Contribution à la vérification de systèmes avec structures de données complexes à l'aide d'automates

Mémoire d'habilitation à diriger des recherches

Peter Habermehl

7 décembre 2009

Table des matières

Avant-propos **5**

1 Introduction **9**

2 Regular model-checking **13**
2.1 Préliminaires 14
2.2 Extension aux arbres 18
2.3 Approches de vérification 19
2.3.1 Regular model-checking et abstractions 19
2.3.2 Apprentissage 27
2.4 Regular model-checking avec NFA 35
2.4.1 Les antichaînes 35
2.4.2 Apprentissage des automates non-déterministes (NL*) . 36
2.5 Perspectives 50

3 Programmes avec pointeurs **51**
3.1 Programmes avec listes 51
3.1.1 Définitions syntaxiques 51
3.1.2 Appliquer le regular model-checking 53
3.1.3 Traduire vers des automates à compteurs 60
3.2 Programmes avec structures de données arborescentes 62
3.2.1 Programmes 62
3.2.2 Les propriétés 64
3.2.3 Appliquer le regular model-checking sur les arbres avec abstraction 66
3.2.4 La terminaison 69
3.3 Travaux connexes 69
3.4 Perspectives 70

4 Arbres équilibrés **71**
4.1 Une méthodologie de vérification basée sur les TASC 73
4.2 Les automates d'arbre avec contraintes de taille 77
4.3 Propriétés de fermeture et décidabilité de TASC 79
4.4 La sémantique des opérations 81
4.4.1 TASC restreints 82
4.4.2 Représentation des configurations de mémoire 82
4.4.3 Calculer l'effet d'une rotation d'arbre 83
4.4.4 Les autres opérations 85
4.5 Une étude de cas : L'insertion dans un arbre rouge et noir . . 87
4.6 Perspectives 91

5 Programmes avec tableaux **93**
5.1 Préliminaires 93
5.2 La logique LIA 94
5.2.1 Travaux connexes 96
5.2.2 Automates à compteurs 97
5.2.3 Définition de LIA 100
5.2.4 Décidabilité du problème de satisfaisabilité 103
5.3 Vérification automatique de programme avec tableaux 115
5.3.1 Travaux connexes 116
5.3.2 Préliminaires 117
5.3.3 Automates à compteurs pour reconnaître les configurations et les transitions 118
5.3.4 La logique SIL 122
5.3.5 Les programmes qui manipulent les tableaux 125
5.3.6 Des boucles aux automates à compteurs 126
5.3.7 Exemples 129
5.4 Perspectives 129

6 Autres travaux **131**

7 Conclusion **133**

Bibliographie **135**

Avant-propos

Ce mémoire est un résumé de la plupart de mes travaux scientifiques des cinq dernières années. Il contient quatre chapitres principaux : Le chapitre 2 résume mes travaux dans le domaine du “regular model-checking”. Le chapitre 3 traite de la vérification de programmes avec pointeurs. Le chapitre 4 parle de mes travaux sur la vérification d'arbres équilibrés et le chapitre 5 de la vérification de programmes avec tableaux. Chaque chapitre est basé sur un certain nombre d'articles que j'ai publié avec des coauteurs durant les cinq dernières années.

Le chapitre 2 est basé sur les articles suivants :

- B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1004–1009, Pasadena, CA, USA, July 2009. AAAI Press.
- A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA*, volume 5148 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2008.
- A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In *Proceedings of the 7th International Workshop on Verification of Infinite State Systems (INFINITY'05)*, volume 149 of *Electronic Notes in Theoretical Computer Science*, pages 37–48, San Francisco, CA, USA, 2006. Elsevier Science Publishers.
- A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. 16th Int. Conf. Computer Aided Verification (CAV 2004), Boston, MA, USA, July 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer Verlag, 2004.
- P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In J. Bradfield and F. Moller, editors, *Procee-*

dings of the 6th International Workshop on Verification of Infinite State Systems (INFINITY'04), volume 138 of *Electronic Notes in Theoretical Computer Science*, pages 21–36, London, UK, Dec. 2005. Elsevier Science Publishers.

Le chapitre 3 est basé sur les articles suivants. Une partie de ces résultats se trouve aussi dans la thèse de Pierre Moro que j'ai co-encadré avec Ahmed Bouajjani.

- A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531, Seattle, Washington, USA, 2006. Springer.
- A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. 11th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, Scotland, UK, Apr. 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 13–29. Springer Verlag, 2005.
- A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proceedings of the 13th International Symposium Static Analysis (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70, Seoul, Korea, 2006. Springer.
- P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*, pages 145–161, Tokyo, Japan, 2007. Springer.

Le chapitre 4 est basé sur l'article suivant :

- P. Habermehl, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. In *Proceedings of the 12th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 350–364, Vienna, Austria, 2006. Springer. version étendue accepté à Acta Informatica.

Le chapitre 5 est basé sur les articles suivants :

- M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *Proceedings of CAV*

09, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer Verlag, 2009.
– P. Habermehl, R. Iosif, and T. Vojnar. A Logic of Singly Indexed Arrays. In *Proc. of LPAR'08*, volume 5330 of *LNAI*. Springer Verlag, 2008.
– P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays ? In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'08)*, volume 4962 of *Lecture Notes in Computer Science*, pages 474–489. Springer Verlag, 2008.

Remerciements Je tiens d'abord à remercier tous mes coauteurs de ces dernières années : Mohamed Faouzi Atig, Benedikt Bollig, Marius Bozga, Manuela-Lidia Grindei, Lukás Holík, Yan Jurski, Filip Konecný, Pierre Moro, Anca Muscholl, Adam Rogalewicz, Thomas Schwentick, Helmut Seidl, Mihaela Sighireanu, et Tayssir Touili. Je remercie en particulier Ahmed Bouajjani, Tomás Vojnar et Radu Iosif. C'est en grande partie grâce à eux que j'ai pu obtenir la plupart de mes résultats. Je tiens à remercier également Philippe Schnoebelen et l'équipe dirigeante du LSV qui m'ont permis de passer presque deux ans en délégation INRIA au LSV. Cette période fut très enrichissante et productive. Je suis très reconnaissant aux institutions qui m'ont soutenu pendant ces années, notamment l'ANR à travers le projet RNTL Averiles dont je suis le responsable, le CNRS, l'INRIA et l'Université Paris Diderot (Paris 7). Je remercie également tous mes collègues et toute l'équipe du LIAFA.

Chapitre 1

Introduction

Les systèmes informatiques sont devenus omniprésents dans la vie moderne. Les logiciels notamment prennent une place de plus en plus importante dans des domaines telles que la télécommunication, etc. La vérification des logiciels (ou programmes) est d'une importance cruciale car leur défaillance peut entraîner des conséquences graves. La vérification de programmes occupe une place importante en informatique depuis le début de la discipline. Elle traite le problème de montrer la correction d'un programme (l'absence d'erreurs) par rapport à une spécification. Le problème est en général indécidable. Il existe néanmoins des méthodes générales basées sur les logiques (par exemple l'approche de Hoare) permettant de spécifier des propriétés de programmes ainsi que de formuler des obligations de preuves pour les vérifier. Cette méthode requiert en général une intervention humaine. Pour les systèmes avec un nombre fini de configurations comme par exemple les circuits ou certains protocoles de télécommunication la méthode dite du model-checking a connu un grand succès depuis les années 80. C'est une méthode entièrement automatique qui nécessite aucune intervention humaine. Elle permet de vérifier des propriétés de logiques temporelles.

Les langages de programmation modernes permettent l'utilisation de structures de données complexes (composés de type de données simples comme les entiers), telles que les tableaux ou les listes ou autres structures utilisant une mémoire dynamique. Ces structures sont intrinsèquement infinies. Dans ce mémoire nous nous intéressons à la vérification *automatique* de programmes utilisant ces structures de données complexes. Nous voulons obtenir des méthodes automatiques avec l'intervention nécessaire de l'utilisateur la plus parcimonieuse possible. Puisque le problème de vérification est en

général indécidable, on ne peut évidemment pas obtenir une méthode qui est en général à 100% automatique. Néanmoins, nous nous intéressons à automatiser le plus possible. Nous proposons d'une part des méthodes permettant d'automatiser une partie du raisonnement nécessaire pour la vérification et d'autre part nous donnons des méthodes qui fonctionnent automatiquement sur une grande partie des programmes rencontrés en pratique.

Pour vérifier automatiquement un programme, il faut d'abord pouvoir exprimer les propriétés auxquelles on s'intéresse. Pour cela, nous définissons entre autres une nouvelle logique expressive pour les tableaux d'entiers dans le chapitre 5. Nous montrons que cette logique est décidable permettant une vérification automatique de triplets de Hoare de la forme $\{P\}\ \mathbf{C}\ \{Q\}$, où C est une instruction d'un programme manipulant les tableaux, P une précondition et Q une postcondition. Nous montrons également pour une autre classe de programmes manipulant des arbres équilibrés comment décider automatiquement la validité de triplets de Hoare (voir le chapitre 4). Dans ce cas, les propriétés P et C ne sont pas décrites dans une logique mais par des automates d'arbres étendus permettant de raisonner sur des contraintes d'équilibre entre les tailles de sous-arbres.

Ces deux méthodes permettent d'automatiser la vérification mais supposent que l'utilisateur fournisse les invariants de boucles. Nous pouvons aller plus loin dans l'automatisation pour certains types de programmes en générant automatiquement des invariants. Ceci est notamment possible dans le cadre du *regular model-checking*. C'est un cadre général de vérification de systèmes avec un nombre infini de configurations. Il est basé sur la représentation d'ensemble de configurations comme des langages de mots/arbres réguliers et de transitions comme des transducteurs réguliers. Dans le chapitre 2 nous montrons plusieurs techniques de génération d'invariants dans ce cadre basées sur les abstractions d'automates et l'inférence (l'apprentissage) de langages réguliers. Le regular model-checking s'applique à tous les systèmes dont les configurations peuvent être codées comme des mots ou des arbres. Nous montrons un exemple de tels systèmes dans le chapitre 3 où nous nous intéressons aux programmes avec pointeurs utilisant une mémoire dynamique. Nous introduisons un codage des configurations de programmes avec mémoire dynamique comme des mots (pour les listes) et comme des arbres (pour les structures plus générales) qui permet l'application du regular model-checking. Nous montrons également une traduction des programmes avec listes vers des automates à compteurs, ce qui permet d'utiliser toutes les techniques de vérification de ces modèles pour les programmes avec listes, no-

tamment cela donne une méthode intéressante pour montrer la terminaison de programmes.

Nous remarquons que toutes les approches de vérification différentes que nous présentons sont basées sur l'utilisation des automates. Il s'agit des automates finis classiques sur les mots et les arbres ainsi que des extensions de ces automates avec des données comme des entiers. D'une part les automates sont utilisés pour décrire les programmes et pour obtenir des modèles des programmes (comme les automates à compteurs pour les programmes avec listes). D'autre part nous les utilisons comme des structures de représentation de *configurations* de programmes. Par exemple, les configurations de programmes travaillant sur des tableaux d'entiers sont naturellement représentées par des calculs d'automates à compteurs. Nous utilisons également les automates comme outil pour montrer la décidabilité de logiques.

Plan Nous présentons d'abord dans le chapitre 2 nos résultats obtenus dans le cadre du regular model-checking et ses extensions. Ensuite, nous donnons dans le chapitre 3 plusieurs méthodes de vérification de programmes avec pointeurs. Dans le chapitre 4 nous détaillons nos résultats pour les arbres équilibres. Dans le chapitre 5 nous résumons nos résultats sur les programmes avec tableaux et dans le chapitre 6 nous esquissons quelques autres travaux avant de conclure.

Dans chaque chapitre nous discutons les travaux connexes.

Chapitre 2

Regular model-checking

Dans ce chapitre nous résumons nos travaux dans le domaine du *regular model-checking* [78, 31]. Nous donnons d'abord une introduction au regular model-checking avant de détailler nos contributions. Le regular model-checking est une approche pour la vérification de systèmes avec un nombre infini de configurations. Cette approche est basée sur l'observation que les automates finis définissent des langages infinis qui peuvent être considérés comme des configurations d'un système. De la même façon les transducteurs finis peuvent être considérés comme des transitions d'un système. Cette approche est très générale car elle permet de représenter la relation de transition d'une machine de Turing. Tous les problèmes "intéressants" de vérification sont donc indécidables dans cette approche. Nous allons illustrer le regular model-checking avec un petit exemple.

Exemple 2.1 *Nous modélisons un très simple protocole de passage de jeton. Le protocole est censé passer un jeton de gauche à droite le long d'une suite ordonnée de nœuds représentant des processus. Leur nombre n'est pas fixé. Le protocole doit satisfaire la propriété qu'il y aura toujours exactement un jeton. Chaque processus a deux états (N, qui signifie qu'il n'a pas le jeton et T, qui signifie qu'il a le jeton). Initialement le premier processus à gauche a le jeton. Une configuration d'un tel système peut être représenté naturellement par un mot sur le vocabulaire $\Sigma = \{N, T\}$ et un ensemble de configurations (possiblement infinis) par un langage $L \subseteq \Sigma^*$. Par exemple les configurations initiales sont données par l'automate de la figure 2.1 et les configurations mauvaises sont données par l'automate de la figure 2.2.*

La relation de transition peut maintenant être donnée (voir figure 2.3)

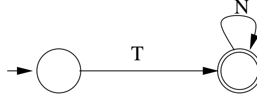


FIG. 2.1 – Les configurations initiales

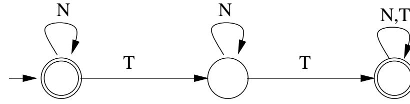


FIG. 2.2 – Les mauvaises configurations

comme un transducteur sur Σ (un automate sur $\Sigma \times \Sigma$).

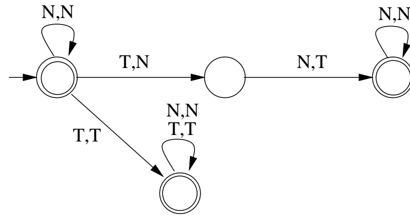


FIG. 2.3 – La relation de transition

Il est aussi possible de définir un transducteur représentant les mauvaises relations du système (le jeton ne doit pas être passé de droite à gauche, voir figure 2.4).

Les configurations atteignables à partir des configurations initiales sont données dans la figure 2.5 et la relation d'atteignabilité est donnée dans la figure 2.6. En général, ces ensembles ne sont pas réguliers.

Le regular model-checking consiste à (essayer de) calculer la relation d'atteignabilité ou les configurations atteignables. Nous donnons par la suite les définitions formelles.

2.1 Préliminaires

Un ensemble fini Σ de lettres est appelé *alphabet*. Des séquences finis de lettres sont des éléments de Σ^*, appelées *mots*. Des ensembles de mots sont

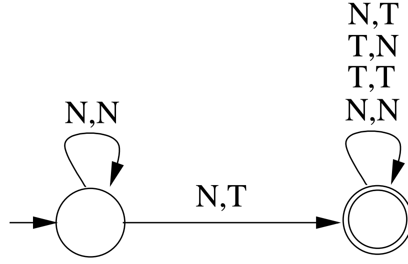


Fig. 2.4 – La mauvaise relation de transition

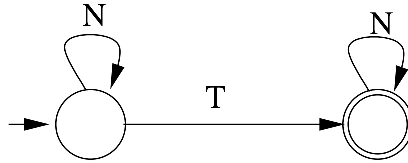


Fig. 2.5 – Les configurations atteignables

appelés *langages* et sont donc des sous-ensembles de Σ^*. Nous écrivons 2^X pour l'ensemble des parties d'un ensemble X. Pour un mot $w \in \Sigma^*$ nous écrivons $Pref(w)$ (resp. $Suff(w)$) pour l'ensemble de tous ses préfixes (resp. suffixes). Ces ensembles contiennent w et le mot vide ϵ.

Definition 2.2 *Un* automate fini *non-déterministe (NFA) est un quintuplet $M = (Q, \Sigma, \delta, Q_0, F)$ où Q est un ensemble fini d'états, Σ un alphabet fini, $\delta : Q \times \Sigma \to 2^Q$ une fonction (partielle) de transition, $Q_0 \subseteq Q$ un ensemble d'états initiaux et $F \subseteq Q$ un ensemble d'états finaux. M est* déterministe *si $|Q_0| = 1$ et $|\delta(q, a)| = 1$ pour tout $q \in Q$ et $a \in \Sigma$. Nous appelons un automate fini déterministe DFA.*

La fonction de transition δ d'un NFA est étendue comme d'habitude vers $\bar{\delta} : Q \times \Sigma^* \to 2^Q$ par $\bar{\delta}(q, \epsilon) = \{q\}$ et $\bar{\delta}(q, aw) = \bigcup_{q' \in \delta(q,a)} \bar{\delta}(q', w)$, et vers des ensemble d'états $Q' \subseteq Q$ par $\hat{\delta}(Q', w) = \bigcup_{q \in Q'} \bar{\delta}(q, w)$. Pour simplifier la notation, nous utilisons δ aussi pour $\bar{\delta}$ et $\hat{\delta}$.

Le langage reconnu par M à partir d'un état $q \in Q$ est défini par $L(M, q) = \{w \in \Sigma^* \mid \delta(q, w) \cap F \neq \emptyset\}$. Quand M est donné par le contexte, nous écrivons aussi L_q pour $L(M, q)$. Le langage accepté par M, $L(M)$ est

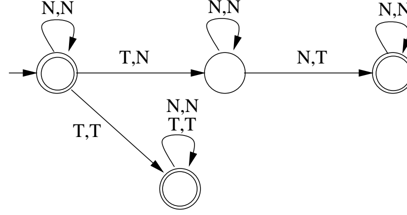


FIG. 2.6 – La relation d'atteignabilité

égal à $\cup_{q_0 \in Q_0} L(M, q_0)$. Deux automates sont appelés *équivalents* s'ils acceptent le même langage. Un ensemble $L \subseteq \Sigma^*$ est *régulier* ss'il existe un automate fini M tel que $L = L(M)$. Nous définissons aussi les langages de mots jusqu'à une certaine longueur : $L^{\leq n} = \{w \in L \mid |w| \leq n\}$, $L^{\leq n}(M, q) = \{w \in L(M, q) \mid |w| \leq n\}$, et $\Sigma^{\leq n} = \{w \in \Sigma^* \mid |w| \leq n\}$.

Un NFA est appelé *minimal* s'il n'y a pas de NFA équivalent avec moins d'états. De la même façon, un DFA est appelé *minimal* s'il n'y a pas de DFA équivalent avec moins d'états. Contrairement aux NFA, un DFA a toujours un automate équivalent minimal unique :

Theorème 2.3 (Myhill-Nerode) *Pour chaque langage régulier L, il existe un automate unique (modulo isomorphisme) M avec $L(M) = L$.*

Nous définissons la *profondeur d'un automate* $M = (Q, \Sigma, \delta, Q_0, F)$ que nous écrivons d_M pour la longueur maximale des chemins les plus courts qui mènent vers les états de M à partir d'un état initial, c.-à-d. $d_M = max_{q \in Q, q_0 \in Q_0}\ min_{w \in \Sigma^* \wedge q \in \delta(q_0, w)} |w|$. Deux états $q, q' \in Q$ de M sont appelés *k-indistinguables*, noté par $q \equiv_k q'$, si $L^{\leq k}(M, q) = L^{\leq k}(M, q')$. Nous définissons ensuite le *dégré de distinguabilité* ρ_M de M comme le plus petit k tel que toutes les pairs d'états q, q' de M soient k-distinguables, c.-à-d. $q \not\equiv_k q'$.

Soit Σ un alphabet fini et $\Sigma_\epsilon = \Sigma \cup \{\epsilon\}$. Un *transducteur fini* sur Σ est un quintuplet $\tau = (Q, \Sigma_\epsilon \times \Sigma_\epsilon, \delta, Q_0, F)$ où Q est un ensemble fini d'états, $\delta : Q \times \Sigma_\epsilon \times \Sigma_\epsilon \to 2^Q$ une fonction (partielle) de transition, $Q_0 \subseteq Q$ un ensemble d'état initiaux, et $F \subseteq Q$ un ensemble d'états finaux. Nous appelons un transducteur fini un transducteur *préservant la longueur* si δ n'est pas défini pour des entrées ϵ (c.est-à-dire si sa relation de transition ne contient pas ϵ [1]).

[1]Des transducteurs dont les transitions contiennent des ϵ peuvent aussi définir des relation qui lient uniquement des mots de même taille. Ces transducteurs peuvent toujours

La fonction de transition δ d'un transducteur est étendue vers $\bar{\delta} : Q \times \Sigma^* \times \Sigma^* \to 2^Q$ par $\bar{\delta}(q, \epsilon, \epsilon) = \{q\}$ et $\bar{\delta}(q, aw, bu) = \bigcup_{q' \in \delta(q,a,b)} \bar{\delta}(q', w, u)$ (pour $a, b \in \Sigma_\epsilon$), et vers des ensemble d'états $Q' \subseteq Q$ par $\hat{\delta}(Q', w, u) = \bigcup_{q \in Q'} \bar{\delta}(q, w, u)$. Pour simplifier la notation, nous utilisons δ aussi pour $\bar{\delta}$ et $\hat{\delta}$.

Par abus de notation nous identifions un transducteur τ avec la relation $\{(w, u) \mid \exists q_0 \in Q_0, q' \in F : q' \in \delta(q_0, w, u)\}$. Nous appelons une relation *régulière* si elle peut être identifiée avec un transducteur. Pour un ensemble $L \subseteq \Sigma^*$ et une relation $R \subseteq \Sigma^* \times \Sigma^*$, nous notons par $R(L)$ l'ensemble $\{w \in \Sigma^* \mid \exists w' \in L : (w', w) \in R\}$. Soit $id \subseteq \Sigma^* \times \Sigma^*$ la relation d'identité et $\circ$ la composition de relations. Nous définissons récursivement les relations $\tau^0 = id$, $\tau^{i+1} = \tau \circ \tau^i$, et $\tau^* = \cup_{i=0}^{\infty} \tau^i$. Dans ce qui suit, nous supposons $id \subseteq \tau$. Cela implique $\tau^i \subseteq \tau^{i+1}$ pour tout $i \geq 0$.

Le regular model-checking peut être utilisé pour vérifier des propriétés de sûreté ainsi que des propriétés de vivacité. Nous nous intéressons par la suite aux propriétés d'atteignabilité. En effet, des problèmes de vérification complexes peuvent souvent se ramener à des problèmes d'atteignabilité.

Pour vérifier les propriétés d'atteignabilité dans le cadre du regular model-checking, il y a principalement deux approches. Dans la première approche la relation de transition d'un système est donnée comme un transducteur τ sur l'alphabet Σ, l'ensemble régulier des configurations initiales $Init \subseteq \Sigma^*$ est décrit par un automate fini et un ensemble de “mauvaises” configurations $Bad \subseteq \Sigma^*$ est donné par un autre automate fini. Nous essayons ensuite de calculer l'ensemble des configurations atteignables $\tau^*(Init) \subseteq \Sigma^*$ (ou une sur-approximation) et de tester si $\tau^*(Init) \cap Bad = \emptyset$.

Problème 2.4 *Étant donnés un alphabet Σ, deux ensembles réguliers $Init \subseteq \Sigma^*$ et $Bad \subseteq \Sigma^*$ et un transducteur fini τ sur Σ, est-ce que $\tau^*(Init) \cap Bad = \emptyset$?*

Une deuxième approche consiste à essayer de calculer τ^*, c.-à-d. la fermeture réflexive et transitive de la relation de transition donnée par le transducteur fini τ. Ensuite, τ^* peut être utilisé pour résoudre le problème 2.4. Alternativement, nous pouvons aussi décrire les mauvais comportements par un transducteur τ_{Bad} qui exprime le fait qu'il est “mauvais” de pouvoir aller de

---

être transformés vers des transducteurs qui préservent la longueur conformément à notre définition

certaines configurations vers d'autres. Ensuite nous testons si $\tau^* \cap \tau_{Bad} = \emptyset$. Cela est en général uniquement possible si les transducteurs préservent la longueur.

Problème 2.5 *Étant donnés deux transducteurs sur Σ qui préservent la longueur τ et τ_{Bad}, est-ce que $\tau^* \cap \tau_{Bad} = \emptyset$?*

Ces deux approches ont leurs avantages et leurs inconvénients. Calculer τ^* est souvent plus difficile que calculer juste $\tau^*(Init)$. En effet, il y a des cas où $\tau^*(Init)$ est régulier mais pas τ^*. D'autre part, utiliser τ^* peut permettre de vérifier des propriétés qui sont plus difficiles ou impossibles à vérifier (comme par exemple les correspondances entrée-sortie entre des éléments particuliers de structures de données non bornées telle que les listes ou les files).

Le regular model-checking est une méthode générale. Nous étudions dans la suite des méthodes pour résoudre les problèmes 2.4 et 2.5. Pour pouvoir appliquer le regular model-checking dans un domaine d'application précis, il faut trouver un codage des configurations comme des mots et un codage de la relation de transition comme un transducteur. Nous présentons une application particulière aux programmes avec listes dans la section 3.1.2.

2.2 Extension aux arbres

Pour traiter d'autres structures que des structures linéaires, le *regular tree model-checking* a été introduit [78, 32, 4, 111, 5]. Les configurations ne sont pas de mots mais des arbres et les automates d'arbre sont utilisés pour représenter des ensembles de configurations. Ensuite, les transducteurs d'arbres modélisent les transitions. Comme dans le cas des mots, plusieurs approches d'analyse d'atteignabilité existent. Les structures arborescentes se trouvent couramment dans plusieurs contextes de modélisation et de vérification. Par exemple dans le cas des réseaux paramétrés d'arbres, des arbres étiquetés d'une hauteur arbitraire représentent les configurations d'un réseaux : chaque processus est un nœud d'un arbre et l'étiquette son état de contrôle. Les arbres apparaissent aussi naturellement comme par exemple les représentations de configurations d'un programme récursif multi-tâche [55], comme une représentation de structures du tas de mémoire d'un programme [81], ou pour représenter des données semi-structurées comme les documents XML [37]. Nous donnons dans la section 3.2 une autre application : les programmes avec structures de données arborescentes.

