal{C}_R$ et tout langage $\mathcal{L}$ de la classe $\mathcal{C}_L$.

D'un autre côté, dans beaucoup de systèmes paramétrés, il y a en général deux sources d'infini : la paramétrisation et les données qui peuvent appartenir à des domaines infinis tels que les entiers, les réels, etc. Dans ce travail, nous n'avons tenu compte que de l'aspect paramétrisation. De même, les programmes récursifs parallèles présentent trois sources d'infini : (1) les appels récursifs qui peuvent avoir une profondeur arbitraire, (2) la création dynamique de processus qui permet d'avoir un nombre quelconque de processus en parallèle, et (3) les données qui appartiennent en général à des domaines infinis. Dans les méthodes d'analyse des programmes récursifs parallèles proposées dans ce travail, nous avons tenu compte des appels récursifs et du dynamisme, mais nous avons abstrait les domaines des données en des domaines finis. Parfois, les valeurs exactes de ces données sont intéressantes pour la propriété que l'on veut vérifier. Il serait alors intéressant d'essayer de combiner les techniques présentées dans cette thèse avec les méthodes d'analyse des systèmes manipulant des données de types infinis tels que les compteurs ou les horloges. Plus précisément, il serait intéressant d'intégrer nos techniques avec les techniques implantées dans HyTech [HHWT97], Uppaal [BLL+95], Kronos [DOTY95], LASH [las], ou TREX [ABS01].

Combiner les deux approches d'analyse des programmes récursifs parallèles. Nous avons considéré dans ce travail deux approches différentes pour l'analyse des programmes récursifs parallèles. L'une modélise le programme de manière approchée par des PRS sans tenir toujours compte de la synchronisation (dans le cas général), et analyse ces systèmes de manière précise en calculant leurs ensembles des accessibles. L'autre modélise les programmes de manière précise par des modèles indécidables, et analyse ces systèmes de manière abstraite en calculant des abstractions des langages de chemins d'exécutions du système. Il serait alors intéressant de combiner ces deux

modélisations et ces deux approches pour avoir des modèles et des méthodes d'analyse plus précis qui permettent d'analyser des classes plus importantes de programmes.

Plus précisément, le modèle SPA est une extension de PA qui permet de tenir compte de la synchronisation. Il serait intéressant de voir si l'on peut définir un modèle "SPAD" (pour Synchronized PAD) qui permettrait d'étendre les systèmes PAD en considérant la synchronisation, et voir si l'on peut combiner la construction donnée à la section 5.5 avec les techniques appliquées aux SPA pour pouvoir analyser cette nouvelle classe. Ceci permettrait d'étendre les techniques présentées dans cette thèse pour pouvoir traiter de manière plus précise les programmes avec récursivité, création dynamique de processus, et possibilité de retour de résultats aux procédures appelantes.

Un passage automatique des systèmes vers les modèles. Les traductions que nous avons proposées des programmes récursifs parallèles représentés par des parallel flow graphs vers les PRS, les CPDS, ou les SPA peuvent être facilement automatisées, ce qui permet à nos techniques d'être complètement automatiques et d'être utilisées dans les milieux industriels par des personnes qui ne maîtrisent pas la théorie des langages et des systèmes de réécriture. Cependant, les modèles des systèmes paramétrés que nous considérons sont obtenus manuellement. Il serait alors intéressant de définir des techniques d'abstraction qui permettraient le passage automatique d'un système vers un modèle basé sur les automates et les systèmes de réécriture.

Annexe A

Application de la technique d'élargissement à l'analyse de protocoles d'exclusion mutuelle

Nous montrons dans cette annexe comment modéliser les protocoles d'exclusion mutuelle de Dijkstra et de Szymanski dans notre cadre ; et comment analyser ces protocoles en utilisant notre technique d'élargissement régulier.

A.1 L'algorithme de Dijkstra

A.1.1 L'algorithme

L'algorithme est représenté à la figure A.1. Le principe est le suivant : chaque processus dispose d'une variable *état* appartenant à $\{0, 1, 2\}$ pour indiquer s'il souhaite entrer dans la section critique : *état(i)*> 0 exprime que le processus i veut entrer dans la section critique. Le processus qui est pointé par la variable globale *tour* est prioritaire pour accéder à la ressource commune, pendant que les autres processus attendent à la ligne 1b. Dès que *état(tour)* devient égal à 0, ce qui veut dire que le processus pointé par *tour* quitte la section critique, un des processus qui attendent à la ligne 1b va accéder à la section citique et passe à la ligne 2 et puis 3. La boucle "pour" va tester s'il y a un seul processus qui a accédé à la ligne 3. Si c'est le cas, ce processus entre à la section critique, sinon, tous les processus recommencent à partir de la ligne 1a.

tour$\in \{1, ..., n\}$, pouvant être lue et mise à jour par tous les processus
Pour chaque processus i, $1 \leq i \leq n$, *état(i)*$\in\{0,1,2\}$, initiallement =0. La variable *état(i)* peut être lue par tous les processus mais, seul i peut la modifier.
Process i :
0 : init
1a : *état(i)* :=1
1b : tantque *tour*$\neq i$ faire
SI *état(tour)*=0 faire
tour :=i
fSI
ftantque
2 : *état(i)* :=2
3 : pour $j \neq i$ faire
SI *état(j)*=2
aller à 1a
fSI
fpour
entrer dans la section critique
sortir de la section critique
état(i) :=0
init

FIG. A.1 – L'algorithme de Dijkstra

A.1.2 Modélisation

L'état local de chaque processus i dépend de son état de contrôle (*pc(i)*=0,...,3)[1], et de deux variables : *état* et *tour*. *état* pouvant être égale à 0, 1 ou 2 et *tour* est, comme nous venons de l'expliquer, une variable globale qui pointe sur un seul processus à la fois. Nous posons *t(i)=0* si *tour*$\neq i$ et *t(i)=1* si *tour=i*. Ainsi, l'état d'un processus peut être représenté par le vecteur suivant : *(pc(i),état(i),t(i))*.

En suivant l'algorithme de Dijkstra, un processus peut être dans l'un des états suivants :

$$\begin{cases} c_0 = (0,0,0) \\ c'_0 = (0,0,1) \\ c_1 = (1,1,0) \\ c'_1 = (1,1,1) \\ c_2 = (2,2,1) \\ c'_2 = (2,2,0) \\ c_3 = (3,2,1) \end{cases}$$

Il est clair que la ligne 1a de l'algorithme peut être représentée par la relation

[1] *pc(i)* correspond à la ligne de l'algorithme dans laquelle se trouve le processus i. Pour simplifier l'analyse, nous ne faisons pas de distinction entre les lignes 1a et 1b.

suivante :

$$copy(\Sigma^*)\big((c_0, c_1) + (c'_0, c'_1)\big)copy(\Sigma^*).$$

Selon la ligne 1b, étant donné un processus i tel que *état(i)=1*, s'il existe un processus j qui possède le *tour* et tel que *état(j)=0*, alors, i peut s'affecter le *tour*. Ceci veut dire que si un processus i est dans l'état c_1, et s'il existe un processus j dans l'état c'_0, alors, i peut passer de l'état c_1 à l'état c'_1 et j de l'état c'_0 à l'état c_0 (puisqu'il n'y a qu'un seul processus qui détient le *tour*).