2.3 Approches de vérification

Pour résoudre les problèmes 2.4 et 2.5 plusieurs méthodes de calculs de $\tau^*(Init)$ resp. τ^* ont été proposées. Dans [76, 49, 3, 19] des méthodes de calculs de fermeture transitive de transducteurs sont proposées. Ces méthodes sont appelées *méthodes d'accélération*. Elles sont basées sur le calcul itératif de τ, τ^2, τ^3, etc. et l'identification de formes qui se répètent dans la structure de ces transducteurs permettant de déduire la limite. Des méthodes similaires ont été proposés pour le calcul de $\tau^*(Init)$ [31, 116]. Il y a des classes de transducteurs où certaines méthodes terminent toujours (comme par exemple les systèmes à *bounded local depth* dans [76]).

Nous introduisons deux autres méthodes de calculs de $\tau^*(Init)$ resp. τ^*. L'une est basée sur l'abstraction et l'autre est basée sur l'apprentissage de langages réguliers.

2.3.1 Regular model-checking et abstractions

Un problème crucial dans le regular model-checking est l'explosion des nombres d'états des automates (ou des transducteurs) qui représentent les ensembles de configurations. Une des sources de ce problème est que les techniques essaient de calculer l'ensemble d'atteignabilité exact indépendamment de la propriété à vérifier.

Il est cependant souvent suffisant de calculer uniquement une sur-approximation de l'ensemble des configurations atteignables qui est suffisamment précise pour vérifier la propriété donnée. Cette façon de faire est utilisée avec succès en dehors du domaine du regular model-checking pour des espaces de configurations très larges ou infinis. Le paradigme utilisé s'appelle *abstraire-tester-raffiner* [72, 109, 41, 50] et est implémenté par exemple dans des outils pour la vérification de logiciels comme Slam [14], Magic [40], ou Blast [73]. Tous ces outils utilisent la méthode de l'*abstraction par prédicats* [110], où un ensemble fini de prédicats est utilisé pour abstraire un système concret C vers un système abstrait A en considérant équivalents les configurations de C qui satisfont les même prédicats. Si une propriété est satisfaite en A, elle est garantie d'être satisfait en C aussi. Si un contre-exemple (typiquement une exécution qui mène à une erreur) est trouvé dans A, alors on peut tester si c'est aussi un contre-exemple pour C. Si ce n'est pas le cas, ce contre-exemple *faux* peut être utilisé pour *raffiner* l'abstraction tel que le nouveau système abstrait A' n'admet plus ce contre-exemple. De cette façon on peut

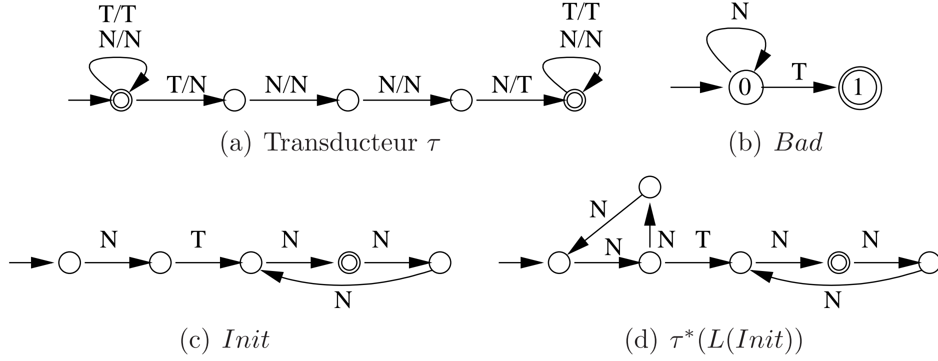
construire des abstractions de plus en plus fines jusqu'à ce qu'une précision suffisante est atteinte et la propriété est vérifiée ou un vrai contre-exemple est trouvé.

Dans ce qui suit, nous proposons d'appliquer le paradigme du *abstraire-tester-raffiner* au regular model-checking. Au lieu des techniques d'accélération précise, nous utilisons des calculs abstraits de point fixe dans un domaine *fini* d'automate. Le calcul abstrait termine toujours et fournit des sur-approximations de l'ensemble d'atteignabilité (ou de la relation d'atteignabilité). Pour cela, nous définissons des techniques qui associent à chaque automate M un automate M' d'un domaine fini tel que M' reconnaît un sur-ensemble du langage de M. Dans le cas où la sur-approximation est trop grossière et un faux contre-exemple est détecté, nous donnons des méthodes permettant le raffinement de l'abstraction tel que le nouveau calcul ne rencontre plus le même contre-exemple.

Nous proposons principalement deux techniques pour abstraire les automates. Elles tiennent compte de la structure de l'automate et sont basées sur la fusion de leurs états par rapport à une relation d'équivalence. La première technique est inspirée par l'abstraction par prédicats. Contrairement à l'abstraction par prédicats classique nous associons des prédicats avec les états d'automates qui représentent des configurations et pas avec les configurations elles-mêmes. Une abstraction est définie par un ensemble de *langages réguliers de prédicats* L_p. Nous considérons qu'un état q d'un automate M "satisfait" un langage prédicat L_p, si l'intersection de L_p avec le langage $L(M, q)$ accepté à partir de l'état q n'est pas vide. Deux états sont alors équivalents, s'ils satisfont les mêmes prédicats. La deuxième technique d'abstraction est obtenue en considérant que deux états sont équivalents si leurs langages de mots jusqu'à une certaine longueur fixé sont égaux. Pour les deux méthodes d'abstraction nous donnons des techniques de raffinement qui nous permettent d'éliminer de faux contre-exemples.

Pour pouvoir illustrer l'approche du regular model-checking avec abstraction nous considérons un exemple un peu plus compliqué.

Exemple 2.6 *Comme dans l'exemple 2.1 nous modélisons un protocole de passage de jeton (avec alphabet $\Sigma = \{T, N\}$). Le transducteur τ dans la figure 2.7 modélise la relation de transition. Chaque processus peut passer le jeton à son troisième voisin de gauche. Dans les configurations initiales décrites par l'automate $Init$, le deuxième processus a le jeton, et le nombre de processus est divisible par trois. Nous voulons montrer qu'il n'est pas possible*



(a) Transducteur τ (b) *Bad*

(c) *Init* (d) $\tau^*(L(Init))$

FIG. 2.7 – Un transducteur modélisant un simple protocole de passage de jeton, les configurations initiales, mauvaises et atteignables

d'atteindre une configuration où le dernier processus a le jeton. Cet ensemble de configurations est décrit par l'automate Bad.

Dans ce qui suit, le problème de vérification est de trouver une sur-approximation régulière $L \supseteq \tau^*(L(Init))$ tel que $L \cap L(Bad) = \emptyset$.

2.3.1.1 L'abstraction des automates

Soit Σ un alphabet et $\mathbb{M}_\Sigma$ l'ensemble de tous les automates sur Σ. Une *fonction d'abstraction d'automate* α est une fonction qui associe à chaque automate M sur Σ un automate $\alpha(M)$ de sorte que son langage soit une sur-approximation du langage de M, c'est-à-dire pour un $\mathbb{A}_\Sigma \subseteq \mathbb{M}_\Sigma$, $\alpha : \mathbb{M}_\Sigma \to \mathbb{A}_\Sigma$ tel que $\forall M \in \mathbb{M}_\Sigma : L(M) \subseteq L(\alpha(M))$. Nous appelons α *finie* ssi son codomaine $\mathbb{A}_\Sigma$ est fini.

Nous introduisons la *fonction de transition abstraite* τ_α pour une relation de transition donnée comme un transducteur τ sur Σ et une fonction d'abstraction d'automate α comme suit : Pour chaque automate $M \in \mathbb{M}_\Sigma$, $\tau_\alpha(M) = \alpha(\hat{\tau}(M))$ où $\hat{\tau}(M)$ est l'automate minimal déterministe de $\tau(L(M))$. Ainsi, nous pouvons calculer itérativement la séquence $(\tau_\alpha^i(M))_{i\geq 0}$. Si nous supposons $id \subseteq \tau$, il est clair que si α est fini, alors il existe un $k \geq 0$ tel que $\tau_\alpha^{k+1}(M) = \tau_\alpha^k(M)$. La définition de α implique $L(\tau_\alpha^k(M)) \supseteq \tau^*(L(M))$. Nous pouvons donc calculer dans un nombre fini de pas une sur-approximation de l'ensemble d'atteignabilité $\tau^*(L(M))$.

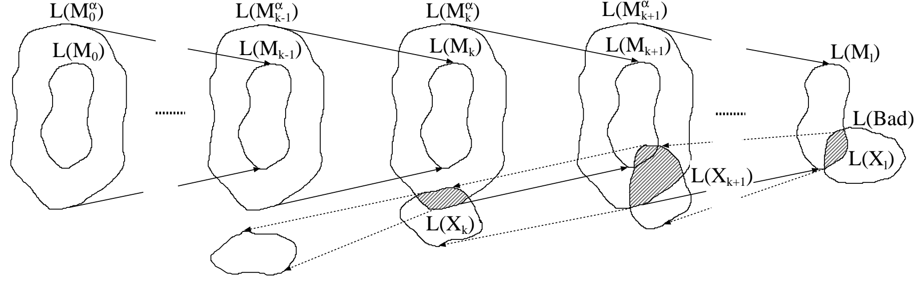


FIG. 2.8 – Un faux contre-exemple dans un calcul de point fixe abstrait régulier

2.3.1.2 Raffiner l'abstraction des automates

Nous appelons une fonction d'abstraction d'automate α' un *raffinement* de α ssi $\forall M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subseteq L(\alpha(M))$. De plus, nous appelons α' un *vrai raffinement* ss'il donne une plus petite sur-approximation dans au moins un cas, formellement ssi $\exists M \in \mathbb{M}_\Sigma : L(\alpha'(M)) \subsetneq L(\alpha(M))$.

Il est nécessaire de raffiner α dans une situation comme décrite dans la figure 2.8. Supposons que nous essayons de vérifier si aucune configuration d'un ensemble décrit par un automate *Bad* est atteignable à partir de l'ensemble de configurations initiales décrit par M_0. Nous supposons $L(M_0) \cap L(Bad) = \emptyset$, sinon la propriété à vérifier est déjà violée par les configurations initiales. Soient $M_0^\alpha = \alpha(M_0)$ et pour chaque $i > 0$, $M_i = \hat{\tau}(M_{i-1}^\alpha)$ et $M_i^\alpha = \alpha(M_i) = \tau_\alpha(M_{i-1}^\alpha)$. Il existe k et l $(0 \leq k < l)$ tel que

1. $\forall i : 0 \leq i < l : L(M_i) \cap L(Bad) = \emptyset$ et
2. $L(M_l) \cap L(Bad) = L(X_l) \neq \emptyset$

Nous définissons X_i comme l'automate déterministe qui accepte$\tau^{-1}(L(X_{i+1}))$ $\cap L(M_i^\alpha)$ pour tout i tel que $0 \leq i < l$. Alors $\forall i : k < i < l : L(X_i) \cap L(M_i) \neq \emptyset$ et $L(X_k) \cap L(M_k) = \emptyset$ bien que $L(X_k) \neq \emptyset$. Donc, soit $k = 0$ ou alors $L(X_{k-1}) = \emptyset$, et il est clair que nous avons détecté un *faux contre-exemple.*

Si aucun l avec $L(M_l) \cap L(Bad) \neq \emptyset$ est trouvé, le calcul atteint un point fixe et la propriété est prouvée. D'un autre côté, si $L(X_0) \cap L(M_0) \neq \emptyset$, alors la propriété est violée.

Le *faux contre-exemple peut être éliminé* en raffinant α vers α' tel que pour chaque automate M dont le langage est disjoint de $L(X_k)$, le langage de son α'-abstraction est disjoint de $L(X_k)$ aussi. Alors le même "faux" calcul

d'atteignabilité (c.-à-d. la même séquence de M_i et M_i^α) ne peut pas être répété car l'abstraction de M_k vers M_k^α est exclue. De plus, l'atteignabilité des mauvaises configurations est en général exclue, s'il n'y a pas d'autres raisons que la sur-approximation avec des sous-ensembles de $L(X_k)$.

Une façon d'élimination un peu plus faible consiste à raffiner α vers α' tel qu'au moins le langage de l'abstraction de M_k est disjoint de $L(X_k)$. Dans ce cas, il n'est pas exclu qu'un sous-ensemble de $L(X_k)$ va être utilisé plus tard pour une sur-approximation, mais une répétition d'exactement le même calcul est exclue. Le raffinement ainsi obtenu peut être plus grossier, ce qui peut amener à plus de raffinements et un calcul plus lent. D'un autre côté, le calcul peut terminer plus rapidement en atteignant plus vite le point fixe et utiliser moins de mémoire, car des ensembles de configurations moins structurés sont utilisés. Dans ce cas l'abstraction ne devient pas inutilement précis. Pour cela, on peut même utiliser des approches de raffinements qui garantissent uniquement que le faux contre-exemple va être exclu un jour (après un certain nombre de raffinement) ou pas du tout.

2.3.1.3 Abstraire des automates en fusionnant leurs états

Nous donnons maintenant plusieurs fonctions d'abstraction d'automates. Elles sont basées sur des schémas d'équivalence d'automates qui définissent pour chaque automate de $\mathbb{M}_\Sigma$ une relation d'équivalence sur leurs états. Un automate est ensuite abstrait en fusionnant tous les états équivalents. Formellement, un *schéma d'équivalence d'automates* $\mathbb{E}$ associe une relation d'équivalence $\sim_M^{\mathbb{E}} \subseteq Q \times Q$ à chaque automate $M = (Q, \Sigma, \delta, Q_0, F)$. Nous définissons la *fonction d'abstraction d'automates* $\alpha_{\mathbb{E}}$ basé sur $\mathbb{E}$ tel que $\forall M \in \mathbb{M}_\Sigma : \alpha_{\mathbb{E}}(M) = M/\sim_M^{\mathbb{E}}$. Nous appelons $\mathbb{E}$ *fini* ssi $\alpha_{\mathbb{E}}$ est fini. Nous *raffinons* $\alpha_{\mathbb{E}}$ en raffinant $\mathbb{E}$ tel que plus d'états sont distingués dans au moins un automate.

Les schémas d'équivalence d'automates présentés ci-dessous sont basés sur les deux principes suivants :

- La comparaison d'états par rapport à l'intersection de leur langages avec certains *langages de prédicats* (représentés par des automates de prédicats correspondant)
- La comparaison d'états par rapport à leur comportement jusqu'à une certaine longueur bornée.

Équivalence basée sur les langages de prédicats Soit $\mathcal{P}$ un ensemble fini d'*automates de prédicats* $\mathcal{P}$. Le schéma d'équivalence d'automates $\mathbb{F}_{\mathcal{P}}$ considère deux états d'un automate comme équivalents, si leur langages ont une *intersection non vide avec les mêmes prédicats* de $\mathcal{P}$. Formellement, pour un automate $M = (Q, \Sigma, \delta, Q_0, F)$, $\mathbb{F}_{\mathcal{P}}$ définit l'équivalence des états comme l'équivalence $\sim_M^{\mathcal{P}}$ tel que $\forall q_1, q_2 \in Q : q_1 \sim_M^{\mathcal{P}} q_2 \Leftrightarrow (\forall P \in \mathcal{P} : L(P) \cap L(M, q_1) \neq \emptyset \Leftrightarrow L(P) \cap L(M, q_2) \neq \emptyset)$.

Puisque $\mathcal{P}$ est fini et il y a un nombre fini de sous-ensembles de $\mathcal{P}$, $\mathbb{F}_{\mathcal{P}}$ est *fini*.

Pour l'exemple 2.6 (figure 2.7), nous prenons comme $\mathcal{P}$ les automates obtenus à partir de *Bad* en considérant un par un chaque état comme état initial. Alors, nous obtenons dans la figure 2.9(a) l'abstraction de *Init* de la figure 2.7(c). Cela est dû au fait que tous les états sauf l'état final de *Init* deviennent équivalents, car tous les langages des états de *Init* ont une intersection vide avec le langage réconnu à partir de l'état 0 de *Bad* et tous les états sauf l'état final ont une intersection vide avec le langage réconnu à partir de l'état 1 de *Bad*. Après avoir identifié les états équivalents et déterminisation et minimisation l'automate de la figure 2.9(a) est obtenu. L'intersection de $\hat{\tau}(\alpha(Init))$ avec les configurations mauvaises (voir figure 2.9(c)) n'est pas vide et l'abstraction doit être raffinée.

Le schéma $\mathbb{F}_{\mathcal{P}}$ peut être *raffiné en ajoutant des nouveaux prédicats* à l'ensemble $\mathcal{P}$. En particulier, nous pouvons étendre $\mathcal{P}$ par des automates correspondants à tous les états de X_k de la figure 2.8. Le théorème 2.7 montre que cela empêche les abstractions de langages disjoints avec $L(X_k)$, tel que $L(M_k)$, d'avoir une intersection non vide avec $L(X_k)$. Une répétition du même faux calcul est donc exclue.

Theorème 2.7 *Étant donnés deux automates fini* $M = (Q_M, \Sigma, \delta_M, q_0^M, F_M)$ *et* $X = (Q_X, \Sigma, \delta_X, q_0^X, F_X)$ *et un ensemble d'automates de prédicats* $\mathcal{P}$ *tels que* $\forall q_X \in Q_X : \exists P \in \mathcal{P} : L(X, q_X) = L(P)$*. Alors, si* $L(M) \cap L(X) = \emptyset$ *nous avons aussi* $L(\alpha_{\mathbb{F}_{\mathcal{P}}}(M)) \cap L(X) = \emptyset$

Dans notre exemple, nous raffinons l'abstraction en étendant $\mathcal{P}$ avec les automates qui représentent les langages d'états de X_0 de la figure 2.9(d). La figure 2.9(e) indique pour chaque état q de *Init*, les prédicats correspondants aux états de *Bad* et de X_0 dont leurs langages ont des intersections non vides avec le langage de q. Par exemple, en partant du 3ème état de *Init*, N est accepté tout comme à partir de 5 dans X_0. Les deux premiers états de *Init*

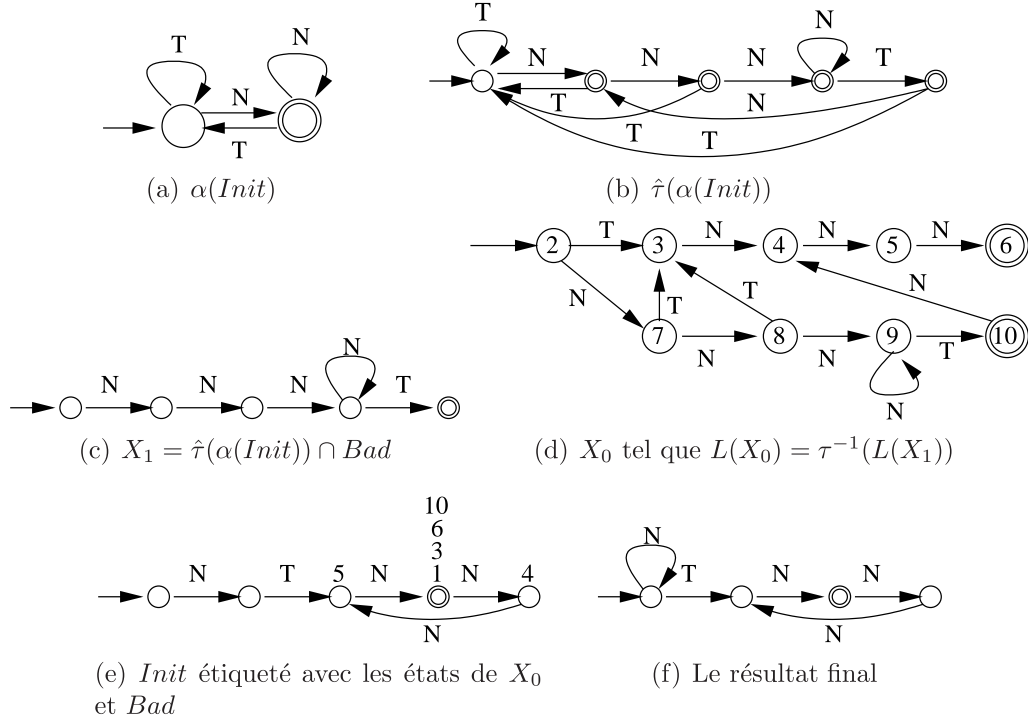


FIG. 2.9 – Un exemple utilisant l'abstraction basée sur les langages de prédicats

sont équivalents et fusionnés pour obtenir l'automate de la figure 2.9(f), qui est un point fixe montrant que la propriété est vérifiée. On remarque que c'est une sur-approximation de l'ensemble de configurations atteignables de la figure 2.7(d).

Le prix de raffiner $\mathbb{F}_{\mathcal{P}}$ en ajoutant des prédicats pour tous les états de X_k semble énorme, mais ce n'est pas le cas en pratique. Comme décrit en détail dans [29] nous pouvons exploiter le fait que les nouveaux prédicats viennent du même automate. En outre le raffinement peut être affaibli en considérant uniquement quelques états de X_k.

De la même façon on peut définir un schéma d'équivalence pour les langages d'état en arrière (pour les détails voir [29]).

Nous terminons cette section en remarquant que pour l'*ensemble initial de prédicats* $\mathcal{P}$ de $\mathbb{F}_{\mathcal{P}}$, nous pouvons utiliser par exemple l'automate décrivant les mauvaises configurations et/ou l'ensemble de configurations

initiales. Nous pouvons aussi utiliser le domaine ou le codomaine des transducteurs représentant les transitions particulières du système considéré (leur union donne la relation itérée τ). Le sens de ces dernières prédicats est similaire à l'utilisation des gardes dans l'abstraction par prédicats [16].

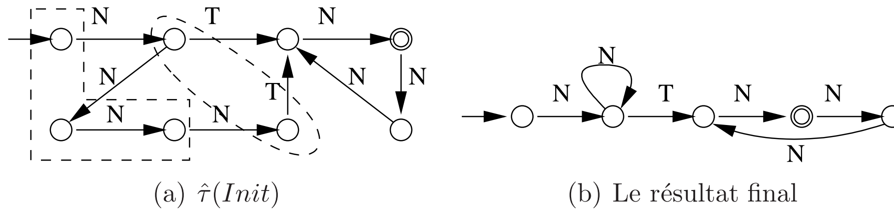
Équivalence basée sur les langages de longueur fini Nous présentons ici la possibilité de définir des schémas d'équivalence d'automates basée sur la comparaison des langages des états jusqu'à une certaine longueur. Nous présentons le schéma $\mathbb{F}_n^L$. Dans [29] d'autres schémas similaires sont introduits.

Le schéma $\mathbb{F}_n^L$ considère deux états d'un automate équivalents si leur *langages de mots jusqu'à une longueur fixée* n sont identiques. Formellement, pour un automate $M = (Q, \Sigma, \delta, q_0, F)$, $\mathbb{F}_n^L$ définit l'équivalence des états comme l'équivalence $\sim_M^n$ tel que $\forall q_1, q_2 \in Q : q_1 \sim_M^n q_2 \Leftrightarrow L^{\leq n}(M, q_1) = L^{\leq n}(M, q_2)$.

$\mathbb{F}_n^L$ est clairement *fini*. Il peut être raffiné en incrémentant la borne n sur la longueur des mots considérés. Puisque nous travaillons avec des automates déterministes finis nous obtenons le raffinement faible décrit ci-dessus. Quand n est incrémenté au-delà du nombre d'états moins un de M_k de la figure 2.8, cela garanti que tous les états peuvent être distingués par $\sim_M^n$ (car M_k est déterministe minimal), et M_k par conséquent ne change pas avec l'abstraction correspondante.

Dans la figure 2.10, nous appliquons $\mathbb{F}_n^L$ à l'exemple 2.6 (figure 2.7). Nous choisissons $n = 2$. Dans ce cas, l'abstraction de l'automate $Init$ est $Init$ luimême. La figure 2.10(a) indique les états de $\hat{\tau}(Init)$ qui ont le même langage de mots jusqu'à longueur deux et sont par conséquent équivalents. Quand ils sont fusionnés nous obtenons après déterminisation et minimisation l'automate de la figure 2.10(b) qui est un point fixe. On remarque que cet automate est une sur-approximation différente des configurations atteignables de l'automate obtenu en utilisant $\mathbb{F}_{\mathcal{P}}$. Si nous choisissons $n = 1$, nous obtenons un résultat similaire, mais nous avons besoin d'un pas de raffinement.

Notons que d'après notre expérience pratique l'incrément de n par $|Q_M| - 1$ est très souvent trop grand. Alternativement, on peut incrémenter n par d'autres valeurs plus petites (comme juste 1 par exemple). Dans ces cas, l'exclusion du mauvais calcul n'est pas garanti immédiatement mais uniquement plus tard. Pour la *valeur initiale de* n on peut, par exemple, utiliser le nombre d'états dans l'automate qui décrit des configurations initiales (ce qui évite de



(a) $\hat{\tau}(Init)$ (b) Le résultat final

FIG. 2.10 – Un exemple utilisant l'abstraction basée sur les langages de mots de longueur jusqu'à n (pour $n = 2$)

l'abstraire directement) ou juste 1. Nous avons considéré plusieurs études de cas. Elles sont détaillées dans [29]. Les systèmes considérés entre autres sont des réseaux paramétrés de processus (algorithmes de Dijsktra, Burns, etc.), les automates à pile, les automates à file, les réseaux de Petri, les automates à compteurs et les programmes à listes chaînées (plus de détails se trouvent dans la section 3.1.2).