Cette ligne du programme peut être représentée par les relations suivantes :

$$copy(c_0 + c_1 + c'_2)^*(c_1, c'_1)copy(c_0 + c_1 + c'_2)^*(c'_0, c_0)copy(c_0 + c_1 + c'_2)^*$$

et

$$copy(c_0 + c_1 + c'_2)^*(c'_0, c_0)copy(c_0 + c_1 + c'_2)^*(c_1, c'_1)copy(c_0 + c_1 + c'_2)^*.$$

D'après la ligne 2 de l'algorithme, si un processus i posséde le *tour* et *état*(i)=1 alors, *état*(i)=2. En d'autres termes, si un processus est dans l'état c'_1, il peut passer à l'état c_2. Ceci est représenté par la relation $copy(c_0+c_1+c'_2)^*(c'_1, c_2)copy(c_0+c_1+c'_2)^*$. Notons que, pour cette dernière relation, les conditions à droite et à gauche de l'endroit de réécriture ne sont pas représentées par Σ^*. En effet, il faut tenir compte du fait que *tour* ne peut pointer que sur un seul processus à la fois. Puisqu'ici *tour* pointe sur le processus i qui est à l'état c'_1, tous les processus j à droite et à gauche de i doivent avoir *t(j)*=0, c'est-à-dire doivent être dans l'un des états suivants : c_0, c_1 ou c'_2.

Selon la ligne 3 de l'algorithme, si un processus i est dans l'état c_2 et s'il n'existe aucun autre processus ayant un *état*=2 (c'est-à-dire si aucun autre processus n'est dans les états c_2 ou c_3), alors, i peut passer de l'état c_2 à l'état c_3 . Ceci est représenté par la relation

$$copy(c_0 + c'_0 + c_1 + c'_1)^*(c_2, c_3)copy(c_0 + c'_0 + c_1 + c'_1)^*.$$

Finalement, à sa sortie de la section critique, chaque processus doit mettre son *état* à 0 tout en gardant le *tour*. Ce qui peut être décrit par la relation :

$$copy(\Sigma^*)(c_3, c'_0)copy(\Sigma^*).$$

Ainsi, l'algorithme de Dijkstra peut être représenté par les relations régulières représentées à la figure A.2.

$\mathcal{R}_1$: $copy(\Sigma^*)\big((c_0, c_1) + (c'_0, c'_1)\big)copy(\Sigma^*)$,
$\mathcal{R}_2$: $copy(c_0 + c_1 + c'_2)^*(c_1, c'_1)copy(c_0 + c_1 + c'_2)^*(c'_0, c_0)copy(c_0 + c_1 + c'_2)^*$,
$\mathcal{R}_3$: $copy(c_0 + c_1 + c'_2)^*(c'_0, c_0)copy(c_0 + c_1 + c'_2)^*(c_1, c'_1)copy(c_0 + c_1 + c'_2)^*$,
$\mathcal{R}_4$: $copy(c_0 + c_1 + c'_2)^*(c'_1, c_2)copy(c_0 + c_1 + c'_2)^*$,
$\mathcal{R}_5$: $copy(c_0 + c'_0 + c_1 + c'_1)^*(c_2, c_3)copy(c_0 + c'_0 + c_1 + c'_1)^*$,
$\mathcal{R}_6$: $copy(\Sigma^*)(c_3, c'_0)copy(\Sigma^*)$.

FIG. A.2 – Relations représentant l'algorithme de Dijkstra

La section critique étant représentée par l'état c_3, la propriété d'exclusion mutuelle peut être écrite sous la forme suivante : $\neg(\Sigma^* c_3 \Sigma^* c_3 \Sigma^*)$. L'ensemble des configurations

L_0	$c_0^* c_0' c_0^*$	$\mathcal{R}_1^*$	L_1	
L_1	$(c_0+c_1)^*(c_0'+c_1')(c_0+c_1)^*$	$\mathcal{R}_4^+$ $\mathcal{R}_2^+$ $\mathcal{R}_3^+$	L_2 L_3 L_4	
L_2	$(c_0+c_1)^* c_2 (c_0+c_1)^*$	$\mathcal{R}_5^+$	L_5	
L_3	$(c_0+c_1)^* c_1' (c_0+c_1)^* c_0 (c_0+c_1)^*$	$\mathcal{R}_4^+$		$\subseteq L_2$
L_4	$(c_0+c_1)^* c_0 (c_0+c_1)^* c_1' (c_0+c_1)^*$	$\mathcal{R}_4^+$		$\subseteq L_2$
L_5	$(c_0+c_1)^* c_3 (c_0+c_1)^*$	$\mathcal{R}_6^+$	L_6	
L_6	$(c_0+c_1)^* c_0' (c_0+c_1)^*$			$\subset L_1$

TAB. A.1 – Analyse d'accessibilité de l'algorithme de Dijkstra

initiales est représenté par $c_0^* c_0' c_0^*$. En effet, initialement, la valeur de la variable *état* est nulle pour tous les processus et la variable globale *tour* pointe sur un seul processus : un seul processus est dans l'état c_0' et tous les autres dans l'état c_0.

A.1.3 Analyse d'accessibilité du protocole de Dijkstra

Comme fait dans le cas du "Bakery algorithm", nous calculons à l'aide de l'élargissement les relations $\mathcal{R}_i^+$. Remarquons que toutes les relations de ce protocole sont bien-fondées (par le théorème 9.3.1). Nous obtenons :

- $\mathcal{R}_1^* = \big(copy(\Sigma) + (c_0, c_1)\big)^* \big((c_0', c_1') + (c_0', c_0')\big)\big(copy(\Sigma) + (c_0, c_1)\big)^*$,
- $\mathcal{R}_2^+ = \mathcal{R}_2$,
- $\mathcal{R}_3^+ = \mathcal{R}_3$,
- $\mathcal{R}_4^+ = \mathcal{R}_4$,
- $\mathcal{R}_5^+ = \mathcal{R}_5$,
- $\mathcal{R}_6^+ = \mathcal{R}_6$,
- $\mathcal{R}_7^+ = \mathcal{R}_7$,
- $\mathcal{R}_8^+ = copy(\Sigma^*)(c_3, c_0')\big(copy(\Sigma) + (c_3, c_0')\big)^*$.

L'analyse d'accessibilité est représentée au tableau A.1 : nous partons de l'ensemble L_0 des configurations initiales, et nous appliquons, successivement, $\mathcal{R}^+$ ou $\mathcal{R}^*$, pour toutes les relations. Nous obtenons de nouveaux ensembles L_i jusqu'à ce qu'aucun nouvel ensemble ne peut être généré. Un algorithme automatique calcule, toujours, $\mathcal{R}^*$ mais pour des raisons de présentation, nous calculons parfois $\mathcal{R}^+$.

Comme dans les tableaux (9.1) et (9.2), dans la première et la deuxième colonnes, nous donnons l'ensemble des configurations auquel nous appliquons la relation. Le choix de la relation figure dans la troisième colonne et le résultat obtenu dans la colonne quatre. Si, dans une certaine étape, L_i est inclus dans un ensemble déjà calculé, les successeurs de L_i ne sont pas pris en compte. Ceci est indiqué dans la colonne 5.

Après avoir calculé l'ensemble des accessibles, nous vérifions que l'intersection entre l'union des L_i calculés et l'ensemble des mauvaises configurations (ici $\Sigma^* c_3 \Sigma^* c_3 \Sigma^*$) est vide. Nous en déduisons que l'algorithme de Dijkstra satisfait bien la propriété d'exclusion mutuelle.

A.2 L'algorithme de Szymanski

A.2.1 Modélisation

Nous considérons ici la modélisation donnée dans [ABJN99]. L'ensemble des états des processus est Σ=$\{c_1$,c_2,c_3,c_4,c_5,c_6, $c_7\}$, où c_7 représente la section critique ; l'ensemble des configurations initiales est représenté par ${c_1}^*$, et celui des mauvaises configurations est donné par : $\Sigma^* c_7 \Sigma^* c_7 \Sigma^*$. Les relations qui modélisent les transitions du système sont représentées à la figure A.3.

$\mathcal{R}_1$: $copy((c_1 + c_2 + c_4)^*)(c_1, c_2)copy((c_1 + c_2 + c_4)^*)$
$\mathcal{R}_2$: $copy(\Sigma^*)(c_2, c_3)copy(\Sigma^*)$
$\mathcal{R}_3$: $copy(\Sigma^*)copy(c_2 + c_5 + c_6 + c_7)copy(\Sigma^*)(c_3, c_4)copy(\Sigma^*)$.
$\mathcal{R}_4$: $copy(\Sigma^*)(c_3, c_4)copy(\Sigma^*)copy(c_2 + c_5 + c_6 + c_7)copy(\Sigma^*)$
$\mathcal{R}_5$: $copy((c_1 + c_3 + c_4)^*)(c_3, c_5)copy((c_1 + c_3 + c_4)^*)$
$\mathcal{R}_6$: $copy(\Sigma^*)copy(c_5 + c_6 + c_7)copy(\Sigma^*)(c_4, c_5)copy(\Sigma^*)$
$\mathcal{R}_7$: $copy(\Sigma^*)(c_4, c_5)copy(\Sigma^*)copy(c_5 + c_6 + c_7)copy(\Sigma^*)$
$\mathcal{R}_8$: $copy((c_1 + c_2 + c_5 + c_6 + c_7)^*)(c_5, c_6)copy((c_1 + c_2 + c_5 + c_6 + c_7)^*)$
$\mathcal{R}_9$: $copy((c_1 + c_2 + c_4)^*)(c_6, c_7)copy(\Sigma^*)$
$\mathcal{R}_{10}$: $copy(\Sigma^*)(c_7, c_1)copy(\Sigma^*)$

FIG. A.3 – Modélisation de l'algorithme de Szymanski

A.2.2 Analyse d'accessibilité

Nous montrons dans ce qui suit comment le mécanisme d'élargissement permet d'analyser le protocole de Szymanski. Nous calculons les $\mathcal{R}_i^*$ par élargissement puisque toutes ces relations sont des context-relations (d'après le théorème 9.4.1, l'élargissement permet de calculer les clôtures transitives des context-relations). Nous appliquons les relations obtenues itérativement à l'ensemble des configurations initiales. L'analyse est représentée au tableau A.2. Nous voyons que ce calcul itératif converge. Il est alors facile de vérifier que l'intersection des ensembles des accessibles avec celui des mauvaises configurations est vide. Nous en déduisons que l'algorithme de Szymanski satisfait bien la propriété d'exclusion mutuelle.

L_0	c_1^*	$\mathcal{R}_1^*$	L_1	
L_1	$(c_1 + c_2)^*$	$\mathcal{R}_2^*$	L_2	
L_2	$(c_1 + c_2 + c_3)^*$	$\mathcal{R}_3^+ \cup \mathcal{R}_4^+$ $\mathcal{R}_5^+$	L_3 L_4	
L_3	$(c_1 + c_2 + c_3)^* c_2 (c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^*$ $+$ $(c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^* c_2 (c_1 + c_2 + c_3)^*$	$\mathcal{R}_3^*$	L_8	
L_4	$(c_1 + c_3)^* c_5 (c_1 + c_3)^*$	$\mathcal{R}_8^+$	L_5	
L_5	$c_1^* c_6 c_1^*$	$\mathcal{R}_9^*$	L_6	
L_6	$c_1^* c_7 c_1^*$	$\mathcal{R}_{10}^+$	L_7	
L_7	$c_1^* c_1 c_1^*$			$\subset L_0$
L_8	$(c_1 + c_2 + c_3)^* c_2 (c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^*$ $+$ $(c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^* c_2 (c_1 + c_2 + c_3 + c_4)^*$	$\mathcal{R}_4^*$	L_9	
L_9	$(c_1 + c_2 + c_3 + c_4)^* c_2 (c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^*$ $+$ $(c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^* c_2 (c_1 + c_2 + c_3 + c_4)^*$	$\mathcal{R}_5^+$	L_{11}	
L_{10}	$(c_1 + c_2 + c_3 + c_4)^* (c_2 + c_3) (c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^*$ $+$ $(c_1 + c_2 + c_3 + c_4)^* c_4 (c_1 + c_2 + c_3 + c_4)^* (c_2 + c_3) (c_1 + c_2 + c_3 + c_4)^*$	$\mathcal{R}_2^*$	L_{10}	
L_{11}	$(c_1 + c_3 + c_4)^* c_5 (c_1 + c_3 + c_4)^* c_4 (c_1 + c_3 + c_4)^*$ $+$ $(c_1 + c_3 + c_4)^* c_4 (c_1 + c_3 + c_4)^* c_5 (c_1 + c_3 + c_4)^*$	$\mathcal{R}_6^*$	L_{12}	
L_{12}	$(c_1 + c_3 + c_4)^* c_5 (c_1 + c_5 + c_3 + c_4)^* (c_4 + c_5) (c_1 + c_5 + c_3 + c_4)^*$ $+$ $(c_1 + c_5 + c_3 + c_4)^* (c_4 + c_5) (c_1 + c_5 + c_3 + c_4)^* c_5 (c_1 + c_3 + c_4)^*$	$\mathcal{R}_7^*$	L_{13}	
L_{13}	$(c_1 + c_3 + c_4 + c_5)^* c_5 (c_1 + c_5 + c_3 + c_4)^* (c_4 + c_5) (c_1 + c_5 + c_3 + c_4)^*$ $+$ $(c_1 + c_5 + c_3 + c_4)^* (c_4 + c_5) (c_1 + c_5 + c_3 + c_4)^* c_5 (c_1 + c_3 + c_4 + c_5)^*$	$\mathcal{R}_8^+$	L_{14}	
L_{14}	$(c_1 + c_6 + c_5)^* c_6 (c_1 + c_5 + c_6)^* (c_6 + c_5) (c_1 + c_5 + c_6)^*$ $+$ $(c_1 + c_5 + c_6)^* (c_6 + c_5) (c_1 + c_5 + c_6)^* c_6 (c_1 + c_6 + c_5)^*$	$\mathcal{R}_9^+$	L_{15}	
L_{15}	$c_1^* c_7 (c_1 + c_5 + c_6)^* (c_6 + c_5) (c_1 + c_5 + c_6)^*$	$\mathcal{R}_{10}$	L_{16}	
L_{16}	$c_1^* c_1 (c_1 + c_5 + c_6)^* (c_6 + c_5) (c_1 + c_5 + c_6)^*$	$\mathcal{R}_9$	L_{17}	
L_{17}	$c_1^* c_1 c_7 (c_1 + c_5 + c_6)^*$	$\mathcal{R}_{10}$	L_{18}	
L_{18}	$c_1^* c_1 c_1 (c_1 + c_5 + c_6)^*$	$\mathcal{R}_9^+$	L_{19}	
L_{19}	$c_1^* c_1 c_1 c_7 (c_1 + c_5 + c_6)^*$			$\subset L_{17}$