2.3.1.4 Extension aux arbres

Dans [27] nous décrivons comment étendre les techniques d'abstraction pour le regular model-checking aux automates d'arbre (abstract regular tree model-checking, ARTMC). Essentiellement, les méthodes d'abstraction se transposent naturellement des automates de mots aux automates d'arbres. Dans [28] nous donnons une adaptation de [27] pour des programmes avec structures de données arborescentes. Ces travaux sont résumés dans la section 3.2. Dans [24] nous montrons comment utiliser les automates non-déterministes dans le cadre du regular tree model-checking avec abstraction. Ce travail est résumé dans la section 2.4.1.

2.3.2 Apprentissage

Dans cette section nous résumons notre approche décrite dans [70]. Cette approche est motivée par l'observation que pour des systèmes infinis qui peuvent être modélisés par des transducteurs préservant la longueur, il y a un calcul fini qui permet d'obtenir toutes les configurations jusqu'à une certaine taille. Ces configurations peuvent être vues comme un échantillon des configurations atteignables d'un système donné. Ensuite, les méthodes

développées pour inférer des langages réguliers peuvent être utilisées pour généraliser l'échantillon avec le but d'obtenir tout l'ensemble d'atteignabilité ou une sur-approximation de cet ensemble suffisant pour montrer la propriété que l'on veut vérifier. Nous utilisons l'algorithme de Trakhtenbrot-Barzdin [117] pour l'inférence. Un avantage de la méthode est qu'elle s'arrête pour tout système, *si son ensemble d'atteignabilité est régulier*.

Avant de proposer la méthode qui est principalement ciblée pour les transducteurs préservant la longueur nous notons que le problème d'atteignabilité pour des transducteurs qui ne préservent pas la longueur peut être ramené au même problème pour les transducteurs préservant la longueur. En effet, pour des calculs finis du système, nous pouvons toujours remplacer des transitions du transducteur qui ajoutent ou retirent un symbole par l'utilisation d'un symbole spécial $\perp$. Plus précisément, nous ajoutons des boucles étiquetées par $\perp, \perp$ dans chaque état de τ (et τ_{Bad}, si utilisé), et remplaçons chaque transition ϵ, a par une transition $\perp, a$, chaque transition a, ϵ par une transition $a, \perp$ et nous ajoutons des boucles étiquetées $\perp$ à chaque état de $Init$ et Bad. Pour une description de cette méthode voir aussi [76].

2.3.2.1 Inférence de langages réguliers d'ensemble d'apprentissage complet

L'inférence de langages régulier est un domaine de recherche très actif (voir par exemple [117, 99, 84, 54]). Nous nous plaçons ici dans le cadre de l'inférence à partir d'exemples. Il existe aussi des algorithmes d'inférence [8] où l'existence d'un *professeur* qui peut répondre à des questions d'appartenance et d'équivalence est supposée. Nous revenons dans la section 2.4.2.6 sur cette approche. Essentiellement, le problème de l'inférence de langage régulier à partir d'exemples consiste à inférer un langage régulier à partir de quelques mots qui lui appartiennent (ou des mots qui ne lui appartiennent pas). Une notion importante dans les algorithmes proposés est celle d'un ensemble d'apprentissage (ou échantillon). Un *ensemble d'apprentissage* $T = (T^+, T^-)$ est une paire de deux ensembles disjoints $T^+, T^- \subseteq \Sigma^*$, où T^+ contient des exemples positifs (des mots membres du langage à inférer) et T^- contient des exemple négatifs. Un ensemble d'apprentissage $T = (T^+, T^-)$ est appelé *n-complet* si $T^+ \cup T^- = \Sigma^{\leq n}$. Pour l'inférence de langages réguliers à partir d'ensemble d'apprentissage il y a plusieurs algorithmes. Nous utilisons ici l'*algorithme de Trakhtenbrot-Barzdin* (abrégé par l'algorithme TB) [117] qui prend comme entrée un ensemble d'apprentissage n-complet qui peut être

obtenu dans notre cadre (voir ci-dessous).

Pour des transducteurs qui préservent la longueur, le problème 2.4 peut être vu comme un problème d'inférence de langage de la façon suivante : Nous voulons calculer (ou au moins approcher) l'ensemble $\tau^*(Init)$ pour un transducteur τ donné qui préserve la longueur et un ensemble régulier $Init$. Puisque τ préserve la longueur l'ensemble $\tau^*(Init^{\leq n})$ est fini pour chaque n et peut être calculé en itérant τ un nombre fini de fois (nous avons $id \subseteq \tau$). Par ailleurs, chaque mot de longueur plus petite ou égale à n qui n'est pas dans $\tau^*(Init^{\leq n})$ n'est pas dans $\tau^*(Init)$ non plus. Grâce à cela, les ensembles $\tau^*(Init^{\leq n})$ et $\Sigma^{\leq n} \setminus \tau^*(Init^{\leq n})$ sont des ensembles d'exemples positifs et négatifs du langage $\tau^*(Init)$ que nous voulons inférer. Plus précisément, ils contiennent exactement *tous* les exemples positifs et négatifs de mots du langage jusqu'à une certaine longueur et sont donc un ensemble d'apprentissage n-complet.

En utilisant des n de plus en plus grands, nous obtenons de plus en plus d'exemples positifs et négatifs. L'ensemble d'apprentissage grandit. Si $\tau^*(Init)$ est régulier cela signifie que nous allons finalement (nous ne savons pas quand par contre) obtenir un ensemble d'apprentissage T qui est assez grand pour que l'algorithme TB puisse inférer $\tau^*(Init)$ à partir de T. Si $\tau^*(Init)$ n'est pas régulier, il est possible d'obtenir quand même une sur-approximation suffisante pour prouver la propriété. La même approche peut être utilisé pour essayer d'inférer la relation τ^* pour des transducteurs qui préservent la longueur en considérant comme alphabet des paires de lettres et en calculant τ^* restreint au mot de longueur inférieure ou égal à n.

2.3.2.2 L'algorithme de Trakhtenbrot-Barzdin

Pour décrire l'algorithme nous avons besoin de quelques définitions. Un DFA A est appelé *consistant* avec un ensemble d'apprentissage $T = (T^+, T^-)$ si $T^+ \subseteq L(A)$ et $L(A) \cap T^- = \emptyset$. Un ensemble d'apprentissage $T = (T^+, T^-)$ est *n-complet par rapport à un DFA A* si $T^+ = L^{\leq n}(A)$. Étant donné un ensemble d'apprentissage n-complet $T = (T^+, T^-)$, nous appelons un automate déterministe $A_T = (Q, \Sigma, \delta, Q_0, F)$ l'*automate de préfixes* de T si $L(A) = T^+$, A_T a la forme d'un arbre et ne contient aucun état acceptant le langage vide (si $T^+ \neq \emptyset$).

Nous décrivons maintenant une version légèrement modifiée de l'algorithme de Trakhtenbrot-Barzdin qui calcule un DFA inféré appelé *automate de cible* avec le nombre minimal d'état qui est consistant avec un ensemble

d'apprentissage n-complet donné. Soit $T = (T^+, T^-)$ un ensemble d'apprentissage n-complet et A_T l'automate de préfixes déterministe de T. Il est clair que les états de A_T doivent correspondre à des états de l'automate cible $\bar{A}_T$, parce qu'il doit accepter tous les mots acceptés par A_T. Par contre, plusieurs états différents de A_T peuvent correspondre au même état de $\bar{A}_T$. Par conséquent, l'idée principale de l'algorithme est de fusionner deux états de A_T si cela n'a pas comme conséquence qu'un mot d'une longueur inférieure ou égale à n sera accepté bien qu'il ne soit pas accepté par A_T. Deux états q et q' de A_T peuvent être fusionnés s'ils sont *compatibles*, c'est-à-dire s'ils sont k-indistinguables ($q \equiv_k q'$) où k est le minimum des deux hauteurs des sous-arbres à partir de q et q'.

Soit *succ* la fonction qui associe à chaque état de A_T son successeur dans un ordre de largeur d'abord. L'algorithme modifie A_T en fusionnant des états compatible :

Algorithme 2.8

`entrée :` *A_T avec état initial q_0*
$q_1 := q_0$;
`tant que` *il y a un successeur de q_1 dans A_T* `faire`
  $q_1 := succ(q_1)$;
  $q_2 := q_0$;
  `tant que` $q_1 \neq q_2$ `et pas` $compatible(q_1, q_2)$ `faire`
    $q_2 := succ(q_2)$;
  `fin tant que`
  `si` $q_1 \neq q_2$ `et` $compatible(q_1, q_2)$ `alors`
    Changer la transition de $pere(q_1)$ vers q_1 en la redirigeant vers q_2 et effacer q_1 et tous ses enfants de A_T ;
`fin tant que`
`sortie :` *l'automate modifié A_T*

La différence avec l'algorithme original de [117] est que celui-ci utilise comme entrée des arbres complets (chaque état intérieur a un fils pour chaque lettre). Dans notre cadre, cela n'est pas nécessaire puisque nous considérons uniquement des langages et pas le comportement de sortie des automates. L'algorithme a une complexité de $O(mn^2)$, où m est la taille de A_T et n la taille de l'automate cible. Dans les figures 2.11 et 2.12, nous donnons les différentes étapes de l'algorithme TB avec en entrée un ensemble d'apprentissage 2-complet de $\tau^*(Init)$ et un ensemble 3-complet de τ^* de l'exemple 2.1.

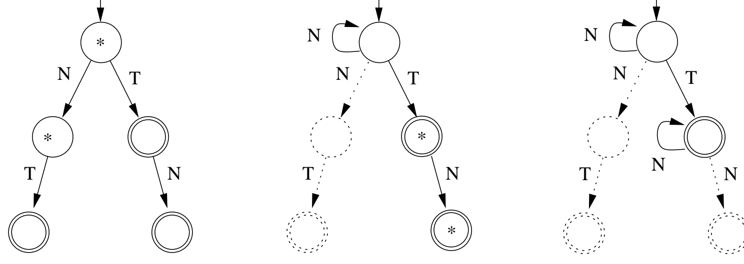


FIG. 2.11 – Un ensemble d'apprentissage 2-complet et les différentes étapes de l'algorithme TB

À chaque étape, deux états fusionnés sont marqués avec $*$. Dans le premier cas, $\tau^*(Init)$ est obtenu exactement, tandis que dans l'autre cas le résultat est une sur-approximation de τ^*.

Nous avons le théorème suivant [117].

Theorème 2.9 *Soit T un ensemble d'apprentissage n-complet. L'algorithme 2.8 calcule un DFA $\bar{A}_T$ consistant avec T avec un nombre minimal d'états.*

Il peut y avoir plusieurs DFA différents consistants avec T avec un nombre minimal d'états. L'algorithme TB calcule juste un de ceux-là. Si tous les mots de l'ensemble d'apprentissage viennent d'un automate déterministe minimal A, alors l'algorithme TB est garanti de l'inférer à partir d'un ensemble d'apprentissage n-complet où n est suffisamment grand par rapport à la taille de l'automate. Le *degré de reconstructibilité* r d'un automate A est défini comme $r = d + \rho + 1$ où d est sa profondeur et ρ son degré de distinguabilité. Alors, nous avons le théorème suivant [117].

Theorème 2.10 *Étant donné un DFA minimal A avec degré de reconstructibilité r et un ensemble d'apprentissage T qui est r-complet par rapport à A, l'algorithme 2.8 calcule A (à isomorphisme près).*

Si A a n états, alors nous avons au pire des cas $d = \rho = n-1$ et $r = 2n-1$. Pour cette raison l'ensemble d'apprentissage complet doit contenir un nombre exponentiel (en n) de mots. Mais en moyenne [117, 84], r est beaucoup plus petit et on peut montrer que la valeur moyenne de d est de $Clog_{|\Sigma|}(n)$ où C est une constante qui dépend de Σ et $\rho = log_{|\Sigma|}log_2(n)$. Cela signifie qu'en moyenne, le degré de reconstructibilité r est petit comparé avec la taille de l'automate et uniquement des ensembles d'apprentissage petit (de taille polynomiale en n) doivent être considérés pour reconstruire l'automate.

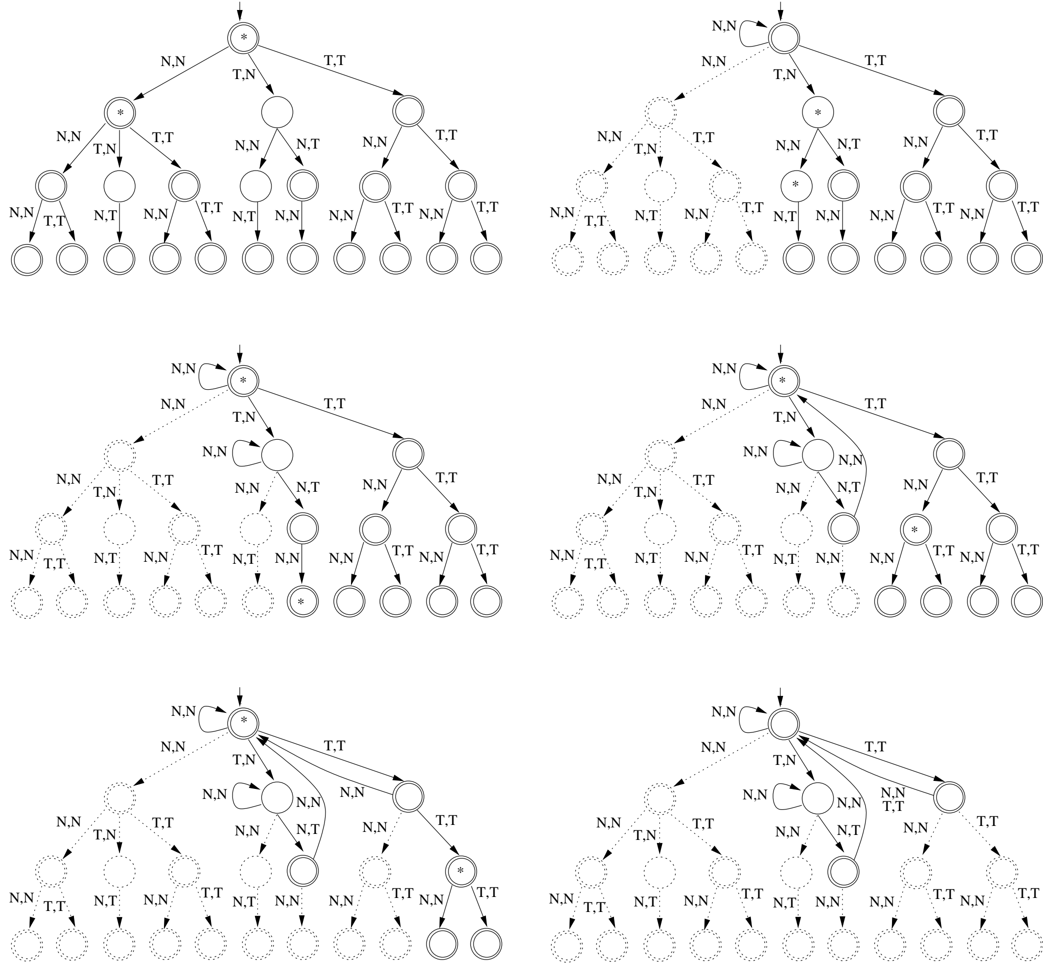


Fig. 2.12 – Un ensemble d'apprentissage 3-complet et les différentes étapes de l'algorithme TB

2.3.2.3 L'algorithme de model-checking

Nous décrivons notre algorithme de model-checking basé sur l'inférence de langages réguliers. Nous commençons avec une version de base et ensuite nous donnons quelques modifications et extensions de cet algorithme. L'idée de l'algorithme est de calculer des ensembles d'apprentissage complets de plus en plus grands. Ces ensembles viennent du langage $\tau^*(Init)$. Ensuite nous inférons un automate à partir des ensembles d'apprentissage et testons si l'automate est un invariant suffisant pour prouver la propriété. Comme

algorithme d'inférence on peut utiliser en principe tous les algorithmes basés sur des exemples positifs et négatifs proposés dans la littérature (par exemple RPNI [99, 84]). L'algorithme de Trakhtenbrot-Barzdin est le plus adapté à notre situation car il utilise des ensembles d'apprentissage complets et est garanti de donner l'automate d'origine, si on lui présente des ensembles d'apprentissage suffisamment grands.

Algorithme 2.11

`entrée` : *un transducteur qui préserve la longueur* τ,
un ensemble régulier de configurations initiales $Init$,
et un ensemble régulier de mauvaises configurations Bad
$i := 1$; `/*` *i peut être initialisé différemment aussi.* `*/`
`Repètez`
  $C := \tau^*(Init^{\leq i})$;
  $\overline{C} := \Sigma^{\leq i} \setminus C$;
  `si` $Bad \cap C \neq \emptyset$ `alors`
    `sortie` : *Propriété violée ;*
  $A := inference(C, \overline{C})$;
  $i := i + 1$;
`jusqu'à` $\tau(L(A)) \subseteq L(A)$ `et` $Init \subseteq L(A)$ `et` $L(A) \cap Bad = \emptyset$;
`sortie` : *Propriété satisfaite*

Quand nous utilisons notre version de l'algorithme TB (algorithme 2.8) comme algorithme d'inférence à l'intérieur de l'algorithme 2.11, l'appel de $inference(C, \overline{C})$ lance l'algorithme 2.8 en prenant l'automate de préfixes de C comme entrée. Dans ce cas, le calcul de $\overline{C}$ n'est pas nécessaire.

Dans l'exemple 2.1 pour vérifier la propriété $\tau^*(Init) \cap Bad = \emptyset$, l'algorithme s'arrête pour $i = 2$ (l'invariant inféré est exactement $\tau^*(Init)$), et pour la propriété $\tau^* \cap \tau_{bad} = \emptyset$, l'algorithme s'arrête pour $i = 3$ avec une sur-approximation de τ^* (voir la figure 2.12).

Pour le calcul de $\tau^*(Init^{\leq i})$, on peut réutiliser $\tau^*(Init^{\leq i-1})$ et uniquement calculer les configurations atteignables de taille i. L'algorithme essaie des ensembles d'apprentissage de plus en plus grands jusqu'à ce qu'il termine parce qu'il trouve soit un contre-exemple à la propriété (l'ensemble Bad est la négation d'une propriété) où un invariant qui contient les configurations initiales et qui n'intersecte pas les mauvaises configurations. Le test $Init \subseteq L(A)$ est nécessaire puisque pour des petits i, les exemples générés

ne sont peut-être pas suffisant pour reconstruire $Init$. Une alternative serait d'initialiser i par rapport à $Init$ (pour être sur de retrouver $Init$ dans l'automate inféré), mais il y a des exemples où des i plus petits suffissent pour prouver la propriétés.

Si $\tau^*(Init)$ est régulier, alors l'algorithme termine.

Theorème 2.12 *Soit τ un transducteur qui préserve la longueur et $Init$ et Bad deux ensembles réguliers. Si $\tau^*(Init)$ est régulier, alors l'algorithme 2.11 avec l'algorithme 2.8 utilisé comme algorithme d'inférence termine toujours.*

Remarquons que la terminaison de l'algorithme avec la propriété vérifiée signifie qu'un invariant suffisamment précis a été inféré. En général on ne peut pas tester si l'ensemble d'atteignabilité exact $\tau^*(Init)$ a été inféré. Cela suit, par exemple, du fait que pour des systèmes à canaux non-fiables $\tau^*(Init)$ est régulier [39, 2] mais pas calculable [1, 90]. Du Théorème 2.12, nous obtenons facilement :

Corollaire 2.13 *Le problème de model-checking 2.4 est décidable si $\tau^*(Init)$ est régulier.*

Cela n'est pas très surprenant, car nous pouvons donner deux procédures de semi-décision pour le problème. La première essaie de trouver des contre-exemples de plus en plus grands et la deuxième énumère tous les langages réguliers et teste si ce sont des invariants (comme expliqué dans [100] pour des système à canaux FIFO). Notre algorithme nous donne une façon "intelligente" d'énumérer des langages réguliers qui sont des candidats pour un invariant.

Il est clair, que de la même façon on peut traiter le problème avec les relations de transitions.

Corollaire 2.14 *Le problème de model-checking 2.5 est décidable si τ^* est régulier.*

Dans [70] nous montrons que la méthode présentée dans cette section donne des résultats pratiques intéressants.

2.3.2.4 Travaux connexes

L'idée de l'utiliser l'apprentissage pour le regular model-checking a été introduite par [58]. L'apprentissage est également étudié dans une série de travaux par Vardhan et ses coauteurs [118, 119, 120, 121].

2.4 Regular model-checking avec automates non-déterministes

Un des goulots d'étranglement du regular model-checking (avec abstractions) que nous avons observé en pratique, est le fait que la taille des automates considérés peut devenir très grande pendant un calcul abstrait de vérification. Cela est entre autres dû au fait qu'après chaque étape d'application du transducteur l'automate produit est non-déterministe et nous le déterminisons et minimisons ensuite. L'automate déterministe obtenu peut être beaucoup plus grand que l'automate non-déterministe et beaucoup de temps est passé dans la déterminisation.

Une possible solution à ce problème est de ne pas déterminiser et d'utiliser les automates non-déterministes dans tout le processus du regular model-checking avec abstractions. Pour cela les opérations nécessaires sur les automates doivent être effectuées sur les automates non-déterministes. Cela est simple pour l'application d'un transducteur et les abstractions. Par contre pour tester l'équivalence de deux automates directement (sans passer par la déterminisation) nous avons besoin de nouvelles méthodes. Nous expliquons ces méthodes dans la section suivante. Ensuite, nous détaillons un algorithme d'apprentissage pour les automates non-déterministes et son utilité pour le regular model-checking.

2.4.1 Les antichaînes

Pour tester si deux automates non-déterministes sont équivalents, on peut utiliser les algorithmes d'inclusion basés sur les antichaînes présentés dans [122]. L'idée principale de l'algorithme qui teste, si $L(A) \subseteq L(B)$ où A et B sont deux automates non-déterministes, est de "déterminiser" à la volée A mais de s'arrêter dès qu'un mot qui est dans $L(A)$ mais pas dans $L(B)$ est trouvé. On mémorise uniquement des antichaînes de sous-ensembles d'états permettant une efficacité intéressante en pratique.

Nous adaptons cette méthode dans [24] aux automates d'arbre ce qui a permis (en utilisant aussi les méthodes de réduction des automates non-déterministes par des relations de simulation [6]) une amélioration significative de la vitesse du regular tree model-checking avec abstractions.

2.4.2 Apprentissage des automates non-déterministes (NL*)

Dans cette section, nous décrivons NL*, un algorithme pour inférer des automates non-déterministes utilisant des requêtes d'appartenance et d'équivalence que nous avons introduit dans [21]. Les automates à états résiduels (RFSA) sont appris d'une façon similaire que dans l'algorithme classique d'Angluin L*[8] qui apprend des automates déterministes. Puisque les RFSA peuvent être exponentiellement plus petit que les DFA, ils sont intéressants à étudier pour des applications pratiques. Nous donnons les résultats de nos expériences montrant un avantage clair de NL* par rapport à L*. Nous esquissons également une possible utilisation de NL* dans le cadre du regular model-checking.

2.4.2.1 Les automates fini à états résiduels

Nous rappelons ici la notion d'*automates finis à états résiduels* (Residual finite state automata) introduit et étudié dans le travail fondateur [51]. Les RFSA sont une sous-classe de NFA qui hérite quelques propriétés souhaitables de DFA. La plus importante pour l'apprentissage est que pour chaque langage régulier il y a un RFSA minimal unique qui l'accepte. Puisque cette propriété n'est pas vraie pour des NFA quelconque il semble difficile de trouver des algorithmes d'apprentissage raisonnables pour la classe entière NFA. En même temps, comme pour les NFA, les RFSA peuvent être exponentiellement plus petits que les DFA, ce qui est important pour des applications pratiques de l'apprentissage.

Les RFSA et DFA ont la propriété que les états des automates correspondent aux *langages résiduels* définis ci-dessous. Cela n'est pas le cas pour les NFA.

Definition 2.15 *Pour un langage $L \subseteq \Sigma^*$ et $u \in \Sigma^*$, nous écrivons $u^{-1}L$ pour l'ensemble $\{v \in \Sigma^* \mid uv \in L\}$. Un langage $L' \subseteq \Sigma^*$ est un* langage résiduel *de L s'il y a un $u \in \Sigma^*$ avec $L' = u^{-1}L$. Nous écrivons $Res(L)$ pour l'ensemble de tous les langages résiduels de L.*

Pour simplifier, nous appelons aussi *résiduels* les langages résiduels. Le théorème de Myhill-Nerode dit que le nombre de langages résiduels d'un langage est fini ssi ce langage est régulier [96]. En outre, pour une DFA M minimal, il y a une bijection entre ses états et les langages résiduels de $L(M)$.

Nous pouvons maintenant introduire les automates à états résiduels.

Definition 2.16 *Un* automate à états résiduels *(RFSA) sur* Σ *est un NFA* $\mathcal{R} = (Q, \Sigma, \delta, Q_0, F)$ *telle que pour chaque* $q \in Q$, $L_q \in Res(L(\mathcal{R}))$.

Autrement dit, chaque état accepte un langage résiduel de $L(\mathcal{R})$, mais pas chaque langage résiduel doit être accepté par un *seul* état. Intuitivement, les états d'un RFSA sont des sous-ensembles des états du DFA minimal correspondant. Mais, en utilisant le non-déterminisme certains états d'un DFA minimal ne sont pas nécessaires puisqu'ils correspondent à l'union de langages d'autres états. Pour cela, nous distinguons les résiduels *premiers* et *composés* : Un résiduel est appelé *composé*, si il est l'union non-triviale d'autres résiduels. Sinon, il est appelé *premier*.

Definition 2.17 *Soit* $L \subseteq \Sigma^*$ *un langage. Un résiduel* $L' \in Res(L)$ *est appelé* composé *s'il y a* $L_1, \ldots, L_n \in Res(L) \setminus \{L'\}$ *tels que* $L' = L_1 \cup \ldots \cup L_n$. *Sinon, il est appelé* premier*. L'ensemble de tous les résiduels premiers de L est noté par* $Primes(L)$.

Nous définissons maintenant le RFSA *canonique* d'un langage régulier. L'idée est que son ensemble d'états correspond exactement aux résiduels premiers. Par ailleurs la fonction de transition doit être *saturée* dans le sens qu'une transition vers un état doit exister, si elle ne change pas le langage accepté. Formellement nous avons :

Definition 2.18 *Soit L un langage régulier. Le RFSA canonique de L, noté* $\mathcal{R}(L)$, *est le quintuplet* $(Q, \Sigma, \delta, Q_0, F)$ *où* $Q = Primes(L)$, $Q_0 = \{L' \in Q \mid L' \subseteq L\}$, $F = \{L' \in Q \mid \epsilon \in L'\}$, *et* $\delta(L_1, a) = \{L_2 \in Q \mid L_2 \subseteq a^{-1}L_1\}$.

Notons que le RFSA d'un langage régulier est bien défini, puisque l'ensemble de résiduels premiers d'un langage régulier est fini, et, pour chaque $a \in \Sigma$ et $L' \in Res(L)$, nous avons $a^{-1}L' \in Res(L)$. De plus, nous avons $L(\mathcal{R}(L)) = L$. Par définition, il y a un seul et unique RFSA pour chaque langage régulier. Nous appelons un RFSA $\mathcal{R}$ *canonique* s'il est le RFSA canonique de $L(\mathcal{R})$.

2.4.2.2 L'algorithme d'apprentissage d'Angluin L*

L'algorithme d'Angluin L*[8] apprend (infère) un DFA minimal pour un langage régulier donné. Dans l'algorithme, un *Apprenti*, qui ne connaît initialement rien du langage donné L essaie d'apprendre un DFA M tel que

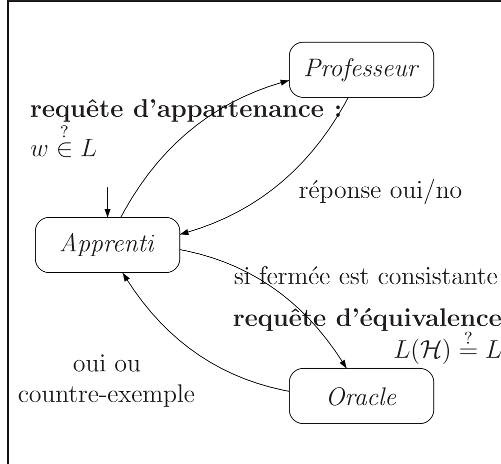


FIG. 2.13 – Les composantes de L* et leur interaction

$L(M) = L$. Pour cela, il pose successivement des questions à un *Professeur* et a un *Oracle*, qui tous les deux connaissent L. Il y a deux types de requêtes (voir la figure 2.13) :

- Une *requête d'appartenance* consiste à demander au *Professeur* si un mot $w \in \Sigma^*$ est dans L.
- Une *requête d'équivalence* consiste à demander à l'*Oracle* si un DFA *hypothèse* $\mathcal{H}$ est correcte, c.-à-d., si $L(\mathcal{H}) = L$. L'*Oracle* répond *oui* si $\mathcal{H}$ est correcte, ou fournit un contre-exemple w de la différence symétrique de L et $L(\mathcal{H})$.

L'*Apprenti* maintient un ensemble $U \subseteq \Sigma^*$ de mots fermé par préfixe qui sont des candidats pour identifier des états, et un ensemble $V \subseteq \Sigma^*$ de mots fermé par suffixe qui sont utilisés pour distinguer de tels états. Les mots de U sont appelés d'habitude *mots d'accès* et les mots de V *expériences*. Les ensembles U et V sont augmentés quand cela est nécessaire. L'*Apprenti* fait des requêtes d'appartenance pour tous les mots de $(U \cup U\Sigma)V$, et organise les résultats dans une *table* $\mathcal{T} = (T, U, V)$ où la fonction T associe à chaque $w \in (U \cup U\Sigma)V$ un élément de $\{+, -\}$ où la *parité* $+$ signifie *accepté* et $-$ *pas accepté*. À un mot $u \in U \cup U\Sigma$, nous associons une fonction $row(u) : V \rightarrow \{+, -\}$ donnée par $row(u)(v) = T(uv)$. Une telle fonction est appelée *ligne* de $\mathcal{T}$ et l'ensemble de toutes les lignes d'une table est noté par $Rows(\mathcal{T})$. Nous écrivons $Rows_{\mathrm{upp}}(\mathcal{T}) = \{row(u) \mid u \in U\}$ pour l'ensemble des lignes de la partie "supérieure" de la table. De la même façon, les lignes de $Rows_{\mathrm{low}}(\mathcal{T}) =$

$\{row(u) \mid u \in U\Sigma\}$ apparaissent dans la partie inférieure. La table $\mathcal{T}$ est

- *fermée*, si pour tout $u \in U$ et $a \in \Sigma$ il y a $u' \in U$ tel que $row(ua) = row(u')$, et
- *consistante*, si pour tout $u, u' \in U$ et $a \in \Sigma$, $row(u) = row(u') \Rightarrow row(ua) = row(u'a)$.

Si $\mathcal{T}$ n'est pas fermée, nous trouvons un $u' \in U\Sigma$ tel que $row(u) \neq row(u')$ pour tout $u \in U$. Nous ajoutons u' à U et effectuant des requêtes d'appartenance pour chaque $u'av$ où $a \in \Sigma$ et $v \in V$. De la même façon, si $\mathcal{T}$ n'est pas consistant, nous trouvons $u, u' \in U$, $a \in \Sigma$ et $v \in V$ tels que $row(u) = row(u')$ et $row(ua)(v) \neq row(u'a)(v)$. Nous ajoutons alors av à V et effectuant des requêtes d'appartenance pour chaque uav où $u \in U \cup U\Sigma$. Quand $\mathcal{T}$ est fermée et consistant l'*Apprenti* construit un DFA hypothèse DFA $\mathcal{H} = (Q, \Sigma, \delta, \{q_0\}, F)$, où $Q = \{row(u) \mid u \in U\} = Rows_{\mathrm{upp}}(\mathcal{T})$, q_0 est la ligne $row(\epsilon)$, δ est définie par $\delta(row(u), a) = row(ua)$, et $F = \{r \in Q \mid r(\epsilon) = +\}$. Ensuite, l'*Apprenti* soumet $\mathcal{H}$ à une requête d'équivalence (demandant si $L(\mathcal{H}) = L$). Si la réponse est *oui*, la procédure d'apprentissage est terminé. Sinon le contre-exemple u retourné est pris en compte en ajoutant chaque préfixe de u (u lui-même inclue) à U, en étendant $U\Sigma$ en conséquence, et en effectuant des requêtes d'appartenance pour rendre la table fermée et consistante. Ensuite un nouveau DFA hypothèse et construit, etc. (voir la figure 2.13).

Remarque 2.19 *L^* peut être modifié en changeant le traitement des contre-exemples. Au lieu d'ajouter le contre-exemple et ses préfixes à U on peut ajouter le contre-exemple et tous ses suffixes à V. Cela assure que la table est* toujours *consistante [86].*

2.4.2.3 Apprentissage d'automates à états résiduels

Ici nous expliquons comment modifier l'algorithme d'apprentissage L* pour apprendre des automates non-déterministes de la classe RFSA au lieu de DFA.

Des tables vers RFSA Pour simplifier notre présentation, nous suivons les notations et notions d'Angluin. Nous utilisons aussi des tables $\mathcal{T} = (T, U, V)$ avec un ensemble de mots U fermé par préfixes, un ensemble de mots V fermé par suffixe, et une application $T : (U \cup U\Sigma)V \rightarrow \{+, -\}$. Comme ci-dessus, à chaque mot $u \in U \cup U\Sigma$ est associé une application

$row(u) : V \to \{+, -\}$. Les membres de U sont utilisés pour atteindre des états et les membres de V pour distinguer des états. La différence principale est que pas tous les lignes d'une table correspondent à des états d'un RFSA hypothèse, mais uniquement les lignes *premières*. Essentiellement, nous devons définir pour les lignes ce qui correspond au notions de *union*, *composé*, *premier*, et *sous-ensemble* introduites précédemment pour les langages.

Definition 2.20 *Soit* $\mathcal{T} = (T, U, V)$ *une table. Le* raccord $(r_1 \sqcup r_2) : V \to \{+, -\}$ *de deux lignes* $r_1, r_2 \in Rows(\mathcal{T})$ *est défini composante par composante pour chaque* $v \in V$ *:* $(r_1 \sqcup r_2)(v) = r_1(v) \sqcup r_2(v)$ *où* $- \sqcup - = -$ *et* $+ \sqcup + = + \sqcup - = - \sqcup + = +$.

Notons que l'opérateur de raccord est associatif, commutatif, et idempotent, mais que le raccord de deux lignes n'est pas forcément une ligne de $\mathcal{T}$.

Definition 2.21 *Soit* $\mathcal{T} = (T, U, V)$ *une table. Une ligne* $r \in Rows(\mathcal{T})$ *est appelée* composée *s'il y a des lignes* $r_1, \ldots, r_n \in Rows(\mathcal{T}) \setminus \{r\}$ *telles que* $r = r_1 \sqcup \ldots \sqcup r_n$*. Sinon,* r *est appelée* première*. L'ensemble des lignes premières dans* $\mathcal{T}$ *est noté* $Primes(\mathcal{T})$*. De plus, nous définissons* $Primes_{\text{upp}}(\mathcal{T}) = Primes(\mathcal{T}) \cap Rows_{\text{upp}}(\mathcal{T})$*.*

Definition 2.22 *Soit* $\mathcal{T} = (T, U, V)$ *une table. Une ligne* $r \in Rows(\mathcal{T})$ *est* couverte *par une ligne* $r' \in Rows(\mathcal{T})$*, noté* $r \sqsubseteq r'$*, si pour tout* $v \in V$*,* $r(v) = +$ *implique* $r'(v) = +$*. Si par ailleurs* $r' \neq r$*, alors* r *est* strictement couverte *par* r'*, noté* $r \sqsubset r'$*.*

Notons que r peut être couverte par r' et r et r' sont toutes les deux premières. Une ligne composée couvre toutes les lignes premières dont elle est composée.

Comme dans l'algorithme d'apprentissage d'Angluin, nous introduisons les concepts comparables à la fermeture et la consistance et nous les appelons RFSA-fermeture et RFSA-consistance.

Pour les DFA, la fermeture assure que chaque ligne de la partie inférieure et aussi présente dans la partie supérieure de la table. Pour les RFSA, cela se traduit vers l'idée que chaque ligne de la partie inférieure de la table est composée de lignes (premières) de la partie supérieure. Formellement,

Definition 2.23 *Une table* $\mathcal{T} = (T, U, V)$ *est appelée* RFSA-fermée *si, pour chaque* $r \in Rows_{\text{low}}(\mathcal{T})$*,* $r = \bigsqcup\{r' \in Primes_{\text{upp}}(\mathcal{T}) \mid r' \sqsubseteq r\}$*.*

Notons qu'une table est RFSA-fermée ssi chaque ligne première de la partie inférieure est une ligne première de la partie supérieure de la table.

L'idée de la consistance pour les DFA est la suivante : Supposons que deux mots u et u' de la table ont la même ligne. Cela suggère que les deux mots atteignent le même état dans l'automate à apprendre car ils ne peuvent pas être distingués par des mots $v \in V$. Par conséquent, ils induisent les même résidus. Alors, ua et $u'a$ doivent aussi induire les mêmes résidus, pour chaque $a \in \Sigma$. Autrement dit, s'il y a un $a \in \Sigma$ et un $v \in V$ tels que $T(uav) \neq T(u'av)$, alors les résidus induits par u et u' ne peuvent pas être les mêmes et doivent être distinguables par le suffixe av, qui est à ajouter à V.

Pour les RFSA, s'il y a u et u' avec $row(u) \sqsubseteq row(u')$, alors cela suggère que le résidu induit par u est un sous-ensemble du résidu induit par u'. Si cela est effectivement le cas, alors la même relation doit être vraie pour tous les successeurs ua et $u'a$. Cela est exprimé formellement comme suit :

Definition 2.24 *Une table $\mathcal{T} = (T, U, V)$ est appelée* RFSA-consistante *si, pour tout $u, u' \in U$ et $a \in \Sigma$, $row(u') \sqsubseteq row(u)$ implique $row(u'a) \sqsubseteq row(ua)$.*

À une table RFSA-fermée et RFSA-consistante nous pouvons associer un NFA. Nous montrons plus tard que ce NFA correspond à un RFSA canonique *après* la terminaison de l'algorithme d'apprentissage.

Definition 2.25 *Pour une table $\mathcal{T} = (T, U, V)$ qui est RFSA-fermée et RFSA-consistante, nous définissons un NFA $\mathcal{R}_{\mathcal{T}} = (Q, \Sigma, \delta, Q_0, F)$ avec*

- *$Q = Primes_{\mathrm{upp}}(\mathcal{T})$,*
- *$Q_0 = \{r \in Q \mid r \sqsubseteq row(\epsilon)\}$,*
- *$F = \{r \in Q \mid r(\epsilon) = +\}$, et*
- *$\delta(row(u), a) = \{r \in Q \mid r \sqsubseteq row(ua)\}$ pour $u \in U$ avec $row(u) \in Q$ et $a \in \Sigma$.*

Notons que $Primes_{\mathrm{upp}}(\mathcal{T}) = Primes(\mathcal{T})$, car $\mathcal{T}$ est fermée. En outre, $row(\epsilon)$ n'est pas dans Q_0, ssi elle est composée. Par ailleurs, δ est bien définie : Prenons u, u' avec $row(u) = row(u')$. Alors, $row(u) \sqsubseteq row(u')$ et $row(u') \sqsubseteq row(u)$. La consistance implique pour tout $a \in \Sigma$ que $row(ua) \sqsubseteq row(u'a)$ et $row(u'a) \sqsubseteq row(ua)$ et donc $row(ua) = row(u'a)$.

Pour le reste de la section, nous fixons une table $\mathcal{T} = (T, U, V)$ qui est RFSA-fermée et RFSA-consistante. Nous donnons quelques propriétés importantes de l'automate $\mathcal{R}_{\mathcal{T}}$ construit à partir de la table.

Lemma 2.26 *Soit* $\mathcal{R}_\mathcal{T} = (Q, Q_0, F, \delta)$*. pour tout* $u \in U$*, nous avons pour tout* $r \in \delta(Q_0, u)$ *que* $r \sqsubseteq row(u)$*.*

Le lemme suivant dit que chaque *état* de $\mathcal{R}_\mathcal{T}$ classifie correctement les mots V.

Lemma 2.27 *Soit* $\mathcal{R}_\mathcal{T} = (Q, Q_0, F, \delta)$*. Pour chaque* $r \in Q$ *et* $v \in V$*, nous avons :*

1. $r(v) = -$ *ssi* $v \notin L_r$*.*
2. $row(\epsilon)(v) = -$ *ssi* $v \notin L(\mathcal{R}_\mathcal{T})$*.*

Le lemme suivant donne une propriété intéressante pour les états dans une relation de couverture et les langages accepté par ces états :

Lemma 2.28 *Soit* $\mathcal{R}_\mathcal{T} = (Q, Q_0, F, \delta)$*. Pour chaque* $r_1, r_2 \in Q$*,* $r_1 \sqsubseteq r_2$ *ssi* $L_{r_1} \subseteq L_{r_2}$*.*

L'automate $\mathcal{R}_\mathcal{T}$ construit à partir d'une table RFSA-fermée et RFSA-consistante $\mathcal{T}$ n'est pas forcément un RFSA canonique (voir [20]). Mais nous pouvons montrer que $\mathcal{R}_\mathcal{T}$ est un RFSA canonique *s'il est consistant avec la table*, c.-à-d. l'automate classifie correctement tous les mots de $\mathcal{T}$.

Definition 2.29 *Soit* $\mathcal{T}$ *une table RFSA-fermée et RFSA-consistante.* $\mathcal{R}_\mathcal{T}$ *est* consistant avec la table $\mathcal{T}$*, si pour tout* $w \in (U \cup U\Sigma)V$*, nous avons* $T(w) = +$ *ssi* $w \in L(\mathcal{R}_\mathcal{T})$*.*

Le lemme suivant est une version plus forte du lemme 2.26, si nous avons de plus que $\mathcal{R}_\mathcal{T}$ est consistant avec $\mathcal{T}$.

Lemma 2.30 *Soit* $\mathcal{T}$ *une table qui est RFSA-fermée et RFSA-consistante et supposons que* $\mathcal{R}_\mathcal{T} = (Q, Q_0, F, \delta)$ *est consistant avec* $\mathcal{T}$*. Alors, pour tout* $u \in U$ *avec* $row(u) \in Q$*, nous avons* $row(u) \in \delta(Q_0, u)$*.*

Ce lemme et les lemmes précédents permettent de montrer le théorème suivant.

Theorème 2.31 *Soit* $\mathcal{T}$ *une table RFSA-fermée et RFSA-consistante et soit* $\mathcal{R}_\mathcal{T}$ *consistant avec* $\mathcal{T}$*. Alors,* $\mathcal{R}_\mathcal{T}$ *est un RFSA canonique.*

L'algorithme Nous décrivons maintenant NL*. Son pseudo-code est donné dans la table 2.1. Après l'initialisation de la table $\mathcal{T}$ la table actuelle est testée successivement pour RFSA-fermeture et RFSA-consistance. Si l'algorithme détecte une violation de la condition de RFSA-fermeture (voir la définition 2.23), c.-à-d. une ligne $row(ua)$ avec $u \in U$ et $a \in \Sigma$ est première mais pas contenu dans $Primes_{\mathrm{upp}}(\mathcal{T})$, alors ua est ajouté à U. Cela peut entraîner des requêtes d'appartenance additionnelles. De l'autre côte, à chaque fois que l'algorithme détecte une violation de la condition de RFSA-consistance (voir la définition 2.24) un suffixe av peut être déterminé qui distingue deux lignes existantes. Dans ce cas, une colonne est ajoutée à V entraînant des requêtes supplémentaires. Cette procédure est répétée jusqu'à ce que $\mathcal{T}$ est RFSA-fermée et RFSA-consistante. Si les deux propriétés sont satisfaites une hypothèse $\mathcal{R}_{\mathcal{T}}$ peut être construite à partir de $\mathcal{T}$ (voir la définition 2.25) et soit un contre-exemple u de la différence symétrique de $L(M)$ et $L(\mathcal{R}_{\mathcal{T}})$ est donné et $Suff(u)$ est ajouté à V en recommençant NL*ou l'algorithme d'apprentissage termine avec succès. Notons que l'algorithme assure que V est toujours fermé par suffixe et U fermé par préfixe.

Remarque 2.32 *Nous choisissons de traiter le contre-example comme dans la variante de L* décrite dans la remarque 2.19. En effet, traiter le contre-exemple comme dans L* original ne permet pas d'obtenir un algorithme qui termine [20]. Notre traitement du contre-exemple assure que chaque ligne peut apparaître une seule fois dans la partie supérieure de la table, car nous ajoutons seulement des lignes quand la table n'est pas RFSA-fermée.*

Montrer la terminaison de l'algorithme d'Angluin L* est assez simple. Dans notre contexte cela n'est plus si facile car comme nous montrons dans [20], après une requête d'équivalence ou une violation de la RFSA-consistance le nombre d'états de l'automate hypothèse n'augmente pas forcément (comme c'est le cas pour L*).

Nous donnons maintenant le théorème principal de cette section. Pour chaque langage régulier L, l'algorithme NL* infère le RFSA canonique $\mathcal{R}$ qui accepte L. Le difficulté principale est de montrer la terminaison et la complexité.

Theorème 2.33 *Soit n le nombre d'états du DFA complet et minimal M^* pour un langage régulier $L \subseteq \Sigma^*$ donné. Par ailleurs, soit m la longueur du plus grand contre-exemple retourné par le test d'équivalence (ou* 1 *si le*

```
NL*(Σ, DFA : M) :

initialiser 𝒯 = (T, U, V) par U = V = {ϵ} et
T(w) pour tout w ∈ (U ∪ UΣ)V
repeat
 while 𝒯 n'est pas RFSA-fermée ou pas RFSA-consistante
   do
     if 𝒯 n'est pas RFSA-fermée then
        trouveru ∈ U et a ∈ Σ tels que row(ua) ∈ Primes(𝒯) \ Primes_upp(𝒯)
         étendre la table à 𝒯 := (T', U ∪ {ua}, V) par des req. d'appartenance
        if 𝒯 n'est pas RFSA-consistante then
           trouver u ∈ U, a ∈ Σ, et v ∈ V tels que :
              T(uav) = − et
              T(u'av) = + pour un u' ∈ U tel que row(u') ⊑ row(u),
           etendre la table à 𝒯 := (T', U, V ∪ {av})
        /* 𝒯 est RFSA-fermée et RFSA-consistante */
        de 𝒯, construite le NFA hypothèse ℛ_𝒯 (voir la definition 2.25)
        /* faire une req. d'équivalence */
        if (L(M) = L(ℛ_𝒯))
           then  test d'équivalence réussit
           else  prendre le contre-exemple w ∈ (L(M) \ L(ℛ_𝒯)) ∪ (L(ℛ_𝒯) \ L(M))
                     étendre la table à𝒯 := (T', U, V ∪ Suff(w)) avec des req. d'appartenance
until le test d'équivalence réussit
return ℛ_𝒯
```

TAB. 2.1 – NL* : la version NFA de l'algorithme d'Angluin L*

test d'équivalence réussi toute de suite). Alors, NL* *retourne après au plus* $O(n^2)$ *requête d'équivalence et* $O(m|\Sigma|n^3)$ *requêtes d'appartenance un RFSA canonique* $\mathcal{R}(L)$ *avec* $L(\mathcal{R}) = L(M^*) = L$.

La complexité théorique que nous obtenons pour NL* en nombre de requêtes d'équivalence est plus grande comparée à L* ou au plus n requêtes d'équivalence sont nécessaires. La complexité en nombre de requêtes d'appartenance est aussi plus grande pour NL* (L* a besoin de grosso modo $|\Sigma|mn^2$ requêtes). Mais nous observons qu'en pratique *moins* de requêtes d'équivalence et d'appartenance sont nécessaires. (voir la section 2.4.2.5).

2.4.2.4 NL* à travers d'un exemple

Soient $\Sigma = \{a, b\}$ un alphabet et $L_n \subseteq \Sigma^*$ le langage de mots avec un a à la $(n+1)$ième position de la fin. Il correspond à l'expression régulière $\Sigma^* \, a \, \Sigma^n$.

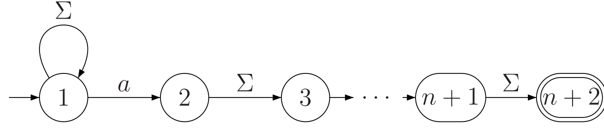


FIG. 2.14 – Un NFA sur $\Sigma = \{a, b\}$ qui accepte le langage L_n avec $n+2$ états

L_n est accepté par un DFA minimal M_n^* avec 2^{n+1} états. Néanmoins, il existe des NFA (voir la figure 2.14) avec seulement $n + 2$ états acceptant L_n. Il est facile de voir qu'il y a même un RFSA canonique $\mathcal{R}_n$ de taille $n + 2$ qui accepte L_n. $\mathcal{R}_n$ est donc exponentiellement plus concis que M_n^*.

Nous montrons maintenant comment L_2 (dont le DFA minimal M_2^* est donné par la figure 2.15) est appris par L* et par notre algorithme NL*. Nous commençons avec un calcul de L*.