TAB. A.2 – Analyse d'accessibilité de l'algorithme de Szymanski

Bibliographie

[AAB99] P. Abdulla, A. Annichini, and A. Bouajjani. Symbolic Verification of Lossy Channel Systems : Application to the Bounded Retransmission Protocol. In *TACAS'99*. LNCS 1579, 1999.

[ABH+97] Rajeev Alur, Robert K. Brayton, Thomas A. Henzinger, Shaz Qadeer, and Sriram K. Rajamani. Partial-order reduction in symbolic state space exploration. In *Computer Aided Verification*, pages 340–351, 1997.

[ABJ98] P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly Analysis of Systems with Unbounded, Lossy Fifo Channels. In *CAV'98*. LNCS 1427, 1998.

[ABJN99] P. A. Abdulla, A. Bouajjani, B. Jonsson, and M. Nilsson. Handling global conditions in parametrized system verification. *Lecture Notes in Computer Science*, 1633 :134–150, 1999.

[ABS01] Aurore Annichini, Ahmed Bouajjani, and Mihaela Sighireanu. Trex : A tool for reachability analysis of complex systems. In *Computer Aided Verification*, pages 368–372, 2001.

[ACaJT96] P. Abdulla, K. Cerans, B. Jonsson, and Y.-K. Tsay. General decidability theorems for infinite-state systems. In *Proc. of the 11th Annual IEEE Symposium on Logic in Computer Science*, pages 313–321. IEEE Computer Society Press, 1996.

[ACD90] R. Alur, C. Courcoubetis, and D. Dill. Model-checking for real-time systems. *5th Symp. on Logic in Computer Science (LICS 90)*, pages 414–425, 1990.

[ACH+95] R. Alur, C. Courcoubetis, N. Halbwachs, T. A. Henzinger, P.-H. 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. Dill. A Theory of Timed Automata. *TCS*, 126, 1994.

[AJ96] P. Abdulla and B. Jonsson. Undecidable verification problems for programs with unreliable channels. *Information and Computation*, 130(1) :71–90, 1996.

[AJ98] Parosh Aziz Abdulla and Bengt Jonsson. Verifying Networks of Timed Processes. In *TACAS'98*. LNCS 1384, 1998.

[AJMd02] P. A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model-checking. In *14th Intern. Conf. on Computer Aided Verification (CAV'02)*. LNCS, Springer-Verlag, 2002.

[AJNd02] P. A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Regular model-checking made simple and efficient. In *In Proceedings 13th International Conference on Concurrency Theory (CONCUR)*, volume 2421 of *Lecture Notes in Computer Science*, pages 116–130. Springer-Verlag, 2002.

[AK86] K .R. Apt and D. C. Kozen. Limits for automatic verificarion of infinite-state concurrent systems. *Information Processing Letters*, pages 22 :307–309, 1986.

[AK95] P. Abdulla and M. Kindahl. Decidability of Simulation and Bisimulation between Lossy Channel Systems and Finite State Systems. In *Proc. Intern. Conf. on Concurrency Theory (CONCUR'95)*. LNCS 962, 1995.

[AMP95] E. Asarin, O. Maler, and A. Pnueli. On the analysis of dynamical systems having piecewise-constant derivatives. *Theoretical Computer Science*, 138 :35–65, 1995.

[APR+01] Tamarah Arons, Amir Pnueli, Sitvanit Ruah, Jiazhao Xu, and Lenore Zuck. Parameterized verification with automatically computed inductive assertions. *Lecture Notes in Computer Science*, 2102 :221+, 2001.

[BBLS00] Kai Baukus, Saddek Bensalem, Yassine Lakhnech, and Karsten Stahl. Abstracting WS1S systems to verify parameterized networks. In *Tools and Algorithms for Construction and Analysis of Systems*, pages 188–203, 2000.

[BBS00] J. Blieberger, B. Burgstaller, and B. Scholz. Symbolic Data Flow Analysis for Detecting Deadlocks in Ada Tasking Programs. In *Proc. of the Ada-Europe International Conference on Reliable Software Technologies*, Potsdam, Germany, 2000.

[BCG89] M. C. Browne, E. M. Clarke, and O. Grumberg. Reasoning about networks with many identical finite state processes. *Information and Computation*, 1989.

[BCM+92] J. Burch, E. M. Clarke, K. McMillan, D. Dill, and L. Hwang. Symbolic model-checking : 10^{20} states and beyond. *Information and Computation*, 98(2) :142–170, 1992.

[BCR01] T. Ball, S. Chaki, and S. K. Rajamani. Parameterized verification of multithreaded software libraries. In *TACAS 2001, LNCS 2031*, 2001.

[BE97] O. Burkart and Javier Esparza. More infinite results. In *EATCS Bulletin*, volume 62, pages 138–159, 1997.

[BEM97] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata : Application to Model Checking. In *CONCUR'97*, volume 1243 of *LNCS*, 1997.

[BET03a] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. In *Proceedings of the 30th ACM SIPGPLAN-SIGACT on Principles of Programming Languages, POPL'03*, 2003.

[BET03b] A. Bouajjani, J. Esparza, and T. Touili. A generic approach to the static analysis of concurrent programs with procedures. *International Journal of Foundations of Computer Science*, 2003.

[BET03c] A. Bouajjani, J. Esparza, and T. Touili. Abstract Reachability Analysis of Programs with Dynamic Creation of Threads and Procedure Calls. Technical report, Laboratoire LIAFA, Université de Paris 7, France, February 2003.

[BG96] B. Boigelot and P. Godefroid. Symbolic Verification of Communication Protocols with Infinite State Spaces using QDDs. In *CAV'96*. LNCS 1102, 1996.

[BGWW97] B. Boigelot, P. Godefroid, B. Willems, and P. Wolper. The power of QDDs. In *SAS'97*. LNCS 1302, 1997.

[BH97] A. Bouajjani and P. Habermehl. Symbolic Reachability Analysis of FIFO-Channel Systems with Nonregular Sets of Configurations. In *ICALP'97*. LNCS 1256, 1997.

[BHV03] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of Parametric Concurrent Systems with Prioritized FIFO Resource Management . In *Proceedings of CONCUR'03*, 2003.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *12th Intern. Conf. on Computer Aided Verification (CAV'00)*. LNCS, Springer-Verlag, 2000.