$\mathcal{T}_1$	ϵ
ϵ	−
a	−
b	−

$\overset{1)}{\Rightarrow}$

$\mathcal{T}_2$	ϵ
ϵ	−
a	−
aa	−
aaa	+
b	−
ab	−
aab	+
$aaaa$	+
$aaab$	+

$\overset{2)}{\Rightarrow}$

$\mathcal{T}_3$	ϵ	a
ϵ	−	−
a	−	−
aa	−	+
aaa	+	+
b	−	−
ab	−	+
aab	+	+
$aaaa$	+	+
$aaab$	+	+

$\overset{3)}{\Rightarrow}$

$\mathcal{T}_4$	ϵ	a	ba
ϵ	−	−	−
a	−	−	+
aa	−	+	+
aaa	+	+	+
b	−	−	−
ab	−	+	−
aab	+	+	−
$aaaa$	+	+	+
$aaab$	+	+	−

$\overset{4)}{\Rightarrow}$

$\mathcal{T}_5$	ϵ	a	ba
ϵ	−	−	−
a	−	−	+
aa	−	+	+
aaa	+	+	+
ab	−	+	−
aab	+	+	−
b	−	−	−
$aaaa$	+	+	+
$aaab$	+	+	−
aba	+	−	+
abb	+	−	−
$aaba$	+	−	+
$aabb$	+	−	−

$\overset{5)}{\Rightarrow}$

$\mathcal{T}_6$	ϵ	a	ba
ϵ	−	−	−
a	−	−	+
aa	−	+	+
aaa	+	+	+
ab	−	+	−
aab	+	+	−
$aaba$	+	−	+
$aabb$	+	−	−
b	−	−	−
$aaaa$	+	+	+
$aaab$	+	+	−
aba	+	−	+
abb	+	−	−
$aabaa$	−	+	+
$aabab$	−	+	−
$aabba$	−	−	+
$aabbb$	−	−	−

La table $\mathcal{T}_1$ est fermée et consistante mais ne représente pas l'automate à apprendre car le mot aaa n'est pas accepté. Nous ajoutons donc $Pref(aaa)$ à U et $Pref(aaa)\Sigma$ à $U\Sigma$. Le résultat (après les requêtes d'appartenance nécessaires) est $\mathcal{T}_2$. Cette table est fermée mais pas consistante ($row(a) = row(aa)$ mais pas $row(aa) = row(aaa)$). Nous ajoutons donc la colonne a

et obtenons $\mathcal{T}_3$ qui n'est toujours pas consistante amenant à $\mathcal{T}_4$. Après avoir fermé la table nous obtenons $\mathcal{T}_6$ qui est consistante aussi et dont l'automate correspondant (voir la figure 2.15) accepte L_2.

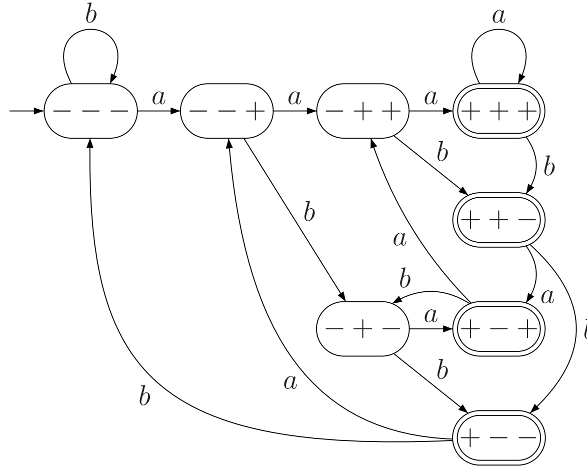


FIG. 2.15 – DFA minimal M_2^* acceptant le langage L_2 avec 8 ($= 2^{2+1}$) états

Maintenant nous donnons l'exécution de NL* sur L_2.

	$\mathcal{T}_1$	ϵ
$*$	ϵ	$-$
$*$	b	$-$
$*$	a	$-$

$\Longrightarrow_{ce}^{1.}$

	$\mathcal{T}_2$	ϵ	aaa	aa	a
$*$	ϵ	$-$	$+$	$-$	$-$
$*$	b	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$

$\Longrightarrow_{ncl}^{2.}$

	$\mathcal{T}_3$	ϵ	aaa	aa	a
$*$	ϵ	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$
$*$	b	$-$	$+$	$-$	$-$
$*$	ab	$-$	$+$	$-$	$+$
	aa	$-$	$+$	$+$	$+$

$\Longrightarrow_{ncl}^{3.}$

	$\mathcal{T}_4$	ϵ	aaa	aa	a
$*$	ϵ	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$
$*$	ab	$-$	$+$	$-$	$+$
$*$	b	$-$	$+$	$-$	$-$
	aa	$-$	$+$	$+$	$+$
$*$	abb	$+$	$+$	$-$	$-$
	aba	$+$	$+$	$+$	$-$

$\Longrightarrow_{ncl}^{4.}$

	$\mathcal{T}_5$	ϵ	aaa	aa	a
$*$	ϵ	$-$	$+$	$-$	$-$
$*$	a	$-$	$+$	$+$	$-$
$*$	ab	$-$	$+$	$-$	$+$
$*$	abb	$+$	$+$	$-$	$-$
$*$	b	$-$	$+$	$-$	$-$
	aa	$-$	$+$	$+$	$+$
	aba	$+$	$+$	$+$	$-$
$*$	$abbb$	$-$	$+$	$-$	$-$
$*$	$abba$	$-$	$+$	$+$	$-$

Les lignes avec $*$ indiquent les lignes premières. La table $\mathcal{T}_1$ est RFSA-fermée et RFSA-consistante mais ne représente pas l'automate à apprendre car le mot aaa n'est pas accepté. Nous ajoutons aaa et tous ses suffixes à V en effectuant des requêtes d'appartenance et obtenons ainsi la table $\mathcal{T}_2$ qui n'est pas fermée. Nous ajoutons a à U et continuons. Après avoir résolu deux violations de la condition de fermeture nous obtenons enfin la table $\mathcal{T}_5$ qui est

RFSA-fermée et RFSA-consistante et l'automate correspondant donné dans la figure 2.16 est le RFSA canonique pour L_2. Notons que la table $\mathcal{T}_5$ n'est pas fermée (dans le sens d'Angluin) et L* continuerait à ajouter des mots à la partie supérieure de la table.

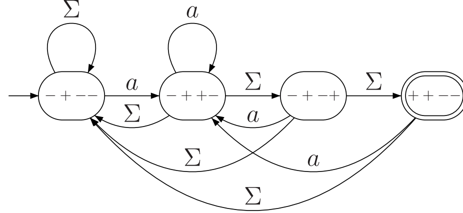


FIG. 2.16 – Le RFSA canonique $\mathcal{R}_2$ qui accepte L_2 avec $4 = 2 + 2$ états

2.4.2.5 Expériences

Pour évaluer la performance de NL*, nous comparons NL* avec l'algorithme L* et sa version modifiée par rapport à la remarque 2.19, que nous appelons L^*_{col}. Puisque NL* est similaire à L^*_{col}, une comparaison avec cet algorithme semble plus juste. Tous les algorithmes ont été implémentés en Java et testés sur beaucoup d'exemples. Nous suivons [52] en générant d'une façon aléatoire des grands ensembles d'expressions régulières sur des alphabets avec des tailles différentes. Une description détaillée de cela et les résultats se trouve dans [20] et [77]. Comme dans [52], nous présentons une sélection caractéristique des résultats pour un alphabet de taille deux.

Résultats Pour les diagrammes dans cette section, nous avons généré un ensemble de 3180 expressions régulières, qui donnent lieu à des DFA minimaux avec des tailles entre 1 et 200 états. Ces DFA ont été donné aux algorithmes d'apprentissage, c.-à-d. les requêtes d'appartenance et d'équivalence ont été traitées par rapport à ces automates. Pour évaluer la performance des algorithmes, nous avons mesuré pour chaque algorithme et chaque expression régulière, le *nombre d'états de l'automate final* (RFSA ou DFA) et le *nombre de requête d'appartenance (resp. d'équivalence)* nécessaire pour l'inférer. Comme la figure 2.17 montre, le nombre d'états de l'automate appris par NL* est considérablement plus petit que L* et L^*_{col} confirmant les résultats de [52]. D'une façon plus importante pour la pratique, les tailles des RFSA comparées aux DFA semblent suivre un écart exponentiel.

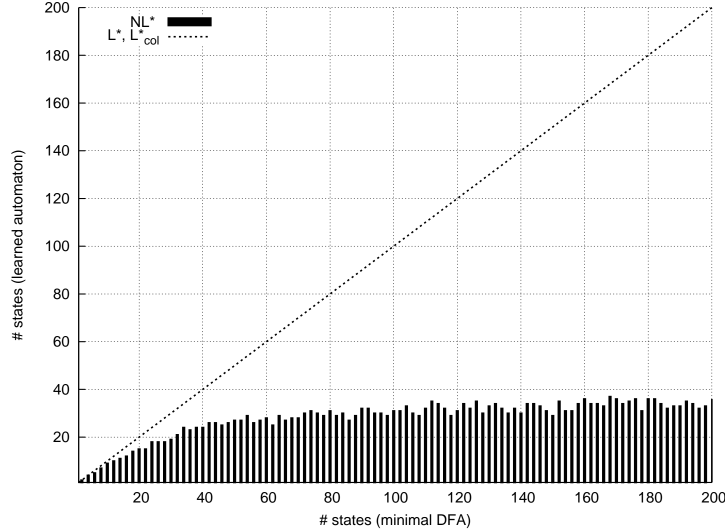


FIG. 2.17 – Nombre d'états

Dans la figure 2.18, le nombre de requêtes d'appartenance est donné. Comme dans le premier cas, NL* se comporte mieux que les autres algorithmes d'apprentissage. Tandis que la différence entre les courbes est assez petite pour les automates avec moins que 40 états, elle augmente d'une façon significative pour des automates plus grands. Le même est le cas pour le nombre de requêtes d'équivalence dans la figure 2.19. Cela est contraire au résultat théorique obtenu dans le théorème 2.33. Les expériences effectués montrent un avantage net de NL* sur L* et L$^*_{\text{col}}$ tant que l'utilisateur n'est pas dépendent d'un modèle déterministe.

2.4.2.6 Application

Nous avons donné en section 2.3.2 une méthode de vérification pour le regular model-checking basée sur l'apprentissage. Cet algorithme est spécifique parce qu'il suppose qu'on a à sa disposition un ensemble d'apprentissage complet. Nous donnons dans cette section un survol d'une autre méthode basée sur l'utilisation des algorithmes d'apprentissage utilisant les requêtes et les contre-exemples comme L* d'Angluin ou notre algorithme NL*. Nous voulons résoudre le problème 2.4, c.-à-d. étant donnés un alphabet Σ, deux ensembles réguliers $Init \subseteq \Sigma^*$ et $Bad \subseteq \Sigma^*$ et un transducteur fini τ sur Σ, est-ce que $\tau^*(Init) \cap Bad = \emptyset$?

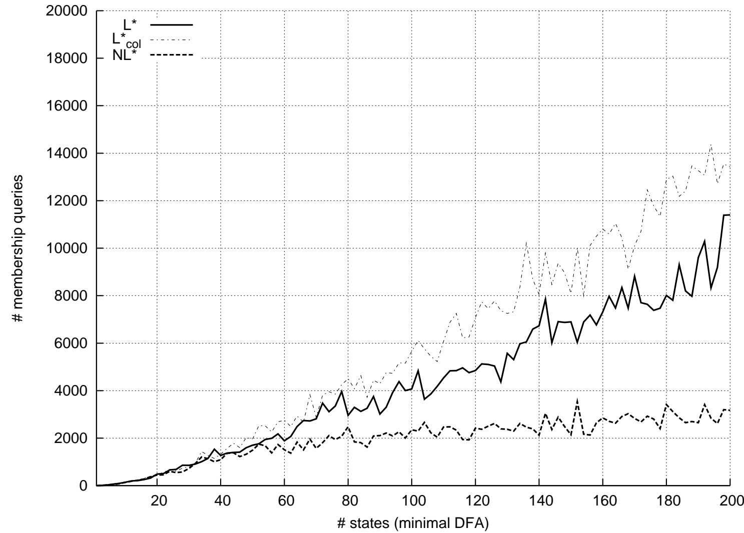


FIG. 2.18 – Nombre de requêtes d'appartenance

L'idée principale [119, 120, 121] est d'apprendre un invariant régulier Inv qui vérifie

1. $Init \subseteq Inv$
2. $\tau(Inv) \subseteq Inv$ et
3. $Inv \cap Bad = \emptyset$

Si un tel Inv est trouvé la réponse à la question posée est oui. Pour utiliser les algorithmes d'apprentissage à la Angluin nous devons pouvoir répondre aux requêtes d'appartenance et d'équivalence. Évidemment, Inv est ici inconnu du professeur, mais nous pouvons répondre aux requêtes d'appartenance en considérant $\tau^*(Init)$, c.-à-d. quand l'algorithme fait une requête d'appartenance sur un mot (une configuration) w nous répondons oui, ssi $w \in \tau^*(Init)$. Pour cela la question $w \in \tau^*(Init)$ doit être décidable. Une façon de la rendre décidable est d'ajouter au système une variable qui compte le nombre de fois τ a été appliqué. D'une façon naturelle cela convient par exemple aux systèmes avec compteurs [119, 120, 121].

Pour répondre aux requêtes d'équivalence avec une hypothèse donné $\mathcal{H}$, nous considérons la question, si $Init \subseteq \mathcal{H}$, $\tau(\mathcal{H}) \subseteq \mathcal{H}$ et $\mathcal{H} \cap Bad = \emptyset$. Si aucune des trois conditions est violées nous avons trouvé un invariant, sinon nous obtenons un mot qui doit être enlevé de l'hypothèse ou un mot qui doit être ajouté.

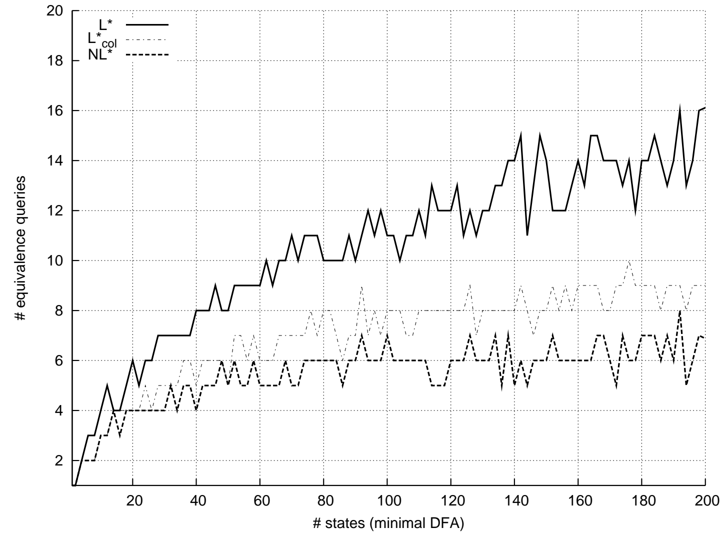


FIG. 2.19 – Nombre de requêtes d'équivalences

Cet algorithme brièvement esquissé est indépendant de l'algorithme d'apprentissage sous-jacent. Nous pouvons utilisé L^* et ses variantes (comme dans [119, 120, 121]) ou NL^*.

2.5 Perspectives

L'utilisation des automates non-déterministes a donné des résultats très intéressants. En effet, les différentes expériences que nous avons effectués montrent qu'en général nos méthodes basées sur les automates non-déterministes sont beaucoup plus rapides que celles sur les automates déterministes. Par contre, pour l'instant un outil très efficace pour les NFA comme l'est Mona [80] pour les DFA manque. Il est clair que l'utilisation des structures de représentation efficaces pour les NFA comme celles utilisées dans Mona (notamment les BDD) pour les DFA augmentera encore considérablement l'efficacité de nos méthodes et ouvrira des nouveaux champs d'application.

Chapitre 3

Programmes avec pointeurs

Dans ce chapitre nous donnons plusieurs méthodes de vérification de programmes avec pointeurs. D'abord nous traitons les programmes avec des listes, donc des données dynamiques qui ont des structures de base avec un pointeur (sélecteur) successeur. Nous esquissons deux techniques : la première, détaillé dans [26], est basée sur le regular model-checking (voir chapitre 2), la deuxième, détaillée dans [22], sur la traduction des programmes vers des automates à compteurs. Ensuite, nous donnons une technique [28] pour des structures arborescentes (plusieurs sélecteurs) basée sur le regular model-checking et qui en la combinant avec les automates à compteurs permet de montrer la terminaison de programmes [64].

3.1 Programmes avec listes

Nous considérons ici des programmes qui manipulent des structures de données liées dynamiquement avec un sélecteur pointeur. Cela correspond à des programmes qui manipulent des listes avec la possibilité de partager des parties de listes et d'avoir des circularités. Ce type de programme est utilisé couramment en pratique.

3.1.1 Définitions syntaxiques

La syntaxe abstraite des programmes considérés est donnée dans la figure 3.1.1. Lab est un ensemble fini d'étiquettes de programme (états de contrôle), $PVar$ est un ensemble fini de variables de pointeurs, et $IVar$ est un ensemble

fini de variables entières (compteurs).

$l \in Lab,\ u, v, w \in PVar,\ i, j, k \in IVar$

$$
\begin{array}{rcl}
Program & := & \{l : Stmnt\ ;\}^* \\
Stmnt & := & WhileStmnt \mid IfStmnt \mid Asgn \\
WhileStmnt & := & \text{while } Guard \text{ do } \{Stmnt\ ;\}^* \text{ od} \\
IfStmnt & := & \text{if } Guard \text{ then } \{Stmnt\ ;\}^* \left[\text{else } \{Stmnt\ ;\}^*\right] \text{ fi} \\
Asgn & := & u := \text{null} \mid u := \text{new} \mid u := w \mid u := w.next \mid \\
 & & u.next := \text{null} \mid u.next := w \mid i := 0 \mid i := i \pm 1 \\
Guard & := & u = v \mid u = \text{null} \mid u.data \leq v.data \mid i = 0 \mid \neg Guard \mid \\
 & & Guard \wedge Guard \mid Guard \vee Guard
\end{array}
$$

FIG. 3.1 – Syntaxe abstraite des programmes avec listes

```
1 : while i ≠ null do
2 :       k := i.next ;
3 :       i.next := j ;
4 :       j := i ;
5 :       i := k ;
6 :       od
```

FIG. 3.2 – Programme de renversement d'une liste

Nous considérons des programmes impératifs qui travaillent avec un ensemble de variables de pointeurs $PVar$ et un ensemble de variables de compteurs $IVar$. Les variables de pointeurs référencent des cellules de listes. Les pointeurs peuvent être utilisés dans des affectations comme `u := null`, `u := w` et `u := w.next`, des mises à jour de sélecteur `u.next := w` et `u.next := null`, et la création de cellules `u := new`. Les compteurs peuvent être incrémentés `i := i + 1`, décrémentés `i := i - 1` et mis à zéro `i := 0`. La structure de contrôle est composé d'itérations (`while`) et des conditions (`if-then-else`). Les gardes des structures de contrôle sont l'égalité de pointeurs `u = w`, la comparaison de données `u.data <= v.data`, des tests sur zéro

pour les compteurs `i = 0` et leurs combinaisons booléennes. Un exemple est le programme qui renverse une liste dans la figure 3.2. Notons que ce programme est correcte pour des listes non-circulaires et circulaires.

3.1.2 Appliquer le regular model-checking

Dans cette section nous décrivons brièvement une technique de vérification basée sur le regular model-checking. Cette technique est détaillée dans [26]. Nous considérons ici des programmes *sans* entiers et données. Il est néanmoins très facile d'ajouter des données de domaine *fini* à l'intérieur des listes.

3.1.2.1 Codage

Pour appliquer le regular model-checking nous devons donner un codage des configurations du système (ici, des mémoires) comme des mots et des opérations comme des transducteurs.

Une mémoire est codée comme la concaténation de plusieurs mots (séparés par un symbole spécial). Chaque mot représente une liste d'éléments. Des éléments successifs de ces listes sont donnés de gauche à droite et les positions des variables des pointeurs sont marquées par un symbole spécial. Nous considérons d'abord des mémoires qui ne contiennent ni de cycles ni de parties partagées. Nous utilisons l'alphabet Σ suivant : Pour chaque variable de pointeurs x utilisé dans le programme, nous avons $x \in \Sigma$, et Σ contient également les lettres | pour séparer les listes, / pour séparer les éléments des listes (donc / représente un pointeur), # pour exprimer qu'un pointeur pointe vers null, et ! pour le fait qu'un pointeur est indéfini.

Ensuite, nous pouvons coder une mémoire comme une séquence de parties séparées part | comme suit :

- La première partie contient une séquence de variables de pointeurs indéfinis. Nous fixons un ordre des variables en avance qui est respecté dans cette situation et les autres similaires ci-dessous.
- La deuxième partie contient les variables de pointeurs pointant vers null.
- La troisième partie contient les séquences de liste séparées par le symbole |. Chaque liste est codée comme suit : Chaque élément de liste est représenté par une séquence (possiblement vide) de variables de pointeurs pointant vers lui, les éléments sont séparés par le symbole /, et

les listes se terminent avec le symbole # (null) ou ! (indéfini).

Par exemple, le mot $k \mid j \mid i \;/\; / \;\# \mid$ code une configuration initiale possible de l'exemple de renversement d'une liste : k est indéfini, j pointe sur null, et i pointe sur une liste avec deux éléments.

Des ensembles réguliers peuvent être utilisés pour décrire des ensembles de mémoire. Pour l'exemple de renversement d'une liste l'expression régulière $(k \mid j \mid i \;/^+ \;\# \mid) \; + (k \mid j \mid i \mid)$ décrit toutes les mémoires initiales.

Remarquons que nous ne permettons pas de cellules inaccessibles dans la mémoire, c.-à-d. des cellules qui ne peuvent pas être atteintes à partir des variables de pointeurs en suivant les liens. Dès qu'une cellule inaccessible est crée par une opération, une erreur est signalée. En fait, cette situation correspond à une fuite de mémoire en C (en Java, on peut par contre toujours "ramasser les miettes" et enlever les cellules inaccessibles).

Remarque Chaque variable de pointeur apparaît exactement une fois dans chaque mot. Le séparateur | et les symboles # et ! apparaissent un nombre borné de fois puisque nous ne considérons pas les mémoires avec des cellules inaccessibles. Le symbole / par contre peut apparaître un nombre non-borné de fois.

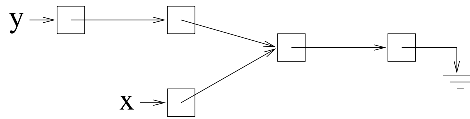


FIG. 3.3 – Une mémoire avec partage

Listes avec partage et/ou cycles Pour coder le partage de parties de listes comme par exemple dans la figure 3.3, nous étendons l'alphabet Σ avec un ensemble fini de paires de marqueurs (m_f,m_t, n_f, n_t, etc.). Un marqueur "from" X_f peut être utilisé après un symbole de pointeur "next" / pour indiquer que ce pointeur pointe vers un élément marqué par X_t (le marqueur "to" correspondant). Par exemple, le mot $\mid \mid x \;/\; m_f \mid y \;/\; / \; n_f \mid n_t \; m_t \;/\; / \;\# \mid$ représente la mémoire de la figure 3.3.

Cette mémoire peut être codée de plusieurs façon (par exemple aussi comme $\mid \mid x \;/\; n_t \;/\; / \;\# \mid y \;/\; / \; n_f \mid$). Bien que nous normalisions partiellement le codage en imposant un ordre sur les symboles attachés à la même cellule,

nous ne définissons ici pas de représentation canonique. Cela compliquerait le codage des opérations du programme et ne pose pas de problème en pratique.

Remarquons aussi que les marqueurs permettent de coder des listes circulaires (comme par exemple $| \; | \; x \; n_t \; / \; / \; n_f \; |$ qui correspond à une liste circulaire de deux éléments pointés par x).

Il n'est pas difficile de voir qu'étant donné une mémoire avec k variables de pointeurs codée avec plus que k paires de marqueurs, on peut coder la même mémoire avec au plus k marqueurs (en supposant qu'il n'y a pas de cellules inaccessibles).

Coder les opérations Nous décrivons ici notre représentation des opérations de programmes par des transducteurs. Nous traitons les programmes sans entiers et manipulations de données (autre que pointeurs). Une étape d'abstraction permet de les obtenir à partir de programmes plus généraux. De plus, nous supposons sans perte de généralité que les structures de contrôles sont de la forme `pointer_assignment ; goto l ;` ou `if (pointer_test) goto l1 ; else goto l2 ;`. Par ailleurs, en introduisons des variables auxiliaires, nous pouvons éliminer les déréférencements multiples de la forme `x.next.next` et considérer uniquement les déréférencements simples.

Nous ajoutons au codage une lettre indiquant la ligne actuelle du programme (succédée par un séparateur |). De plus, pour les besoins du codage nous ajoutons une lettre indiquant le mode de calcul qui est soit n (normal), e (erreur — un déréférencement d'un pointeur null ou la manipulation d'un pointeur indéfini a été détecté), s (déplacement, utilisé plus tard pour implémenter les manipulations de pointeurs qui ne peuvent pas être codé par un seul transducteur), ou u (inconnu, apparaît quand un nombre insuffisant de marqueurs est utilisé). Les configurations initiales de l'exemple de renversement d'une liste sont $(n \; l_1 \; | \; k \; | \; j \; | \; i \; /^+ \; \# \; |) \; + (n \; l_1 \; | \; k \; | \; j \; | \; i \; |)$

Les sauts conditionnels basés sur des tests comme `x==null` ou `x==y` sont assez faciles à coder. Le transducteur vérifie, si `x` est dans la section correspondant à null ou dans la même section que `y` (/ et | séparent ici les sections), et conformément à cela change la lettre qui codé la ligne actuelle du programme. Si `x` ou `y` sont dans la section des variables indéfinies, le transducteur change le mode de calcul vers erreur. D'une façon similaire, les affectations de la forme `x=null` ou `x=y` sont facile à coder—`x` est enlevée de sa position actuelle (utilisant une x, ε transition) et mise dans la section de y (utilisant une ε, x transition).

Un cas un peu plus compliqué est celui des tests basés sur `x.next` et l'affectation `y=x.next`. Mis à part la génération d'une erreur quand `x` est indéfini ou null, nous devons considérer le successeur de `x`, ce qui peut consister à suivre le "from" marqueur au "to" marqueur correspondant. Néanmoins, le transducteur pour facilement se rappeler vers quel marqueur il doit aller, car le nombre de marqueurs est fini.

Ajouter/enlever des marqueurs Le cas le plus difficile est celui des affectations de la forme `l.next=x`. Le transducteur essaie d'abord de simuler l'opération en utilisant une paire de marqueurs inutilisés (prenons m_f/m_t) d'un ensemble fixé de paires de marqueurs (une paire n'est pas utilisée, si ses marqueurs n'apparaissent pas dans la configuration actuelle). Ensuite, le transducteur met m_f dans la section après `l` et marque la section de `x` avec m_t. Par exemple, dans le renversement d'une liste, $n\ l_3 \mid \mid \mid j\ /\ /\ \#\mid i\ /\ k\ /\ \#\mid$ est transformé par `i.next=j` en $n\ l_5 \mid \mid \mid m_t\ j\ /\ /\ \#\mid i\ /\ m_f \mid k\ /\ \#\mid$

Il est possible, qu'il n'y a plus de marqueurs inutilisés. Dans ce cas, le transducteur essaie de libérer des marqueurs en réarrangeant la configuration. Cela peut être fait en déplaçant une séquence de cellules qui commence avec un marqueur "to" directement derrière le marqueur "from" correspondant (si ces deux marqueurs ne constituent pas une boucle). Comme expliqué dans la section 3.1.2.1, cela est toujours possible si le nombre de paires de marqueurs est suffisamment grand (plus que le nombre de variables de pointeurs). Par exemple, $n\ l_4 \mid \mid \mid m_t\ j\ /\ /\ \#\mid i\ /\ m_f \mid k\ /\ \#\mid$ peut être réarrangé vers $n\ l_4 \mid \mid \mid i\ /\ j\ /\ /\ \#\mid k\ /\ \#\mid$.

L'opération esquissée ci-dessus ne peut par contre pas être réalisée avec un simple transducteur, car elle consiste à déplacer une séquence non-bornée (telle que la liste après i dans notre exemple) vers un autre endroit. Pour résoudre ce problème nous utilisons un simple transducteur τ qui effectue un pas du déplacement. Le résultat attendu est la limite $\tau^*(Conf)$ où $Conf$ est un ensemble régulier de configurations sur lequel l'opération est appliquée. La limite (ou une surapproximation) est calculée en utilisant notre technique d'analyse d'atteignabilité avec abstraction. Pour ne pas confondre avec les autres opérations nous utilisons le mode de calcul spécial s. [1]

Si un marqueur doit être libéré mais cela n'est pas possible, nous allons dans le mode u et arrêtons le calcul. Une telle situation ne peut pas arrivée, si

[1] Le déplacement peut aussi être implémenté avec une opération atomique complexe directement sur les automates, car le résultat d'une telle opération reste régulier.

nous utilisons autant de marqueurs que de variables de pointeurs. Néanmoins, elle peut arriver, si l'utilisateur essaie d'utiliser un nombre plus petit de marqueurs pour réduire le temps de calcul.

Finalement, l'opération `x := new` est facile à coder. Elle introduit une séquence avec un seul élément pointé par x.

Ajouter des données aux éléments d'une liste Le codage peut être facilement étendu pour des données finis. Ces valeurs peuvent être codées dans Σ et une séquence encadrée par / et/ou | ne contient pas seulement les pointeurs qui pointent vers elle mais aussi les valeurs des données.

3.1.2.2 Abstractions

Dans la section 2.3.1.3 nous avons proposé des abstractions sur les automates (langages) qui sont *guidées par la représentation*, c.-à-d. leur principe générale est d'identifier des états équivalents par rapport à une certaine relation qui ne dépend pas du type de configuration représentée du système à analyser. Pour les listes, nous considérons des abstractions *guidées par la configuration*. Nous proposons des schémas généraux pour définir des familles d'abstractions de ce type. Nous adaptons d'abord la notion d'abstraction aux langages. Une *abstraction de langage* est une fonction $\alpha : 2^{\Sigma^*} \to 2^{\Sigma^*}$ telle que $\forall L \in 2^{\Sigma^*}.\ L \subseteq \alpha(L)$. Une abstraction de langage α' *raffine* (ou est un *raffinement* de α) si $\forall L \in 2^{\Sigma^*}.\ \alpha'(L) \subseteq \alpha(L)$. Une abstraction de langage α est *finie* si l'ensemble $\{L \in 2^{\Sigma^*}\ :\ \exists L' \in 2^{\Sigma^*}.\ \alpha(L') = L\}$ est fini. Nous appelons une fonction d'abstraction *régulière*, si elle peut être définie par un transducteur fini.

Abstractions de parties par abstraction de comptage 0-k Nous considérons une décomposition finie de chaque mot et appliquons ensuite l'abstraction de comptage 0-k (qui perd de l'information sur l'ordre des symboles et garde uniquement l'information sur le nombre d'occurrences jusqu'à k) sur chaque partie du mot.

Formellement, pour $w \in \Sigma^*$, soit $dec(w) = (a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$ tel que $w = a_1 w_1 a_2 w_2 \cdots a_n w_n$, $\forall i, j \in \{1, \ldots, n\}.\ a_i \in \Sigma$ et $a_i \neq a_j$, et $\forall i \in \{1, \ldots, n\}.\ w_i \in \{a_1, \ldots, a_i\}^*$. Intuitivement, $dec(w)$ correspond à la décomposition unique de w par rapport à la première occurrence dans w de chacun des symboles de Σ.

Étant donnés un mot w et un symbole a, soit $|w|_a$ le nombre d'occurrences de a dans w. Soit $k \in \mathbb{N}^{>0}$. Nous définissons une fonction α_k des mots vers les langages telle que pour chaque $w \in \Sigma^*$, si $dec(w) = (a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$, alors $\alpha_k(w) = a_1 L_1 a_2 L_2 \cdots a_n L_n$ où $\forall i \in \{1, \ldots, n\}.\ L_i = \{u \in \{a_1, \ldots, a_i\}^* \ :\ \forall j \in \{1, \ldots, i\}.\ |w_i|_{a_j} < k$ et $|u|_{a_j} = |w_i|_{a_j}$, ou $|w_i|_{a_j} \geq k$ et $|u|_{a_j} \geq k\}$. Nous généralisons α_k des mots vers les langages et obtenons une abstraction de langage. On peut facilement montrer :

Proposition 3.1 *Pour chaque $k \geq 0$, α_k est régulier.*

Clairement, pour chaque alphabet Σ donné, l'ensemble des abstractions 0-k est fini et donc le nombre d'abstraction de parties est aussi fini.

Proposition 3.2 *Pour chaque $k \in \mathbb{N}$, l'abstraction α_k est finie.*

Nous considérons une généralisation du schéma ci-dessus obtenu comme suit. Nous autorisons que les décompositions soient calculées uniquement par rapport aux premières occurrences d'un *sous-ensemble* de l'alphabet, appelé *symboles de décomposition*. Par ailleurs, nous autorisons que l'abstraction ne concerne pas certains symboles, appelés *symboles forts*, c.-à-d. toutes leurs occurrences sont préservées par l'abstraction à leurs positions originelles dans le mot. Typiquement, les symboles forts sont ceux dont on sait qu'ils ont un nombre borné d'occurrences dans tous les mots considérés. Par exemple, dans les mots qui correspondent à des configurations de programmes, les symboles forts sont les marqueurs, les séparateurs, et les variables de pointeurs.

Formellement, soient $\Sigma_1, \Sigma_2 \subseteq \Sigma$ deux ensembles de symboles tels que $\Sigma_1 \cap \Sigma_2 = \emptyset$, où Σ_1 est l'ensemble des symboles de décomposition et Σ_2 l'ensemble des symboles forts. (Notons qu'il peut y avoir des symboles qui ne sont ni dans Σ_1 ni dans Σ_2.) Étant donné $w \in \Sigma^*$, nous définissons $dec(w)$ comme la décomposition $(a_1, w_1, a_2, w_2, \cdots, a_n, w_n)$ telle que (1) $w = a_1 w_1 a_2 w_2 \cdots a_n w_n$, (2) $\forall i \in \{1, \ldots, n\}.\ a_i \in \Sigma_1 \cup \Sigma_2$ et, $a_i \in \Sigma_1 \Rightarrow |a_1 a_2 \cdots a_n|_{a_i} = 1$, et (3) $\forall i \in \{1, \ldots, n\}.\ w_i \in (\{a_1, \ldots, a_i\} \setminus \Sigma_2)^*$. Pour chaque k, l'abstraction α_k est alors définie comme précédemment.

La proposition précédente est toujours vraie si le nombre d'occurrences de chaque symbole fort est borné. Tout ensemble de mots L tel que $\forall w \in L.\ \forall a \in \Sigma_2.\ |w|_a \leq p$ est appelé un langage p-Σ_2-borné.

Proposition 3.3 *Pour chaque borne $p \geq 0$, et chaque $k \in \mathbb{N}$, l'abstraction α_k est finie quand elle est appliquée sur un langage p-Σ_2-borné.*

Il est facile de voir que chaque schéma introduit ci-dessus définit une famille d'abstractions raffinables.

Proposition 3.4 *Pour chaque langage p-Σ_2-borné L, et chaque $k \geq 0$, nous avons $\alpha_{k+1}(L) \subseteq \alpha_k(L)$. De plus, si L est infini, alors $\alpha_{k+1}(L) \subsetneq \alpha_k(L)$.*

Abstractions de clôture Nous introduisons ici une autre famille d'abstractions régulières. L'idée est d'appliquer itérativement des règles d'extrapolation qui peuvent être vues comme des règles de réécriture remplaçant des mots de la forme u^k (pour un mot donné u et un entier positif k) par le langage $u^k u^*$.

Soit $u \in \Sigma^*$ et soit $k \in \mathbb{N}^{>0}$. Une relation $R \subseteq \Sigma^* \times \Sigma^*$ est une *règle d'extrapolation* par rapport à la paire (u, k) si $R = \{(w, w') \in \Sigma^* \times \Sigma^* : w = u_1 u^k u_2$ et $w' \in u_1 u^k u^* u_2\}$. Un *système d'extrapolation* est une union finie de règles d'extrapolation.

Clairement, pour chaque langage L, nous avons $L \subseteq R(L)$ (c.-à-d. R définie une abstraction de langage). En fait, nous sommes intéressés par des abstractions qui sont le résultat d'*itération de système d'extrapolation*. Nous définissons donc une *abstraction de clôture* comme la fermeture réflexive et transitive R^* d'un système d'extrapolation R.

Il est clair que chaque système d'extrapolation correspond à une relation régulière (c.-à-d. définissable par un transducteur fini). En général il n'est pas connu, si les abstractions de clôture sont régulières. Dans [26] nous donnons des conditions sur les systèmes d'extrapolation qui garantissent la régularité.

Les abstraction de clôture ne sont pas fini en général. Considérons par exemple, la famille infinie de langages $L_n = (ab)^n$ pour $n \geq 0$ et la règle d'extrapolation R avec $U = \{a\}$ et $k = 1$. Les images des langages donnent alors une famille infinie de langages définie par $R^*(L_n) = (a^+b)^n$ pour chaque $n \geq 0$.

Néanmoins, en pratique [26] le calcul abstrait du regular model-checking termine. Nous montrons aussi dans [26] que le schéma d'abstraction ci-dessus est raffinable. Les abstractions introduites dans cette section permettent une amélioration des temps de calcul par rapport aux abstraction générales (voir [26]).

3.1.3 Traduire vers des automates à compteurs

Dans [22], nous donnons une méthode permettant de vérifier des programmes travaillant avec des listes en passant par les automates à compteurs. Ici, nous esquissons cette méthode et nous donnons un exemple de traduction. La première observation (déjà faite dans la section précédente) est que si nous ne considérons pas les parties de la mémoire qui sont inaccessibles à partir des variables des pointeurs, le graphe qui décrit la mémoire est une collection finie de graphe d'une forme spéciale proche d'un arbre : Il est un ensemble d'arbres (où les arêtes sont dirigées vers la racine) ou d'arbres dont les racines sont connectées à un cycle simple. Le nombre de tels graphes est infini, mais il est facile de montrer que le nombre de nœuds pointé directement par plusieurs pointeurs (nœuds de partage) est borné par le nombre de variables de pointeurs du programme.

Une abstraction naturelle pour les programmes qui ne sont pas sensibles aux données (comme le renversement d'une liste) est d'associer à chaque séquence d'éléments entre deux nœuds de partage une séquence abstraite d'une certaine taille fixé. Néanmoins cette abstraction n'est pas précise en générale. Pour définir une abstraction précise nous avons besoin de raisonner sur la taille exacte des séquences entre les nœuds partagés. Cela nous amène à utiliser des compteurs dans notre modèle abstrait et d'utiliser des automates à compteurs comme modèle abstrait.

Considérer des modèles à base d'automates à compteurs a plusieurs avantages. Cela permet de ne pas seulement définir des abstractions précises, mais aussi de traiter des propriétés quantitatives qui dépendent des tailles de certaines parties de la mémoire. Nous pouvons traiter des programmes avec des variables entières dont les valeurs sont liées d'une certaine façon quantitative aux contenus des listes (c.-à-d. par exemple à leur longueur). Par ailleurs, cela permet une façon puissante de vérifier la terminaison qui nécessite typiquement un raisonnement sur des valeurs décroissantes (par exemple, la taille d'une partie de liste à traiter).

Nous montrons qu'on peut définir une fonction d'abstraction de programmes insensitifs aux données vers les automates à compteurs de sorte que le programme et l'automate à compteurs sont *bisimilaires*. Ce résultat signifie que notre abstraction préserve toutes les propriétés de la classe de programmes insensitifs aux données. Les états de contrôle de l'automate à compteurs construit correspond à des formes de mémoire abstraite (graphes de mémoire où les séquences entre deux nœuds de partage sont réduites à

une arête) et chaque transition correspond à l'exécution d'une opération du programme. Elle représente une modification de la forme abstraite avec une modification des compteurs correspondants. Ces compteurs sont attachés aux arêtes qui correspondent aux séquences abstraites entre deux nœuds de partage.

Les structures de contrôle des automates à compteurs construits sont arbitraires en général. Néanmoins ils ont une propriété importante : si on considère l'évolution de la somme de tous les compteurs, l'effet d'exécuter chaque boucle de contrôle est d'incrémenter la somme par une constante qui dépend du programme. Ce fait peut être utilisé pour montrer un résultat de décidabilité : pour chaque programme insensitifs aux données, si la structure de contrôle de l'automate à compteurs correspondant généré n'a pas de boucle imbriquée (est plat), alors le problème de vérification de propriété de sûreté et de terminaison est décidable.

Nous considérons aussi une extension aux programmes avec des données sur un domaine potentiellement infini avec une relation d'ordre et nous supposons que la seule opération autorisée sur les données est la comparaison par rapport à cette relation d'ordre. Cette classe de programme contient entre autres les tris. Nous étendons le principe d'abstraction sur les graphes de mémoire de ces programmes en tenant compte de certaines informations sur l'ordre des éléments des séquences abstraites entre deux nœuds de partage (comme par exemple le fait que la séquence est triée ou que le premier élément de la séquence est plus grand que le dernier élément d'une autre séquence, etc.).

L'automate à compteurs produit peut être vérifié avec tous les outils existants pour ce type de modèle comme notre regular model-checking avec abstraction de la section 2.3.1. Pour la terminaison on peut utiliser par exemple l'outil de [46].

3.1.3.1 Exemples

Dans la figure 3.4 nous montrons l'automate à compteurs qui correspond au programme de renversement de liste de la figure 3.2 avec initialement une liste non-circulaire pointée par i comme entrée. Les variables de compteur qui correspondent à chaque nœud abstrait sont données à l'intérieur de chaque nœud. L'automate à compteurs pour le même programme qui travaille sur une entrée circulaire est montré dans la figure 3.5. Nous indiquons uniquement les états de contrôle où un branchement se passe.

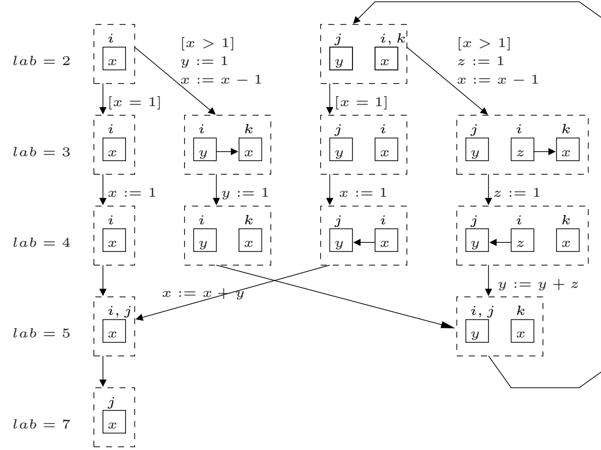


Fig. 3.4 – Renversement de liste non-circulaire

3.2 Programmes avec structures de données arborescentes

Nous considérons ici les programmes qui manipulent des structures de données liées dynamiquement avec *plusieurs* sélecteurs (pointeurs) avec des données d'un domaine fini. Nous visons la vérification des propriétés de bases (pas d'affectations de pointeurs nulls, pas d'utilisation de pointeurs indéfinis, pas de références à des cellules de mémoire libérées) et des invariants simples de forme de mémoire dont la violation peut être exprimée dans un fragment existentiel d'une logique de premier ordre sur les graphes. Nous formalisons ce fragment pour spécifier des mauvais motifs de mémoire. Les formules de ce fragment peuvent être traduites vers de testeurs écrit en pseudo-C qui peuvent être attachés au programme. Ainsi nous réduisons le problème de vérification au problème d'atteignabilité d'une ligne de contrôle (erreur) du programme. Nous codons les graphes de mémoire avec des automates d'arbre étendus et nous représentons les opérations du programme comme des transducteurs d'arbres. Nous pouvons ensuite utilisé le cadre du regular model-checking avec abstraction sur les arbres décrit dans la section 2.3.1.4.

3.2.1 Programmes

Nous considérons une variante des programmes présentés dans la section 3.1 pour les listes. Nous ne considérons pas les variables entiers. Par

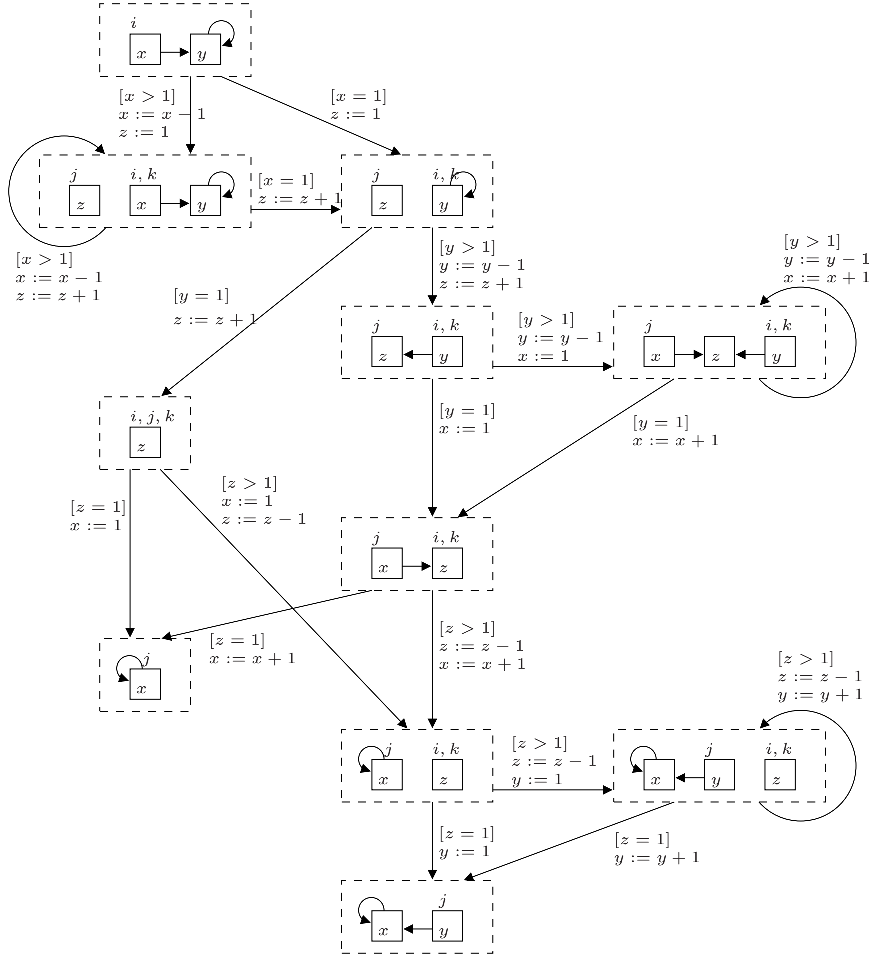


FIG. 3.5 – Renversement de liste circulaire

contre nous autorisons plusieurs pointeurs *next*. Des données d'un domaine fini peuvent aussi être traitées. Nous avons les opérations suivantes : `x=NULL`, `x=y`, `x = y->next`, `x->next = y`, `x = malloc()`, `free(x)`, et `if (x==y) goto L1 ; else goto L2 ;` pour des variables de pointeurs `x` et `y` et des étiquettes de lignes de programme `L1` et `L2`. Un exemple d'un tel programme est donné dans la figure 3.6.

```
// Listes doublement chaînées
typedef struct {
    DLL *next, *prev ;
} DLL ;

DLL *DLL_reverse(DLL *x) {
    DLL *y,*z ;
    z = NULL ;
    y = x->next ;
    while (y !=NULL) {
        x->next = z ;
        x->prev = y ;
        z = x ; x = y ;
        y = x->next
    }
    return x ;
}
```

FIG. 3.6 – Renverser une liste doublement chaînée

3.2.2 Les propriétés

Mise à part les propriétés de base de consistance des manipulations de pointeurs, nous voulons vérifier des propriétés d'invariance de forme de mémoire, telles que l'absence de partage, la non-circularité, ou par exemple le fait que si `x->next == y` (et `y` n'est pas null) dans une DLL, alors aussi `y->prev == x`, etc.). Pour définir des telles propriétés nous proposons deux approches.

3.2.2.1 Testeurs de forme

Premièrement, nous utilisons les *testeurs* écrit en pseudo-C. Ces testeurs peuvent être vus comme du code d'instrumentation qui essaie de détecter des violations des propriétés de forme de mémoire à des locations choisies dans le programme à vérifier. Nous étendons légèrement le langage C pour pouvoir suivre des pointeurs à l'envers et nous autorisons le branchement non-déterministe. Pour notre méthode de vérification le testeur fait partie du programme à vérifier. Une erreur est signalée si une location étiquetée par

erreur est atteinte. De cette façon nous pouvons vérifier toute une gamme de propriétés (voir un exemple dans la figure 3.7).

```
x = aDLLHead ;
while (x != NULL && random())
   x = x->next ;
if (x != NULL
       && x->next->prev != x)
   error() ;
```

FIG. 3.7 – Tester la consistance des pointeurs next et prev

3.2.2.2 Une logique de mauvais motifs de mémoire

Deuxièmement, pour définir les violations des invariants de forme de mémoire d'une façon logique, nous proposons le formalisme LBMP (logic of bad memory patterns), une logique de mauvais motifs de mémoire. Cette logique est un fragment existentiel d'une logique de premier ordre sur les graphes avec des prédicats d'atteignabilité et une quantification existentielle implicite sur les chemins. Au lieu de donner la syntaxe et sémantique complète (voir [28]) nous donnons ici quelques exemples de la logique qui l'illustrent. Ces exemples sont des situations à éviter quand on manipule des listes doublement chaînées acycliques. Il est indésirable que les situations suivantes arrivent après avoir fait une opération sur une liste (par exemple, le renversement). Nous supposons que la liste est pointé par l. Dans les formules suivantes p correspond à la position courante dans un graphe.

1. la liste ne se termine pas avec null :
   $l \xrightarrow{n}^{*} [p = \top]$,
2. le prédécesseur du premier élément de la liste n'est pas null :
   $l[\neg(p \xrightarrow{b} \bot)]$,
3. le prédécesseur d'un successeur d'un nœud n n'est pas n :
   $l \xrightarrow{n}^{*} [\exists x.\ p \xrightarrow{n} x \ \wedge \ x \neq \bot \ \wedge \ \neg(x \xrightarrow{p} p)]$, ou
4. la liste est cyclique
   $\exists x.\ l \xrightarrow{n}^{*} [p = x] \xrightarrow{n}\xrightarrow{n}^{*} [p = x]$.

Nous montrons dans [28] comment traduire une formule de la logique vers un testeur.

Le problème de vérification Les deux approches de spécification présentées ci-dessus permettent d'avoir comme problème de vérification uniquement l'atteignabilité d'une location du programme. Plus précisément, pour montrer qu'un programme est correct il suffit de montrer qu'une location erreur n'est pas atteignable.

3.2.3 Appliquer le regular model-checking sur les arbres avec abstraction

Nous esquissons ici le codage des configurations de mémoire comme des arbres permettant l'application des méthodes de vérification décrites dans la section 2.3.1.4. Les configurations de mémoire des programmes considérés avec un ensemble fini de variables de pointeurs $\mathcal{V}$, un ensemble fini de sélecteurs $\mathcal{S} = \{1, ..., k\}$, et un domaine fini $\mathcal{D}$ de données stockées dans des cellules de mémoire dynamiquement allouées peuvent être décrites comme des graphe de forme de mémoire (shape graphs) de la façon suivante. Un *graphe de forme de mémoire* est un quadruple $SG = (N, S, V, D)$ où N est un ensemble fini de cellules (nœuds) de mémoire, $N \cap \{\bot, \top\} = \emptyset$ (nous utilisons $\bot$ pour représenter null et $\top$ pour représenter une valeur de pointeur indéfinie) $N_{\bot,\top} = N \cup \{\bot, \top\}$, $S : N \times \mathcal{S} \to N_{\bot,\top}$ est la fonction de successeur, $V : \mathcal{V} \to N_{\bot,\top}$ est une application qui définie vers où les variables pointeurs pointent actuellement, et $D : N \to \mathcal{D}$ définie quelles données sont stockées dans les nœuds de mémoire. Nous supposons $\top \in \mathcal{D}$—la valeur $\top$ est utilisée pour représenter des nœuds effacés que nous gardons pour détecter toutes les tentatives de les accéder.