[BK85] J. A. Bergstra and J. W. Klop. Algebra of communicating processes with abstraction. *Theoretical Computer Science*, 37, 1985.

[BKMW01] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular tree and regular hedge languages over unranked alphabets. Research report, 2001.

[BLL+95] Johan Bengtsson, Kim Guldstrand Larsen, Fredrik Larsson, Paul Pettersson, and Wang Yi. UPPAAL - a tool suite for automatic verification of real-time systems. In *Hybrid Systems*, pages 232–243, 1995.

[BLW03] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *15th Intern. Conf. on Computer Aided Verification (CAV'03)*. LNCS, Springer-Verlag, 2003.

[BMT01] A. Bouajjani, A. Muscholl, and T. Touili. Permutation Rewriting and Algorithmic Verification. In *Proc. 17th Symp. on Logic in Computer Science (LICS'01)*. IEEE, 2001.

[Bou01] A. Bouajjani. Languages, Rewriting systems, and Verification of Infinte-State Systems. In *ICALP'01*. LNCS 2076, 2001. invited paper.

[BQ96] O. Burkart and Y.M. Quemener. Model-checking of infinite graphs defined by graph grammars. In *In Proc. 4th Int. Workshop Verification of Infinite State Systems (INFINITY'96)*, 1996.

[BR00] T. Ball and S. Rajamani. Bebop : A symbolic model checker for boolean programs. In *SPIN00 : SPIN Workshop*, volume 1885 of *Lecture Notes in Computer Science*, pages 113–130. Springer Verlag, 2000.

[Bra69] W. S. Brained. Tree generating regular systems. *Information and Control*, 14 :217–231, 1969.

[Bry92] Randal E. Bryant. Symbolic boolean manipulation with oredered binary-decision diagrams. *ACM Computing Surveys*, 24(3) :293–318, September 1992.

[BS95] O. Burkart and B. Steffen. Composition, decomposition, and model-checking of pushdown processes. *Nordic Journal of Computing*, 2(2) :89–125, 1995.

[BS97] O. Burkart and B. Steffen. Model-checking the full modal mu-calculus for infinite sequential processes. In *Proc. ICALP'97*, volume 1256 of *LNCS*, pages 419–429, 1997.

[BT02] A. Bouajjani and T. Touili. Extrapolating tree transformations. In *14th Intern. Conf. on Computer Aided Verification (CAV'02)*. LNCS, Springer-Verlag, 2002.

[BT03a] Ahmed Bouajjani and Tayssir Touili. Effective Reachability Sets of Process Rewrite Systems. Research report, Laboratoire LIAFA, Université de Paris 7, France, 2003.

[BT03b] Ahmed Bouajjani and Tayssir Touili. Reachability Analysis of Process Rewrite Systems. In *FSTTCS'03*, 2003. To appear.

[BW94] B. Boigelot and P. Wolper. Symbolic Verification with Periodic Sets. In *CAV'94*. LNCS 818, 1994.

[BW98] Bernard Boigelot and Pierre Wolper. Verifying systems with infinite but regular state spaces. In *Proc. 10th Int. Conf. on Computer Aided Verification, CAV'98*, volume 1254 of *Lecture Notes in Computer Science*, pages 88–97. Springer Verlag, July 1998.

[BW00] Bernard Boigelot and Pierre Wolper. On the construction of automata from linear arithmetic constraints. In *TACAS'00*, volume 1785 of *Lecture Notes in Computer Science*, pages 1–19. Springer Verlag, 2000.

[Cau92] Didier Caucal. On the regular structure of prefix rewriting. *Theoretical Computer Science*, 106 :61–86, 1992.

[CC77] P. Cousot and R. Cousot. Static Determination of Dynamic Properties of Recursive Procedures. In *IFIP Conf. on Formal Description of Programming Concepts*. North-Holland Pub., 1977.

[CCM01] H. Comon, V. Cortier, and J. Mitchell. Tree automata with one memory, set constraints and ping-pong protocols. In *ICALP'2001*. LNCS 2076, 2001.

[CDG+97] H. Comon, M. Dauchet, R. Gilleron, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications. Available on : http ://www.grappa.univ-lille3.fr/tata, 1997.

[CDGV94] J.L. Coquidé, M. Dauchet, R. Gilleron, and S. Vágvölgyi. Bottom-up tree pushdown automata and rewrite systems. *Theoretical Computer Science*, 127(1) :69–98, 1994.

[CES83] E.M. Clarke, E.A. Emerson, and E. Sistla. Automatic Verification of Finite State Concurrent Systems using Temporal Logic Specifications : A Practical Approach. In *POPL'83*. ACM, 1983.

[CFI96] G. Cécé, A. Finkel, and S. P. Iyer. Unreliable channels are easier to verify than perfect channels. *Information and Computation*, 124(1) :20–31, 1996.

[CGJ95] E. M. Clarke, O. Grumberg, and S. Jha. Verifying parameterised networks using abstraction and regular languages. *Lecture Notes in Computer Science*, 962 :395–407, 1995.

[CH78] Patrick Cousot and Nicholas Halbwachs. Automatic discovery of linear restraints among variables of a program. In *POPL'78*. ACM, 1978.

[CH88] T. Coquand and G. Huet. The calculus of construction. *Formal Methods in System Design*, 76(2), 1988.

[Chr93] S. Christensen. Decidability and Decomposition in Process Algebras. Phd. thesis, Edinburgh University, 1993.

[Col02] T. Colcombet. Rewriting in the partial algebra of typed terms modulo ac. In *In Antonin Kucera and Richard Mayr, editors, Electronic Notes in Theoretical Computer Science, volume 68. Elsevier Science Publishers*, 2002.

[Cor92] J. C. Corbett. Automated Formal Analysis Methods for Concurrent and Real-Time Software. Phd thesis, University of Massachusetts at Amherst, 1992.

[DBR01] G. Delzanno, L. Van Begin, and J.-F. Raskin. Attacking symbolic state explosion in parametrized verification. In *CAV'01*, 2001.

[DBR02] G. Delzanno, L. Van Begin, and J.-F. Raskin. Toward the automated verification of multithreaded java programs. In *TACAS 2002*, 2002.

[DGR95] V. Diekert and editors G. Rozenberg. *The Book of Traces*. World Scientific, Singapore, 1995.

[Dil89] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In *In Proc. of the International Workshop on Automartic Verification Methods for Finite State Systems*, volume 407, pages 197–212. LNCS, 1989.

[DLS01] Dennis Dams, Yassine Lakhnech, and Martin Steffen". Iterating transducers. In *13th Intern. Conf. on Computer Aided Verification (CAV'01)*. LNCS, Springer-Verlag, 2001.

[DOTY95] C. Daws, A. Olivero, S. Tripakis, and S. Yovine. The tool KRONOS. In *Hybrid Systems III : Verification and Control*, volume 1066, pages 208–219, Rutgers University, New Brunswick, NJ, USA, 22–25 October 1995. Springer.

[DS91] E. Duesterwald and M.L. Soffa. Concurrency analysis in the presence of of procedures using a data-flow framework. In *Proc. of the Symposium on Testing, Analysis, and Verification, Victoria, Canada*, pages 36–48. ACM Press, 1991.

[DT90] M. Dauchet and S. Tison. The theory of ground rewrite systems is decidable. In *Proc. IEEE Conference on Logic in Computer Science (LICS)*, pages 242–248. IEEE Computer Society Press, 1990.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithm for model checking pushdown systems. In *CAV'00*, volume 1885 of *LNCS*, 2000.

[EK99] J. Esparza and J. Knoop. An automata-theoretic approach to interprocedural data-flow analysis. In Wolfgang Thomas, editor, *Proc of Foundations of Software Science and Computation Structure, FoSSaCS'99*, volume 1578 of *Lecture Notes in Computer Science*. Springer, 1999.

[EK02] E. Allen Emerson and Vineet Kahlon. Model-checking large-scale and parametrized resource allocation systems. In *TACAS'02*, 2002.

[EM96] Javier Esparza and Stephan Melzer. Checking system properties via integer programming. In *European Symposium on Programming*, pages 250–264, 1996.

[Eme90] E. A. Emerson. *Temporal and Modal Logic*, volume B, chapter 16, pages 996–1072. Elsevier, 1990.

[EN94] J. Esparza and M. Nielsen. Decidability issues for Petri nets - a survey. *Bulletin of the EATCS*, 52 :85–107, 1994.

[EN95] E. A. Emerson and K. S. Namjoshi. Reasoning about rings. In *Proc. 22th ACM Symposium on Principles of Programming Languages, POPL'95, San Francisco*, January 1995.

[EN96] E. A. Emerson and K. S. Namjoshi. Automatic verification of parametrized synchronous systems. In *Proc. 8th International Conference on Computer Aided Verification, CAV'96*, Rutgers (N.J.), 1996.

[Eng75] Joost Engelfriet. Bottom-up and top-down tree transformations – a comparison. In *Mathematical Systems Theory*, volume 9(3), pages 198–231, 1975.

[EP00] J. Esparza and A. Podelski. Efficient algorithms for pre* and post* on interprocedural parallel flow graphs. In *Proceedings of the 27th ACM SIGPLAN-SIGACT on Principles of Programming Languages, POPL 2000*, pages 1–11. ACM Press, 2000.

[ES01] J. Esparza and S. Schwoon. A bdd-based model checker for recursive programs. In *In Proc. of CAV'01, number 2102 in Lecture Notes in Computer Science, pages 324-336. Springer-Verlag*, 2001.

[Esp97a] Javier Esparza. Decidability of model checking for infinite-state concurrent systems. *Acta Informatica*, 34(2) :85–107, 1997.

[Esp97b] Javier Esparza. Petri nets, commutative context-free grammars, and basic parallel processes. In *Fundamenta Informaticae*, number 31(1), pages 13–25, 1997.

[Esp02] J. Esparza. Grammars as processes. In *Formal and Natural Computing*. LNCS 2300, 2002.

[Fin94] A. Finkel. Decidability of the termination problem for completely specified protocols. *Distributed Computing*, 7(3) :129–135, 1994.

[FJJM92] J-C. Fernandez, C. Jard, T. Jeron, and L. Mounier. "on-the-fly" verification of finite transition systems. *Formal Methods in System Design*, 1992.

[FL02] A. Finkel and J. Leroux. How to compose Presburger-accelerations : Applications to broadcast protocols. In *FSTTCS'02*. LNCS 2556, 2002.

[FO97] L. Fribourg and H. Olsen. Reachability sets of parametrized rings as regular languages. In *Infinity'97*. volume 9 of Electronical Notes in Theoretical Computer Science. Elsevier Science, 1997.

[FP01] Dana Fisman and Amir Pnueli. Beyond regular model checking. *Lecture Notes in Computer Science*, 2245 :156–??, 2001.

[FR93] M. Fischer and M. Rabin. Super-exponential complexity of Presburger arithmetic. Proceedings of the Symposium on the Complexity of Real Computation Processes, 1993.

[FRSB02] A. Finkel, J.-F. Raskin, M. Samuelides, and L. Van Begin. Monotonic extensions of petri nets : Forward and backward search revisited. In *In Proc. 4th Int. Workshop Verification of Infinite State Systems (INFINITY'2002)*, 2002.

[FWW97] A. Finkel, B. Willems, and P. Wolper. A Direct Symbolic Approach to Model Checking Pushdown Systems. In *Infinity'97*, 1997.

[GD89] R. Gilleron and A. Deruyver. The reachability problem for ground TRS and some extensions. In *Proc. Int. Joint Conf. Theory and Practice of Software Development (TAPSOFT'89)*, pages 227–243. LNCS 351, 1989.

[Gil91] R. Gilleron. Decision problems for term rewriting systems and recognizable tree languages. In *Proc. of STACS'91*, volume 480 of *LNCS*, pages 148–159, 1991.

[GK00] T. Genet and F. Klay. Rewriting for cryptographic protocol verification. In *In Proceedings 17th International Conference on Automated Deduction, Pittsburgh (Pen., USA)*. volume 1831 of Lecture Notes in Artificial Intelligence. Springer-Verlag, 2000.

[GL93] S. Graf and C. Loiseaux. A tool for symbolic program verification methods. In *Proc. 5th Int. Conf. on Computer Aided Verification, CAV'97*, volume 697 of *Lecture Notes in Computer Science*, pages 71–84. Springer Verlag, 1993.

[GL00] J. Goubault-Larrecq. A method for automatic cryptographic protocol verification. In *15th IPDPS 2000 Workshops*. LNCS 1800, 2000.

[God96] P. Godefroid. Partial-order methods for the verification of concurrent systems - An approach to the state-explosion problem. *Lecture Notes in Computer Science*, 1032, 1996.

[GS66] S. Ginsburg and E. Spanier. Semigroups, presburger formulas, and languages. *Pacic J. of Mathematics*, (16) :285–296, 1966.

[GS92] S. M. German and A. P. Sistla. Reasoning about systems with many processes. *J. ACM*, 39 :675–735, 1992.

[GW93] P. Godefroid and P. Wolper. Using partial orders for the efficient verification of deadlock freedom and safety properties. *Formal Methods in System Design*, 2(2) :149–164, 1993.

[GZ98] E.P. Gribomont and G. Zenner. Automated verification of szymanski's algorithm. In *TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 424–438. Springer Verlag, 1998.

[Hab98] P. Habermehl. Sur la vérification des systèmes infinis. Phd. thesis, Université Joseph Fourier, Grenoble, France, 1998.

[Har78] M.A. Harrison. *Introduction to Formal Language Theory*. Addison-Wesley Publishing Company, 1978.

[HHWT97] Thomas A. Henzinger, Pei-Hsin Ho, and Howard Wong-Toi. HYTECH : A model checker for hybrid systems. *International Journal on Software Tools for Technology Transfer*, 1(1–2) :110–122, 1997.

[Hir93] Y. Hirshfeld. Petri nets and the equivalence problem. In Springer Verlag, editor, *Proceedings of CSL'93*, volume 832 of *LNCS*, pages 165–174, 1993.

[HJM94] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimulation of normed context free processes. technical report, LFCS, Edinburgh University, 1994.