Pour pouvoir représenter des graphes de forme de mémoire plus généraux que les arbres, nous n'identifions pas simplement les pointeurs next avec les branches d'un arbre accepté par un automate d'arbre. Nous utilisons plutôt la structure d'arbre juste comme un squelette sur lequel les liens entre les nœuds de mémoire sont exprimés en utilisant des *expressions de routage*, qui sont des expressions régulières sur les directions dans un arbre (comme monter dans l'arbre, descendre à gauche, etc.) et sur les nœuds visités. À partir des nœuds des arbres décrit par un automate d'arbre, on peut faire référence aux expressions de routage en utilisant des noms symboliques, appelés *descripteurs de pointeur*. Nous supposons de travailler avec un nombre fini des ces descripteur $\mathcal{R}$. Par ailleurs, nous associons à chaque descripteur de pointeur un *marqueur* unique d'un ensemble $\mathcal{M}$ (et donc $||\mathcal{R}|| = ||\mathcal{M}||$).

Les expressions de routage peuvent identifier plusieurs nœuds de mémoire cibles pour un nœud de mémoire et un sélecteur au départ. Les marqueurs permettent de diminuer ce non-déterminisme de la description (uniquement les nœuds marqués avec le marqueur correct sont considérés comme cible). Nous montrons un exemple d'un codage dans la figure 3.8, qui représente une liste doublement chaînée. Nous utilisons deux marqueurs M_1 et M_2, et deux descripteurs D_1 et D_2. Les sélecteurs sont 1 et 2. Le descripteur D_1 indique de descendre dans l'arbre tandis que le descripteur D_2 indique de remonter.

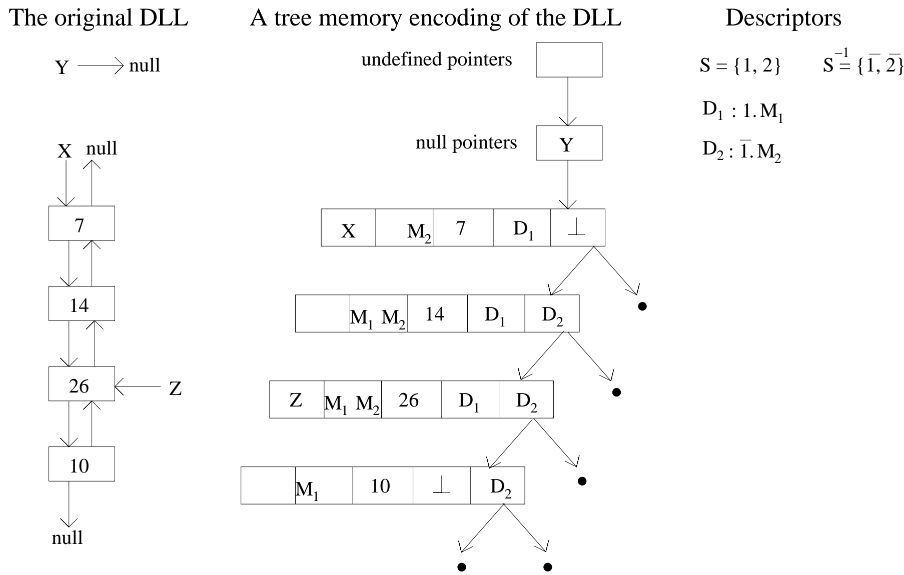


FIG. 3.8 – Un exemple d'un codage d'une mémoire par un arbre— une liste doublement chaînée (DLL)

3.2.3.1 Coder les opérations d'un programme comme des transducteurs

Comme pour les listes, chaque opération du programme est codée comme un transducteur. Dans le cas des arbres (qui représentent des graphes) plusieurs difficultés apparaissent : Un graphe peut avoir plusieurs codage par un arbre. Nous choisissons de ne pas imposer un codage canonique. Néanmoins, si un graphe SG est représenté par un arbre et SG est transformé par une

opération vers un arbre SG' alors le transducteur transforme l'arbre de sorte qu'il représente SG'. Pour cela il faut tenir compte des expressions de routage. Tous les détails du codage se trouvent dans [28].

3.2.3.2 Structures d'entrée

Nous considérons deux possibilités de coder les structures d'entrée. D'une part nous pouvons directement donner un codage des graphes de mémoire d'entrée comme des arbres, par exemple pour toutes les listes doublement chaînées en utilisant les expressions de routage adéquates. Un tel codage peut être donné manuellement ou dérivé automatiquement d'une description de la structure de donnée traitée comme par exemple un graph type [81]. L'avantage est que le processus de vérification commence avec un codage exacte de toutes les instances possibles de la structure de donnée considérée.

Une autre approche est de commencer avec un graphe de mémoire vide où toutes les variables sont indéfinies. Ensuite, nous supposons que l'ensemble des graphes de mémoires sur lequel le programme devrait être vérifié est généré par un *constructeur écrit en C* par l'utilisateur (comme, par exemple dans la figure 3.9). Ce constructeur est mis avant la procédure à vérifier et le model-checker est lancé. L'avantage est qu'aucune notation supplémentaire est nécessaire. Le désavantage est que plus de code doit être vérifié ce qui peut ralentir le processus de vérification.

```
aDLLHead = malloc() ;
aDLLHead->prev = null ;
x = aDLLHead ;
while (random()) {
  x->next = malloc() ;
  x->next->prev = x ;
  x = x->next ;
}
x->next = null ;
```

Fig. 3.9 – Générer des DLLs

3.2.3.3 Appliquer ARTMC

Pour appliquer l'approche du regular model-checking sur les arbres avec abstraction (ARTMC), la difficulté principale est le fait que ARTMC utilise des arbres, qui ont dans notre cadre une sémantique en terme de graphe de mémoire. Un langage d'arbre représente un ensemble de graphe de mémoire. Il est possible que le langage d'arbre n'est pas vide mais que l'ensemble de graphe correspondant est vide (l'inverse n'est pas possible). Nous avons donc uniquement un test du vide approché. Si on choisit une abstraction fini α, alors le calcul abstrait s'arrête, car le nombre d'expressions de routage est borné. Notre méthode a été implémenté dans un outil disponible [11] qui est basé sur la libraire d'automate MONA [80]. Une version légère de l'outil a été implémenté basée sur TIMBUK [115]. Cet outil implémente la version de ARTMC basée sur les automates non-déterministes (voir section 2.4.1) Nous avons obtenu des résultats intéressants pour plusieurs programmes non-triviaux qui manipulent des structures de données arborescentes (comme l'algorithme Deutsch-Schorr-Waite), les listes doublement chaînées, etc.

3.2.4 La terminaison

Dans [64] nous donnons une méthode pour vérifier la terminaison de programmes sur les structures de données arborescentes. Cette méthode est basée sur une boucle abstraire-tester-raffiner. Nous utilisons ARTMC pour obtenir des invariants du programme. Ensuite, nous traduisons le programme vers un automate à compteurs qui le simule. Si la terminaison de l'automate à compteurs peut être montrée en utilisons des techniques dédiées, alors le programme termine aussi. Sinon, nous analysons le contre-exemple qui est peut-être donné par l'outil de terminaison d'automate à compteurs et nous concluons que le programme ne termine pas, ou nous raffinons l'abstraction et recommençons. Nous montrons que le problème de savoir si un contre-exemple en forme de lasso est un vrai contre-exemple est décidable pour quelques cas non-triviaux. Cette méthode a été appliquée a plusieurs études de cas.

3.3 Travaux connexes

Le domaine de la vérification de programmes avec pointeurs est très vaste. Il est impossible de résumer tous ces travaux dans une section. Les approches

les plus utilisées sont la “shape analysis” [108], une méthode basée sur l’interprétation abstraite et l’utilisation d’une logique dédiée pour raisonner sur les structures de mémoire, appelé logique de séparation [106] qui était conçue pour faciliter les preuves à la Hoare. Ces deux techniques peuvent être combinées (voir par exemple [53]). L’idée d’utiliser les automates à compteurs pour représenter des programmes avec listes apparaît aussi indépendamment dans [15] et a été reprise dans [103].

3.4 Perspectives

L’amélioration des méthodes du regular model-checking présentées dans le chapitre 2 profitera directement aux méthodes présentées ici. Nous travaillons également sur une méthode dédiée aux programmes avec plusieurs sélecteurs en utilisant un codage plus simple et on codant directement les opérations des programmes comme des modifications des automates. L’idée est de tirer profit du fait que les opérations du programmes entraînent uniquement des changements locaux dans la structure de la mémoire. Cela devrait être exploité dans les structures de représentation des configurations.

Chapitre 4

Arbres équilibrés

Dans ce chapitre nous considérons la vérification de propriétés non-régulières d'algorithmes sur les arbres, notamment les propriétés d'*équilibre* d'arbres. Par exemple, dont les *arbres rouges et noirs* il y a une propriété qui parle du *nombre* de noeuds rouges et noirs. Ces propriétés ne peuvent pas être directement traitées avec la méthode présentée dans le chapitre précédent, car cette méthode est basée sur les automates d'arbre réconnaissant des langages réguliers d'arbre.

Pour traiter ces propriétés nous introduisons donc une nouvelle classe d'automate d'arbre, appelé TASC (*Tree Automata with Size Constraints*). Les TASC sont des automates d'arbres dont les règles sont conditionnées par des contraintes arithmétiques sur les *tailles* des sous-arbres du nœud considéré. La taille d'un arbre est une valeur numérique définie inductivement sur la structure de l'arbre, par exemple la hauteur ou le nombre de nœuds noirs sur tous les chemins, etc. L'avantage principale des TASC est qu'ils peuvent reconnaître des ensembles non réguliers d'arbres, tel que les arbres AVL, les arbres rouges et noirs et en général des ensembles d'arbres avec des pointeurs vers le parent et qui nécessitent des contraintes arithmétiques sur les longueurs de certains chemins dans l'arbre. Nous montrons que la classe des TASC est close par les opérations d'union, d'intersection et de complément. Leur problème du vide est décidable. De plus, la sémantique des opérations de programmes qui changent l'arbre (comme le changement de couleur, les rotations, etc.) peut être vue comme des changements de la structure des automates.

Dans notre approche de vérification basée sur les TASC, l'utilisateur donne la précondition, la postcondition et les invariants de boucles du pro-

gramme impératif à vérifier. Le problème de vérification est alors de tester la validité de triplets de Hoare de la forme $\{P\}$ **C** $\{Q\}$ où P et Q sont des ensembles de configurations donnés par des TASC qui correspondent à la précondition et à la postcondition du programme ou à un invariant de boucle, et **C** est une partie du programme sans boucle à vérifier. Ce problème est réduit au problème du vide de TASC.

Travaux connexes Plusieurs travaux sur la vérification de programmes qui manipulent des structures arborescentes existent venant de chercheurs de différents horizons, tel que l'analysis statique [101, 107], la théorie des preuves [38], la théorie des langages formels [94, 28]. L'approche qui est la plus proche de la notre est celle de PALE (Pointer Assertion Logic Engine) [94], qui consiste à traduire le problème de vérification vers la logique SkS [105] et à utiliser les automates d'arbre (classiques) pour le résoudre. Notre approche ressemble à PALE puisque nous supposons que l'utilisateur fournisse des pré- et postconditions et les invariants de boucle et nous réduisons le problème de validité de triplet de Hoare au problème du vide d'un langage. Néanmoins, nous pouvons traiter des propriétés quantitatives qui ne sont pas traitées dans PALE.

Dans [107], un cadre spécialisé pour la "shape analysis" quantitative est introduit pour vérifier la manipulation des arbres AVL. Dans [13], la vérification de quelques propriétés de l'insertion dans un arbre rouge et noir est rapportée. Ce travail utilise des systèmes de réécriture de graphes pour l'insertion. Le modèle est construit manuellement. Ensuite, une sur-approximation utilisant des graphes de Petri (des réseaux de Petri avec une structure de graphe supplémentaire) est utilisée pour vérifier que deux nœuds rouges n'apparaissent jamais ensemble. De plus, un système de typage de graphe est utilisé pour la propriété d'équilibre. Ces étapes demandent une intervention de l'utilisateur.

Récemment, [88] a proposé une approche pour vérifier des algorithmes sur les arbres équilibrés (en particulier les arbres rouges et noirs). Elle est basée sur un théorie décidable d'algèbre de termes avec l'arithmétique de Presburger. Cet algèbre permet de définir des fonctions de termes vers les entiers, par exemple le nombre maximal des nœuds noirs dans un chemin de la racine vers une feuille. Par contre, on ne peut pas exprimer des changements locaux à une ligne précise du programme. Dans un autre travail récent [97], un modèle basé sur une extension de la logique de séparation [106] avec des

prédicats de forme définissables par l'utilisateur est utilisé. Ce travail utilise aussi l'approche des triplets de Hoare, mais leurs vérification est effectuée par des règles qui ne sont pas forcément complètes.

La définition des TASC est le résultat de la recherche d'une classe d'automates d'arbres à compteurs qui combine des propriétés intéressantes de fermeture (par union, intersection, complément) avec la décidabilité du problème du vide. Les travaux existants qui étendent les automates d'arbres avec des compteurs (par exemple [48, 95, 85]) concernent plutôt le comptage en *largeur* des nœuds. Nous donnons la possibilité de compter en *profondeur*. Notons que des modèles de calcul similaires, tels que des machines alternantes à multi-bandes ou à compteurs ont un problème du vide indécidable dès qu'on considère deux ou plus de bandes d'entrée à une lettre. Ceci est équivalent à utiliser des compteurs qui ne décroissent pas [102][1]. Néanmoins, restreindre le nombre de compteurs est problématique pour obtenir la clôture des automates par intersection. La solution que nous adoptons ici est de restreindre les opérations sur les compteurs en les rendant dépendants de l'alphabet d'entrée de l'arbre (ce qui revient à les encoder directement dans l'entrée comme des fonctions de taille). Cette solution peut être vue comme une généralisation des langages à pile avec visibilité [7] aux arbres (pour des alphabets de pile d'une lettre). Une approche similaire a été utilisé récemment dans [45], où les automates d'arbre avec visibilité et une mémoire (VTAM) ont été introduits. Les VTAM définissent une sous-classe d'automates d'arbre à une mémoire [42] avec des propriétés de fermeture. Les arbres rouges et noirs et d'autres types d'arbres équilibrés peuvent être reconnus par des automates de ce formalisme. Par contre le travail [45] ne traite pas directement le problème de vérification de programmes manipulant des arbres.

Toutes les preuves se trouvent dans le papier [66] et le rapport technique correspondant [65].

4.1 Une méthodologie de vérification basée sur les TASC

Dans cette section nous introduisons notre méthodologie de vérification pour des programmes utilisant des arbres équilibrés. Comme exemple nous

[1]Ce résultat est une amélioration d'un travail considérant des automates à multi-bandes reconnaissant des langages à une lettre [59].

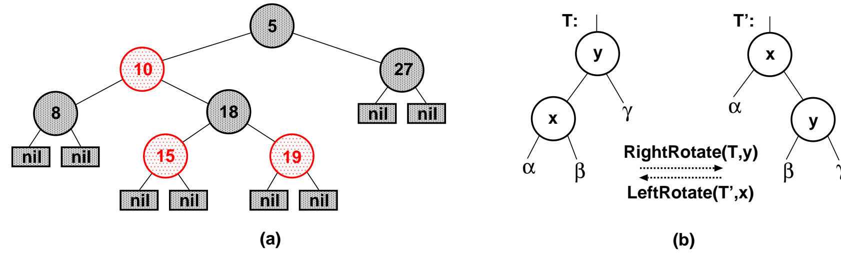


FIG. 4.1 – (a) Un arbre rouge et noir, les nœuds 10, 15, 19 sont rouges (b) la rotation d'arbre gauche et droite

utilisons les *arbres rouges et noirs*. Ce sont des arbres binaires de recherche dont les nœuds sont colorés en rouge ou en noir. Ils sont équilibrés en contraignant la façon dont les nœuds sont colorés. Ces contraintes assurent qu'aucun chemin maximal dans l'arbre peut être plus que deux fois plus grand qu'un autre.

Plus précisément, les arbres rouges et noirs sont des arbres de recherche binaires dont les nœuds contiennent un élément d'un domaine ordonné de données, une couleur, un pointeur gauche et droite, un pointeur vers son parent et qui satisfont les contraintes suivantes :

1. Chaque nœud est soit rouge soit noir.
2. La racine est noire.
3. Chaque feuille est noire.
4. Si un nœud est rouge, alors ses deux enfants sont noirs.
5. Tous les chemins de la racine vers une feuille contiennent le même nombre de nœuds noirs.

Un exemple d'un arbre rouge et noir est donné dans la figure 4.1 (a). Les opérations principales sur les arbres équilibrés (et donc en particulier aussi sur les arbres rouges et noirs) sont la recherche d'une valeur, l'insertion et la suppression. Quand on implémente les deux dernières opérations on doit s'assurer que l'arbre reste équilibré. Cela est typiquement fait en utilisant des rotations d'arbre (voir la figure 4.1 (b)), qui dans le cas des arbres rouges et noirs peut changer le nombre de nœuds noirs sur un chemin donné.

À cause de la dernière condition sur les arbres rouges et noirs (c.-à-d. tous les chemins doivent avoir le même nombre de nœuds noirs), il est évident que

l'ensemble des arbres rouges et noirs n'est pas régulier (reconnaissable par un automate d'arbre standard [43]). Pour cette raison nous introduisons une classe d'automates d'arbre qui permet de décrire des configurations contenant des arbres équilibrés. Cette classe devrait être assez puissante pour décrire ces arbres et en même temps avoir les propriétés nécessaires permettant la vérification automatique (par exemple, la décidabilité de l'inclusion, la fermeture par certaines opérations, etc.).

Ici, nous définissons une telle classe d'automates d'arbre étendus, appelé TASC (tree automata with size constraints). Nous supposons que le contenu des nœuds est abstrait (nous ne vérifions pas, si l'arbre est trié). Les blocs de base des programmes (c.-à-d. les commandes élémentaires ou des groupes de commandes que nous considérons comme atomiques, comme par exemple les rotations) définissent des transformations sur les TASC.

Nous supposons que l'utilisateur spécifie la précondition et la postcondition du programme à vérifier. Nous supposons par ailleurs que l'utilisateur fournisse un invariant pour chaque boucle. Les préconditions, postconditions et les invariants sont donnés par des TASC. Ensuite, la vérification consiste à tester automatiquement la validité de chaque triplet de Hoare $\{P\}$ $\mathbf{C}$ $\{Q\}$, où :

- P est une précondition ou un invariant de boucle.
- Q est une postcondition ou un invariant de boucle.
- $\mathbf{C}$ est une partie du code sans boucle entre P et Q.

Cela est fait en calculant l'image de P après une application du code du bloc $\mathbf{C}$ et en testant que l'image implique Q. Ce test correspond à tester l'inclusion pour du langage d'un TASC dans un autre.

Dans la figure 4.2, nous donnons le pseudo-code de l'opération d'insertion pour les arbres rouges et noirs [47]. Pour ce programme, nous voulons montrer qu'après insertion d'un nœud un arbre rouge et noir reste un arbre rouge et noir. Nous nous restreignons à calculer les effets de bloc de programmes qui préservent la structure d'arbre de la mémoire. Cela n'est pas le cas en général car des opérations sur les pointeurs peuvent changer temporairement la structure arborescente, par exemple dans le code effectuant une rotation. Les opérations que nous traitons sont les suivantes :

1. tester la structure de l'arbre
   (comme, par exemple `x->parent == x->parent->parent->left`),
2. changer la valeur d'une donnée (de domaine fini) d'un nœud
   (comme, par exemple `x->colour = red`),

```
RB-Insert(T,x) :
  Tree-Insert(T,x) ;   % insère une nouvelle feuille x
  x->colour = red ;
  while (x != root && x->parent->colour == red) {
    if (x->parent == x->parent->parent->left) {
      if (x->parent->parent->right->colour == red) {
        x->parent->colour = black ;                    % Cas 1
        x->parent->parent->right->colour = black ;
        x->parent->parent->colour = red ;
        x = x->parent->parent ;
      }
      else {
        if (x == x->parent->right) {                   % Cas 2
          x = x->parent ;
          LeftRotate(T,x) ;
        }
        x->parent->colour = black ;                    % Cas 3
        x->parent->parent->colour = red ;
        RightRotate(T,x->parent->parent) ;
      }
    }
    else .... % le même que ci-dessus avec right et left échangé
  }
  root->colour = black ;
```

Fig. 4.2 – Une procédure pour insérer dans un arbre rouge et noir

3. les rotations gauche et droite (figure 4.1 (b)),

4. déplacer un pointeur vers le haut ou le bas de la structure d'arbre (comme `x = x->parent->parent`),

5. insérer ou supprimer de bas niveau, c.-à-d. insérer/supprimer en mémoire une feuille, ce qui est suivi d'une opération de rééquilibrage.

4.2 Les automates d'arbre avec contraintes de taille

Dans ce qui suit, nous travaillons avec l'ensemble $\mathfrak{D}$ de toutes les combinaisons booléennes de formules de la forme $x - y \diamond c$ or $x \diamond c$, pour un $c \in \mathbb{Z}$ et $\diamond \in \{\leq, \geq\}$. L'égalité est introduite comme abréviation : $x - y = c \; : \; x - y \leq c \wedge x - y \geq c$. La négation peut être éliminée de chaque formule de $\mathfrak{D}$ car $x - y \not\leq c \iff x - y \geq c + 1$. Aussi, chaque contrainte de la forme $x - y \geq c$ peut être écrite d'une façon équivalente comme $y - x \leq -c$. Pour une formule fermée φ, nous écrivons $\models \varphi$ pour indiquer que φ est valide. Nous utilisons, $\top$ pour *true* et $\perp$ pour *false*.

Un *alphabet* Σ muni de rang est un ensemble de symbole avec une fonction $\# : \Sigma \to \mathbb{N}$. Pour $f \in \Sigma$, la valeur $\#(f)$ est appelé le *rang* de f. Les symboles de rang zéro sont appelés *constantes*. Nous écrivons Σ_n pour l'ensemble de tous les symboles de rang n de Σ. Soit λ la séquence vide. Un *arbre* t sur un alphabet Σ est une fonction partielle $t : \mathbb{N}^* \to \Sigma$ qui satisfait les conditions suivantes :

- $dom(t)$ est un sous-ensemble fini de $\mathbb{N}^*$ fermé par préfixe, et
- pour chaque $p \in dom(t)$, si $\#(t(p)) = n > 0$, alors $\{i \mid pi \in dom(t)\} = \{1, \ldots, n\}$.

Un cas spécial d'un alphabet muni de rang est l'*alphabet binaire* dans lequel tous les symboles ont rang zéro ou deux. Les arbres avec alphabet binaire sont appelés *arbres binaires*.

Un *sous-arbre* de t commençant à la position $p \in dom(t)$ est l'arbre $t_{|p}$ défini comme $t_{|p}(q) = t(pq)$ si $pq \in dom(t)$, et indéfini sinon. Nous notons $T(\Sigma)$ l'ensemble de tous les arbres sur l'alphabet Σ.

Une *fonction de taille* (ou mesure) associe à chaque arbre $t \in T(\Sigma)$ un entier $|t| \in \mathbb{Z}$. Les fonction de taille sont définies récursivement sur la structure d'un arbre. Pour chaque $f \in \Sigma$, si $\#(f) = 0$, alors $|f|$ est une constante c_f, sinon, pour $\#(f) = n$, nous avons :

$$|f(t_1, \ldots, t_n)| = \begin{cases} b_1|t_1| + c_1 & \text{si } \models \delta_1(|t_1|, \ldots, |t_n|) \\ \quad \ldots & \\ b_n|t_n| + c_n & \text{si } \models \delta_n(|t_1|, \ldots, |t_n|) \end{cases}$$

où $b_1, \ldots, b_n \in \{0, 1\}$, $c_1, \ldots, c_n \in \mathbb{Z}$, et $\delta_1, \ldots, \delta_n \in \mathfrak{D}$, qui dépendent tous de f. Pour avoir une définition cohérente, $\delta_1, \ldots, \delta_n$ doivent définir une partition de $\mathbb{N}^n$, c.-à-d. $\models \forall x_1 \ldots \forall x_n \; \bigvee_{1 \leq i \leq n} \delta_i(x_1, \ldots, x_n) \wedge \bigwedge_{1 \leq i < j \leq n} \neg(\delta_i(x_1, \ldots, x_n)$

$\wedge\delta_j(x_1, \ldots, x_n))$.[2] Un *t-alphabet* $(\Sigma, |.|)$ est un alphabet muni d'un rang avec une fonction de taille associée.

Exemple 4.1 *La taille d'un arbre binaire est un exemple d'une fonction de taille (mesure). Elle est définie comme* $|c| = 1$ *si* $\#(c) = 0$*, et*

$$|f(t_1, t_2)| = \begin{cases} |t_1| + 1 & \textit{if } |t_1| \geq |t_2| \\ |t_2| + 1 & \textit{if } |t_1| < |t_2| \end{cases}$$

si $\#(f) = 2$. □

Un *automate d'arbre avec contraintes de taille* (TASC) sur un t-alphabet $(\Sigma, |.|)$ est un triplet $A = (Q, \Delta, F)$ où Q est un ensemble fini d'états, $F \subseteq Q$ est un ensemble d'états finaux et Δ est un ensemble de règles de la forme

$$f(q_1, \ldots, q_n) \xrightarrow{\varphi(|1|, \ldots, |n|)} q$$

où $f \in \Sigma$, $\#(f) = n$, et $\varphi \in \mathfrak{D}$ est une formule avec n variables libres. Pour des constantes $a \in \Sigma$, $\#(a) = 0$, l'automate a des règles sans contraintes de la forme $a \to q$.