[HJM96] Y. Hirshfeld, M. Jerrum, and F. Moller. A polynomial algorithm for deciding bisimulation equivalence of normed Basic Parallel Processes. *Journal of Mathematical Structures in Computer Science*, 1996.

[HK99] M.W. Hopkins and D.C. Kozen. Parikh's Theorem in Commutative Kleene Algebra. In *Proc. IEEE Conf. Logic in Computer Science (LICS'99)*. IEEE, 1999.

[HKPM97] G. Huet, G. Kahn, and C. Paulin-Mohring. The coq proof assistant - a tutorial, version 6.1. Technical report, INRIA, August 1997.

[HKPV95] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata ? In *Proc. of the 27th Annual Symposium on Theory of Computing*, pages 373–382. ACM Press, 1995.

[HLR97] N. Halbwachs, D. Lesens, and P. Raymond. Automatic Verification of parametrized linear networks of processes. In *24th ACM Symposium on Principles of Programming Languages, POPL'97*, 1997.

[HNSY94] T. A. Henzinger, X. Nicollin, J. Sifakis, and S. Yovine. Symbolic model-checking for real-time systems. *Information and Computation*, 111(2) :193–244, 1994.

[Hol94] G. Holzmann. Basic SPIN Manual . Technical report, Bell Laboratories, 1994.

[HP79] John E. Hopcroft and Jean-Jacques Pansiot. On the reachability problem for 5-dimensional vector addition systems. In *Journal of Computer and System Sciences*, volume 8, pages 135–159, 1979.

[HRP94] N. Halbwachs, P. Raymond, and Y.E. Proy. Verification of linear hybrid systems by means of convex approximation. In *International Symposium on Static Analysis (SAS'94)*. LNCS 1579, 1994.

[HRY91] R. Howell, L. Rosier, and H. C. Yen. A taxonomy of fairness and temporal logic problems for petri nets. *Theoretical Computer Science*, 82, 1991.

[HU79] J. Hopcroft and J. Ullman. *Introduction to automata Theory, languages, and computation.* Addison-Wesley Publishing Company, 1979.

[Jan90] P. Jancar. Decidability of a temporal logic problem for petri nets. *Theoretical Computer Science*, 74, 1990.

[Jan97] P. Jancar. Decidability Questions for Bisimilarity of Petri Nets and Some Related Problems . Technical report, INRIA, August 1997.

[JHA+96] J. -C. Fernandez, H. Garavel, A. Kerbrat, L. Mounier, R. Mateescu, and M. Sighireanu. CADP : a protocol validation and verification toolbox. In Rajeev Alur and Thomas A. Henzinger, editors, *Proceedings of the Eighth International Conference on Computer Aided Verification CAV*, volume 1102, pages 437–440, New Brunswick, NJ, USA, / 1996. Springer Verlag.

[JJ89] C. Jard and T. Jeron. On-line model-checking for finite linear temporal logic. *International Workshop on Automatic Verification Methods for Finite State Systems*, 407, 1989.

[JL98] E. Jensen and N.A. Lynch. A proof of burn's n-process mutual exclusion algorithm using abstraction. In *TACAS'98*, volume 1384 of *Lecture Notes in Computer Science*, pages 409–423. Springer Verlag, 1998.

[JM95] P. Jancar and F. Moller. Checking regular properties of petri nets. In *Proc. 6th Int. Conf. Concurrency Theory (CONCUR'95)*, volume 962 of *LNCS*, 1995.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *6th TACAS (TACAS 00)*. to appear in LNCS, Springer-Verlag, 2000.

[JRS03] S. Jha, T. Reps, and S. Schwoon. Weighted Pushdown Systems and their application to Interprocedural Dataflow Analysis. In *Proc. of the 10th International Symposium on Static Analysis, SAS 2003*, 2003.

[KM89] R.P. Kurshan and K. McMillan. A structural induction theorem for processes. In *ACM Symp. on Principles of Distributed Computing*, pages 239–247, 1989.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proc. CAV'97*, volume 1254 of *LNCS*, pages 424–435. Springer, June 1997.

[Kno98] J. Knoop. Optimal interprocedural program optimization : A new framework and its applications. Phd. thesis, University of Kiel, Germany. LNCS Tutorial 1428, 1998.

[Koz83] D.C. Kozen. Results on the Propositional μ-calculus. *Theoretical Computer Science*, 27, 1983.

[las] *The Li'ege Automata-based Symbolic Handler (LASH). Available at http ://www.montefiore.ulg.ac.be/ boigelot/research/lash/.*

[Lon93] D. E. Long. Model-checking, abstraction, and compositional verification. Report ecs-lfcs-93-261, Dep. of. Computer Science, University of Edimburgh, 1993.

[LS98] D. Lugiez and Ph. Schnoebelen. The regular viewpoint on PA-processes. In *Proc. 9th Int. Conf. Concurrency Theory (CONCUR'98), Nice, France, Sep. 1998*, volume 1466, pages 50–66. Springer, 1998.

[Lug03] D. Lugiez. Counting and equality constraints for multitree automata. In *Fossacs 2003, to appear*, 2003.

[Mau99] L. Mauborgne. Representation of Sets of Trees for Abstract Interpretation. Phd. thesis, Ecole Normale Supérieure d'Ulm, Paris, October 1999.

[May98] R. Mayr. Decidability and Complexity of Model Checking Problems for Infinite-State Systems. Phd. thesis, Technical University Munich, 1998.

[McM93] K.L. McMillan. *Symbolic Model-Checking : an Approach to the State-Explosion Problem.* Kluwer, 1993.

[McM98] K. McMillan. Getting started with smv. User's manual, Cadence Berkeley Laboratories, USA, 1998.

[MDIS96] R. Melton, D.L. Dill, C. Norris Ip, and U. Stern. Murphi annotated reference manual, release 3.0. Technical report, Stanford University, Palo Alto, California, USA, July 1996.

[Mer91] N. Mercouroff. An algorithm for analyzing communicating processes. In *Mathematical Foundations of Programming Semantics*, volume 598 of *LNCS*, pages 312–25, 1991.

[Mil80] R. Milner. *A Calculus of Communicating Systems*, volume 92 of *LNCS*. Springer Verlag, 1980.

[Mo02] M. Müller-olm. Variations on constants. Habilitation thesis, Dortmund University, 2002.

[Mol96] F. Moller. Infinite results. In Springer Verlag, editor, *Proceedings of CONCUR'96*, volume 1119 of *LNCS*, 1996.

[Mon02] D. Monniaux. Abstracting cryptographic protocols with tree automata. *Science of Computer Programming*, 2002.

[MP90] Z. Manna and A. Pnueli. An exercise in the verification of multi-process programs. In *Beauty in Our Business*, pages 289–301. Springer Verlag, 1990.

[MR98] R. Mayr and M. Rusinowitch. Reachability is decidable for ground ac rewrite systems. In *In Proc. Int. Workshop Verification of Infinite State Systems (INFINITY'98)*, 1998.

[Mus96] A. Muscholl. Über die Erkennbarkeit unendlicher Spuren. Phd. thesis, Stuttgart University, 1996.

[NA98] G. Naumovich and G. Avrunin. A conservative data flow algorithm for detecting all pairs of statement that may happen in parallel. In *Proceedings of the ACM SIGSOFT International Symposium on Foundations of Software Engineering*, pages 24–34. ACM Press, 1998.

[NPW02] T. Nipkow, L. C. Paulson, and M. Wenzel. Isabelle/hol–a proof assistant for higher-order logic. *LNCS*, 2283, 2002.

[OCKS02] F. Oehl, G. Cécé, O. Kouchnarenko, and D. Sinclair. Automatic Approximation for the Verification of Cryptographic Protocols. In *Proceedings of the International Conference on Formal Aspects of Security (FASec'02)*, 2002.

[Opp78] D. Oppen. A $2^{2^{2^n}}$ upper bound on the complexity of Presburger arithmetic. In *Journal of Computer and System Sciences.*, number 16, pages 323–332, 1978.

[ORS92] S. Owre, J. Rushby, and N. Shankar. PPS : a prototype verification system. In *Proc. of the 11th Conf. on Automated Deduction*, LNCS, pages 748–752. Springer-Verlag, 1992.

[Pnu77] A. Pnueli. The temporal logic of programs. In *Proc. of the 18th IEEE Symposium on the Foundations of Computer Science (FOCS-77)*, pages 46–57, 1977.

[PP91] W. Peng and S. Puroshothaman. Data Flow Analysis of Communicating Finite State Machines. *ACM Transactions on Programming Languages and Systems*, 13(3) :399–442, 1991.

[Pre29] M. Presburger. uber die Vollstandingen einer gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. Comptes Rendus du premier Congr'es des Math'ematiciens des Pays slaves, 1929.

[PRZ01] Amir Pnueli, Sitvanit Ruah, and Lenore Zuck. Automatic deductive verification with invisible invariants. In *TACAS'01*, volume 2031 of *Lecture Notes in Computer Science*, 2001.

[PS00] A. Pnueli and E. Shahar. Liveness and acceleration in parametrized verification. In *12th Intern. Conf. on Computer Aided Verification (CAV'00)*. LNCS, Springer-Verlag, 2000.

[PW97] J.-E. Pin and P. Weil. Polynomial closure and unambiguous product. In *Theory of Computing Systems*, volume 30, pages 383–422, 1997.

[PWW98] D.A. Peled, T. Wilke, and P. Wolper. An algorithmic approach for checking closure properties of temporal logic specifications and omega-regular languages. *Theoretical Computer Science*, 195 :183–203, 1998.

[QS82] J-P. Queille and J. Sifakis. Specification and Verification of Concurrent Systems in Cesar. In *Proc. Intern. Symposium on Programming*. LNCS 137, 1982.

[Ram00] G. Ramalingam. Context-sensitive synchronization-sensitive analysis is undecidable. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 22 :416–430, 2000.

[Ric53] H. G. Rice. Classes of recursively enumerable sets and their decision problems. *Trans. AMS*, 89 :25–59, 1953.

[Rin01] M. Rinard. Analysis of multithreaded programs. In Patrick Cousot, editor, *Proc. of the 8th International Symposium on Static Analysis, SAS 2001*, volume 2126 of *LNCS*, 2001.

[Sal88] K. Salomaa. Deterministic tree pushdown automata and monadic tree rewriting systems. *J. Comput. System Sci*, 37 :367–394, 1988.

[Sch98] D.A. Schmidt. Data flow analysis is model checking of abstract interpretation. In *Proceedings of POPL'98*, pages 38–48. ACM Press, 1998.

[Sch02] S. Schwoon. Moped - a model-checker for pushdown systems. http ://www7.in.tum.de/∼schwoon/moped, 2002.

[SG89] Z. Stadler and O. Grumberg. Network grammars, communication behaviors and automatic verification. In *Proc. Workshop on Automatic Verification Methods for Finite State Systems*, pages 151–165, 1989.

[SOR93] N. Shankar, S. Owre, and J.M. Rushby. The PVS proof checker : A reference manual (draft). Technical report, Comp. Sci. Laboratory, SRI International, Menlo Park, CA, 1993.

[SS98] D.A. Schmidt and B. Steffen. Program analysis as model checking of abstract interpretations. In *Proceedings of SAS'98*, pages 351–380. LNCS 1503, 1998.

[SS00] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. In G. Smolka, editor, *Programming Languages and Systems (ESOP'2000)*, volume 1782 of *Electronic Notes in Theoretical Computer Science*, pages 351–365. Springer-Verlag, 2000.

[SS01] H. Seidl and B. Steffen. Constraint-based inter-procedural analysis of parallel programs. *Nordic Journal of Computing*, 7(4) :371 – 400, 2001. special Issue for ESOP'2000.

[Ste91] B. Steffen. Data fow analysis as model checking. In *TACS'91*, pages 346–364. LNCS 526, 1991.

[Sti96] C. Stirling. Modal and temporal logic for processes. *Logics for Concurrency*, 1043 :149–237, 1996.

[Tar55] A. Tarski. A lattice-theoretical fixpoint theorem and its applications. In *Pacific J. Math.*, number 5, pages 285–309, 1955.

[Tay83] R.N. Taylor. A general-purpose algorithm for analyzing concurrent programs. *Communications of the ACM*, 26 :362–376, 1983.

[Tho82] W. Thomas. Classifying regular events in symbolic logic. *Journal of Computer and System Sciences*, 25 :360–376, 1982.

[TKS00] T. Takai, Y. Kaji, and H. Seki. Right-linear finite path overlapping term rewriting systems effectively preserve recognizability. In *RTA 2000*, volume 1833 of *LNCS*, pages 246–260, 2000.

[Tou01] T. Touili. Widening Techniques for Regular Model Checking. In *1st vepas workshop*. Volume 50 of Electronic Notes in Theoretical Computer Science, 2001.

[Val91] A. Valmari. Stubborn sets for reduced state space generation. In *Advances in Petri Nets*, volume 483 of *Lecture Notes in Computer Science*, pages 491–515. Springer Verlag, 1991.

[VW86] M.Y. Vardi and P. Wolper. An Automata-Theoretic Approach to Automatic Program Verification. In *Proc. 2nd Symp. on Logic in Computer Science (LICS'86)*. IEEE, 1986.

[Wal96] Igor Walukiewicz. Pushdown processes : Games and model checking. In *8th Intern. Conf. on Computer Aided Verification (CAV'96)*, pages 62–74. LNCS, Springer-Verlag, 1996.

[WL89] P. Wolper and V. Lovinfosse. Verifying Properties of Large Sets of Processes with Network Invariants. In *Workshop on Automatic Verification Methods for Finite State Systems*. LNCS 407, 1989.

[Wol86] P. Wolper. Expressing interesting properties of programs in propositional temporal logic. In *Proc. of the 13th ACM Symposium on Principles of Programming Languages*, pages 184–193, 1986.

[Wol93] P. Wolper. Temporal logic can be more expressive. *Formal Methods in System Design*, 2(2) :149–164, 1993.

[Yah01] Eran Yahav. Verifying safety properties of concurrent Java programs using 3-valued logic. *ACM SIGPLAN Notices*, 36(3) :27–40, 2001.

[Yov98] S. Yovine. Model-checking timed automata. *Lecture Notes in Computer Science*, 1494 :114–153, 1998.

Index