Un *calcul* de A sur un arbre $t : \mathbb{N}^* \to \Sigma$ est une fonction $\pi : dom(t) \to Q$ qui associe à chaque position de t un état acceptant le sous-arbre à partir de cette position. Formellement, pour chaque position $p \in dom(t)$, où $q = \pi(p)$, nous avons :

- si $\#(t(p)) = n > 0$ et $q_i = \pi(pi)$, $1 \leq i \leq n$, alors Δ a une règle $t(p)(q_1, \ldots, q_n) \xrightarrow{\varphi(|1|, \ldots, |n|)} q$ et $\models \varphi(|t_{|p1}|, \ldots, |t_{|pn}|)$,
- sinon, si $\#(t(p)) = 0$, alors Δ a une règle $t(p) \to q$.

Un calcul π est *acceptant* ssi $\pi(\lambda) \in F$. Comme d'habitude le *langage* de A, noté $\mathcal{L}(A)$ est l'ensemble de tous les arbres pour lesquels A a un calcul acceptant.

Exemple 4.2 *Le TASC suivant reconnaît l'ensemble de tous les arbres rouges et noirs. Soit* $\Sigma = \{red, black, null\}$ *avec* $\#(red) = \#(black) = 2$ *et* $\#(null) =$

[2]Pour des raisons technique liées à la décidabilité du problème du vide pour TASC, nous n'autorisons pas de combinaisons linéaires arbitraires de $|t_i|$ dans la définition de $|f(t_1, \ldots, t_n)|$.

0. *D'abord, nous définissons la fonction de taille comme le nombre maximal de nœuds noirs sur un chemin de la racine vers une feuille :* $|null| = 1$, $|red(t_1, t_2)| = max(|t_1|, |t_2|)$, *et* $|black(t_1, t_2)| = max(|t_1|, |t_2|) + 1$. *Le TASC qui reconnaît l'ensemble de tous les arbres rouges et noirs est défini comme* $A_{rb} = (\{q_b, q_r\}, \Delta, \{q_b\})$ *avec l'ensemble des règles :*

$$\Delta = \{null \to q_b, black(q_{b/r}, q_{b/r}) \xrightarrow{|1| = |2|} q_b, red(q_b, q_b) \xrightarrow{|1| = |2|} q_r\}$$

L'utilisation de $q_{x/y}$ *dans la partie gauche d'une règle signifie que nous prenons toutes les règles où soit* q_x *soit* q_y *prennent la place de* $q_{x/y}$.

Pour des arbres *binaires* nous définissons la notion de *facteur d'équilibrage*. Soit t un arbre binaire et $p \in dom(t)$ une position. Le facteur d'équilibrage de t à p est la différence $|t_{|p0}| - |t_{|p1}|$ entre la taille du sous-arbre gauche et droite de p.

4.3 Propriétés de fermeture et décidabilité de TASC

Cette section est consacrée à la fermeture de la classe des TASC par les opérations d'union, intersection et complément. La preuve de la décidabilité du problème du vide est aussi esquissée.

Un TASC est appelé *déterministe*, si pour chaque arbre d'entrée, l'automate a au maximum un calcul. Pour chaque TASC A nous pouvons construire un TASC déterministe A_d tel que $\mathcal{L}(A) = \mathcal{L}(A_d)$. Pour cela nous adaptons la construction par sous-ensemble classique pour la déterminisation d'automates d'arbres ascendants (bottom-up). Nous devons tenir compte du fait que dans un TASC déterministe deux règles avec la même partie gauche ne devraient pas être applicables simultanément. Ce problème est résolu en introduisant des nouvelles règles dont les gardes sont des conjonctions des gardes de A.

Theorème 4.3 *Pour chaque TASC* A *il existe un TASC* A_d *déterministe avec* $\mathcal{L}(A_d) = \mathcal{L}(A)$.

Soient $A_1 = (Q_1, \Delta_1, F_1)$ et $A_2 = (Q_2, \Delta_2, F_2)$ deux TASC. Nous supposons sans perte de généralité que Q_1 et Q_2 sont disjoints. Soit $A_1 \cup A_2 = (Q_1 \cup Q_2, \Delta_1 \cup \Delta_2, F_1 \cup F_2)$.

Lemma 4.4 *Soit* Σ *un t-alphabet et* $A_i = (Q_i, \Delta_i, F_i)$, $i = 1, 2$, *deux TASC sur* Σ. *Alors,* $\mathcal{L}(A_1 \cup A_2) = \mathcal{L}(A_1) \cup \mathcal{L}(A_2)$.

Un TASC $A = (Q, \Delta, F)$ est appelé *complet* si, pour chaque arbre $t \in T(\Sigma)$ il existe un état $q \in Q$ tel que $t \xrightarrow[A]{*} q$. Un TASC $A = (Q, \Delta, F)$ peut être facilement complété.

Lemma 4.5 *Soit* Σ *un t-alphabet et* $A = (Q, \Delta, F)$ *un TASC sur* Σ. *Alors,* $\mathcal{L}(A_c) = \mathcal{L}(A)$. *Si* A *est déterministe, alors* A_c *l'est aussi.*

Le *complément* d'un TASC $A = (Q, \Delta, F)$ déterministe complet est défini comme $\overline{A} = (Q, \Delta, Q \setminus F)$.

Lemma 4.6 *Soit* Σ *un t-alphabet et* $A = (Q, \Delta, F)$ *un TASC complet et déterministe sur* Σ. *Alors* $t \in \mathcal{L}(A)$ *ssi* $t \notin \mathcal{L}(\overline{A})$ *pour chaque* $t \in T(\Sigma)$.

Puisque nous pouvons construire des automates pour la complémentation et l'union de TASC, on peut définir l'intersection comme $A_1 \cap A_2 = \overline{\overline{A_1} \cup \overline{A_2}}$.

Pour montrer que le problème du vide des TASC est décidable nous remarquons que tous les calculs d'un TASC sont en correspondance directe avec les calculs acceptants d'un automate à pile alternant (APDS) qui peut être construit directement à partir du TASC. L'existence d'un calcul acceptant d'un APDS est un problème décidable bien connu, qui est une conséquence des résultats de [23]. En effet, il est montré qu'étant donné un ensemble régulier C de configurations (paires de la forme $\langle q, w \rangle$, où q est un état de contrôle et w le contenu de la pile), l'ensemble $pre^*(C)$ de toutes les configurations prédécesseurs est aussi régulier et effectivement calculable à partir de C. En particulier l'ensemble $pre^*_q(C) = \{w \mid \langle q, w \rangle \in pre^*(C)\}$ est régulier et effectivement calculable à partir de C. Autrement dit, le APDS a un calcul à partir d'un état de contrôle q_0 vers une configuration de C ssi l'ensemble $pre^*_{q_0}$ n'est pas vide. Puisque le dernier est un ensemble régulier (reconnu par un automate alternant) son problème du vide est décidable. Cela implique la décidabilité du problème du vide pour les APDS. La construction de l'APDS à partir du TASC est assez technique et se trouve dans [65].

Remarque. La décidabilité du problème du vide pour les TASC peut aussi être montrée en utilisant une réduction vers la classe des *automates d'arbres à une mémoire* [42] en codant la taille d'un arbre comme un terme unaire.

Les contraintes d'inégalité des gardes des TASC peuvent être simulées d'une façon analogue en ajoutant des boucles incrément/décrément aux automates d'arbres avec une mémoire.

4.4 La sémantique des opérations

Comme expliqué dans la section 4.1, il y a trois types d'opérations qui sont utilisés couramment dans les procédures pour rééquilibrer les arbres binaires après une insertion ou une suppression : (1) la navigation dans l'arbre, par exemple tester ou changer la position vers laquelle une variable pointeur pointe, (2) tester ou changer certaines données des nœuds, comme leur couleur, et (3) les rotations. De plus, nous devons considérer les insertions/suppressions de bas niveaux qui précédent un rééquilibrage.

Les TASC définis dans la section 4.2 ne sont pas fermés par rapport à l'effet de certaines des opérations mentionnées ci-dessus, notamment celles qui changent le facteur d'équilibrage d'une position. Pour cette raison nous introduisons les *TASC restreints* (rTASC), pour lesquels nous allons montrer la fermeture par toutes les opérations nécessaires sur les arbres binaires. Par ailleurs, les rTASC sont fermés par rapport à l'intersection, l'union, et ils peuvent être déterminisés et minimisés [65]. Par contre, ils ne sont pas fermés par complément. Nous utilisons donc les rTASC pour décrire les invariants de boucle, les pré- et postconditions de programmes ainsi que pour effectuer les calculs d'atteignabilité nécessaires. Les TASC sont ensuite utilisés pour effectuer les tests d'inclusion associés (où ils apparaissent comme des négations de rTASC).

Remarque. Pour simplifier la présentation de l'effet de commandes de programmes sur un ensemble de configurations de mémoire représenté par un rTASC, nous supposons par la suite que les commandes n'amènent pas à une erreur de mémoire (comme une deréférence d'un pointeur null). Néanmoins il est simple d'implémenter des tests pour ces erreurs potentiels de la même manière que les conditions sont implémentées (voir la section 4.4.4).

4.4.1 TASC restreints

Un *t-alphabet restreint* est un t-alphabet contenant uniquement des constantes et des symboles binaires et la fonction de taille est de la forme

$$|f(t_1, t_2)| = \max(|t_1|, |t_2|) + a$$

avec $a \in \mathbb{Z}$ pour des symboles binaires. Un *TASC restreint* est un TASC avec t-alphabet restreint et dont les règles binaires sont de la forme

$$f(q_1, q_2) \xrightarrow{|1| - |2| = b} q$$

avec $b \in \mathbb{Z}$, permettant uniquement de tester des différences constantes entre deux sous-arbres.

Notons que toute conjonction de gardes d'un rTASC et leurs négations se réduit soit à faux soit à une formule de la même forme, $|1| - |2| = b$. Avec cela, il est clair que l'intersection de deux rTASC est un rTASC et que la déterminisation de la section 4.3 appliquée à un rTASC donne un rTASC. En outre, puisque les règles de rTASC contiennent au plus deux variables, pour décider le problème du vide de rTASC il n'est pas nécessaire d'appliquer le pas possiblement cher de la conversion en forme normale comme pour les TASC [65]. L'intersection d'un rTASC avec un automate d'arbre standard est un rTASC, car un automate d'arbre ascendant peut être vu comme un rTASC où toutes les gardes sont vraies. De l'autre coté, il est clair que les rTASC ne sont pas fermés par complément, puisque les inégalités ne sont pas permises comme gardes. Dans [65] nous donnons une procédure très simple de minimisation de rTASC.

4.4.2 Représentation des configurations de mémoire

Avant de décrire comment les rotations (et les autres opérations) peuvent être implémentées sur les rTASC, nous expliquons d'abord comment les rTASC peuvent être utilisés pour représenter des ensembles de configurations de programmes qui manipulent des arbres équilibrés comme les arbres AVL ou les arbres rouges et noirs. Intuitivement, nous représentons des configurations de mémoire (graphes de mémoire) qui ont la forme d'un arbre par des arbres reconnus par des rTASC, où les nœuds sont étiquetés par (1) les variables qui pointent vers eux et (2) les données contenues dans les nœuds. Nous utilisant aussi l'étiquette *null* pour indiqués les successeurs des feuilles.

Formellement, nous considérons un ensemble fini de *variables de pointeurs* $\mathcal{V} = \{x, y, \ldots\}$ et un ensemble fini de valeurs de donnée $\mathcal{D}$, par exemple, $\mathcal{D} = \{red, black\}$. Soit $\Sigma = \mathcal{P}(\mathcal{V}) \times \mathcal{D} \cup \{null\}$. Les rangs sont définis ainsi : $\#(f) = 2$ pour tout $f \in \mathcal{P}(\mathcal{V}) \times \mathcal{D}$, et $\#(null) = 0$. Pour chaque symbole non-null $f \in \mathcal{P}(\mathcal{V}) \times \mathcal{D}$, $v(f) \subseteq \mathcal{V}$ et $d(f) \in \mathcal{D}$ dénotent les variables qui pointent vers le nœud étiqueté par f et la valeur de donnée de ce nœud, respectivement, par exemple, $f = (v(f), d(f))$. Pour un arbre $t \in T(\Sigma)$ et une variable $x \in \mathcal{V}$, un nœud $p \in dom(t)$ est *pointé par* x ssi $t(p) \neq null$ et $x \in v(t(p))$. S'il n'y a pas de nœud pointé par une variable $x \in \mathcal{V}$ dans un arbre $t \in T(\Sigma)$, c.-à-d. $\forall p \in dom(t).\ t(p) \neq null \Rightarrow x \notin v(t(p))$, nous supposons x d'être null.[3]

Pour la suite de la section, soit $A = (Q, \Delta, F)$ un rTASC sur Σ. A *représente* un ensemble de configurations de mémoire ssi pour tout $t \in L(A)$ et tout $x \in \mathcal{V}$, il y a au plus un $p \in dom(t)$ pointé par x. Cette condition peut toujours être forcée en intersectant un rTASC donné par le rTASC $A' = (Q', \Delta', Q')$ où $Q' = \mathcal{P}(\mathcal{V})$, et $\Delta' = \{null \longrightarrow \emptyset\} \cup \{f(v_1, v_2) \xrightarrow{\top} v \mid v = v(f) \cup v_1 \cup v_2 \ \wedge\ v(f) \cap v_1 = v(f) \cap v_2 = v_1 \cap v_2 = \emptyset\}$. Intuitivement, A' se rappelle dans ses états de contrôle de toutes les variables rencontrées jusqu'à présent et vérifie que chaque variable est rencontrée une seule fois au maximum.

Un exemple d'un rTASC qui représente les configurations de mémoire de l'invariant de la procédure d'insertion dans les arbres rouges et noirs est donné au début de la section 4.5.

4.4.3 Calculer l'effet d'une rotation d'arbre

Soient $x \in \mathcal{V}$ une variable et $A = (Q, \Delta, F)$ un rTASC. Nous donnons ici un survol et un exemple d'une méthode pour calculer un rTASC $A' = (Q', \Delta', F')$ qui décrit l'ensemble des arbres qui sont le résultat d'une *rotation gauche* appliqué aux arbres de $L(A)$ au nœuds pointés par x. Le cas d'une rotation droite est similaire.[4] Tous les détails se trouvent dans [65].

Dans la figure 4.3 nous illustrons une rotation gauche sur un rTASC.

Pour la rotation gauche, les règles concernant les nœuds pointés par x et leurs voisins doivent être changées, comme par exemple les règles r_1,r_2 et r_3

[3] Pour simplifier,nous ne distinguons pas explicitement entre null et une valeur de pointeur indéfini. Une telle distinction pourrait être introduite assez simplement.

[4] La rotation droite pourrait être implémentée en échangeant temporairement les enfants des règles impliquées, effectuer une rotation gauche et échanger les enfants encore une foi.

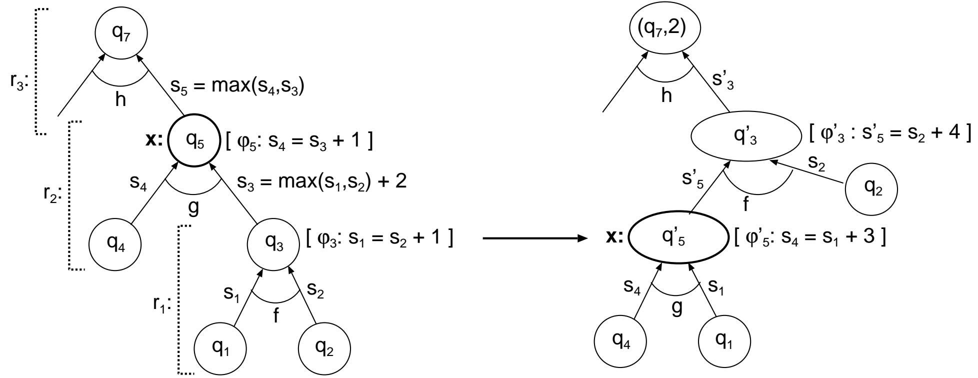


FIG. 4.3 – la rotation gauche sur un rTASC

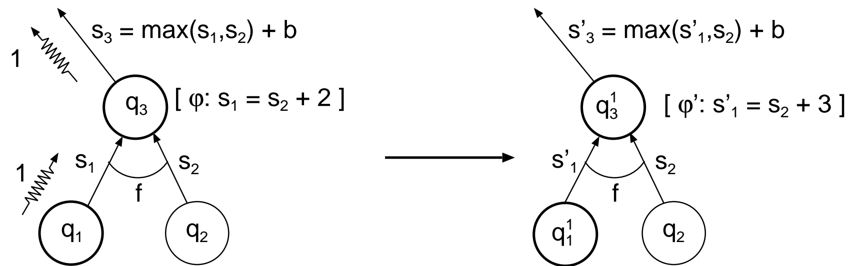


FIG. 4.4 – Propagation de changement des facteurs d'équilibrage dans un rTASC

dans la figure 4.3. D'autres règles doivent être modifiées pour traiter

- les changements de certains états de contrôle, par exemple le changement de q_5 vers q'_3 dans la règle r_3 de la figure 4.3, où
- les changements du facteur d'équilibrage qui résulte de la rotation, c.-à-d. les changements de la différence de taille entre les sous-arbres gauche et droite, qui sont propagés du sous-arbre de la rotation d'une façon ascendante.

Dans la figure 4.4 un exemple de propagation de changement du facteur d'équilibrage est donné.

4.4.4 Les autres opérations

Nous montrons ici que les TASC sont aussi fermés par les autres opérations couramment utilisées avec les arbres binaires équilibrés. Nous avons donné ces opérations dans la section section 4.1. Nous donnons ici uniquement une description informelle, leur formalisation étant simple.

Tester et changer des pointeurs et données. Premièrement, nous considérons l'opération qui consiste à tester si deux expressions sur les pointeurs désignent le même nœud de l'arbre. Des exemples de telles expressions sont `x == root` ou `x->parent->right == x`. En général, nous considérons des tests de la forme e_1`==`e_2 où e_1, e_2 sont de la forme v`->`n_1`->` n_2`->`...n_m avec $v \in \mathcal{V}$, $m \in \mathbb{N}$, et $n_1, ..., n_m \in \{\texttt{left}, \texttt{right}, \texttt{parent}\}$. Soient A un rTASC reconnaissant un ensemble S d'arbres et un test d'égalité de pointeurs c. Alors le rTASC qui décrit le sous-ensemble S' de S des arbres qui satisfont c est l'intersection de A et le TASC A_c qui décrit c. Puisque c décrit un ensemble régulier d'arbre, ce TASC peut être calculé facilement.

Pour illustrer cette construction, nous donnons comme exemple un automate A_c pour la condition `x->parent->right == x`. Rappelons que $\Sigma = \mathcal{P}(\mathcal{V}) \times \mathcal{D} \cup \{null\}$. Alors, $A_c = (Q, \Delta, F)$ est défini par $Q = \{q_1, q_2, q_3\}$, $F = \{q_3\}$, et $\Delta = \{null \to q_1\} \cup \{f(q_1, q_1) \to q_1, g(q_1, q_1) \to q_2, f(q_1, q_2) \to q_3, f(q_3, q_1) \to q_3, f(q_1, q_3) \to q_3 \mid f, g \in \mathcal{P}(\mathcal{V}) \times \mathcal{D}, x \notin v(f), x \in v(g)\}$. Ici, la condition est exprimée par la règle $f(q_1, q_2) \to q_3$.

Deuxièmement, les affectations de pointeurs de la forme v' `=` v`->`n_1`->` n_2`->`...n_m peuvent être implémentées dans notre cadre comme une transformation simple du rTASC qui enlève v' du nœud vers lequel elle pointe et l'ajoute au nœud référencé par v`->`n_1`->`n_2`->`...n_m. Notons que nous ne traitons pas les affectations de la forme v`->`n_1`->`n_2`->`...n_m `=` v'`->`n'_1`->`n'_2`->`...$n'_{m'}$, c.-à-d., les opérations qui changent la structure de la mémoire (*destructive updates*). Ses affectations sont cachées en codant l'effet des procédures entières où elles apparaissent, par exemple dans les rotations ou l'insertion ou la suppression de bas niveau. Ces opérations cassent temporairement la forme de l'arbre en introduisant du partage des nœuds et même des cycles. Nous supposons que la validité des ces opérations est vérifiée indépendamment (par exemple avec nos méthodes du chapitre précédent).

Troisièmement, tester et changer le contenu de donnée d'un nœud pointé par une expression de la forme v`->`n_1`->`n_2`->`...n_m est presque analogue aux tests et aux affectations traiter précédemment. Cependant, changer les don-

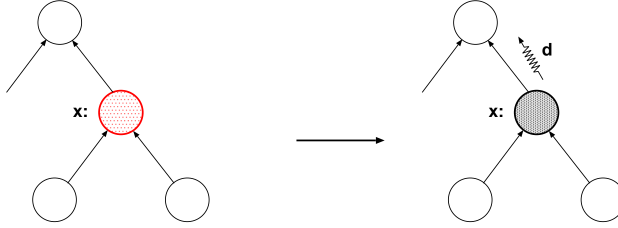


FIG. 4.5 – Changer les données dans un rTASC

nées d'un nœuds (par exemple changer la couleur dans un arbre rouge et noir) peut changer la taille du sous-arbre correspondant. Dans ce cas, les gardes de toutes les règles qui peuvent être utilisées au-dessus du nœud dont la couleur est changé doivent être changées dans la même manière que dans la section 4.4.3 pour tenir compte du changement du facteur d'équilibrage. Dans la figure 4.5 nous donnons un exemple de l'affectation x->$colour = black$.

Insérer des nouveaux nœuds. Concernant l'insertion de bas niveau de nœuds feuilles, nous rappelons que nous supposons que les successeurs nulls de tels nœuds sont explicitement représentés dans notre modèle par des nœuds étiquetés par `null`. Nous ajoutons donc une couche de nœuds supplémentaires par rapport au vrai contenu de la mémoire, car les nœuds nulls ne sont pas alloués dans la vrai mémoire. Insérer un nouveau nœud feuille pointé par une variable de pointeur x (qui est indéfini ou `null` avant) avec une valeur de donnée c revient donc à remplacer un des enfants `null` d'un nœud par un nouveau nœud non-`null` avec deux enfants `nulls`. Nous abstrayons ici de la propriété d'être trié et choisissons un endroit pour insérer une feuille d'une façon arbitraire. L'opération peut être implémentée par une simple transformation d'un rTASC en choisissant non-déterministiquement un nœud `null`, changer sa couleur à $(\{x\}, c)$, et ajouter deux enfants `null`. Ensuite les changements dans le nombre de nœuds marqués par c doivent être propagés en utilisant la même technique que dans la section 4.4.3 (voir la figure 4.6 pour une illustration).

Supprimer des nœuds. Enfin, la suppression d'un nœud pointé par une variable de pointeur y est réalisée en supprimant les règles $(\{y\}, c)(q, q_{\texttt{null}}) \xrightarrow{\varphi} q_y$, où $null \rightarrow q_{\texttt{null}}$ (notons que nous supposons que le nœud à supprimer

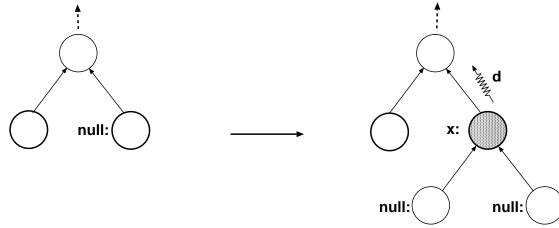


FIG. 4.6 – Insérer un nœuds dans un rTASC

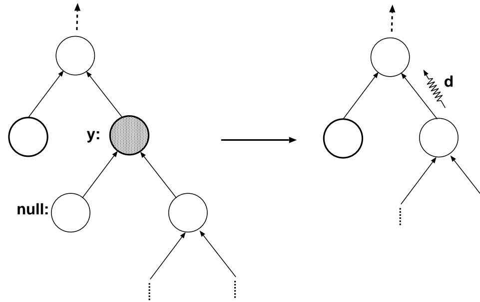


FIG. 4.7 – Enlever un nœud d'un rTASC

a au moins un enfant `null`. Dans les autres règles nous remplaçons simplement toutes les apparences de q_y par tous les états q qui apparaissent dans les règles supprimées. Ensuite, nous utilisons la même technique que dans la section 4.4.3 pour tenir compte des changements des facteurs d'équilibrage qui résultent de la suppression d'un nœud (voir la figure 4.7 pour une illustration).

4.5 Une étude de cas : L'insertion dans un arbre rouge et noir

Pour illustrer notre méthodologie nous montrons comment on peut prouver un invariant de la boucle principale de la procédure RB-Insert (voir la figure 4.2). Toutes les étapes peuvent être faites automatiquement. L'invariant est nécessaire pour prouver que la procédure d'insertion est correcte, c.-à-d. qu'étant donné un arbre rouge et noir comme entrée de la procédure,

la sortie est aussi un arbre rouge et noir. L'invariant est la conjonction des faits suivants :

1. x pointe vers un nœud non-`null` dans l'arbre.
2. Si un nœud est rouge, alors (i) son fils gauche est soit noir soit pointé par x, et (ii) son fils droit est soit noir soit pointé par x. Cette condition est nécessaire, car pendant le rééquilibrage de l'arbre, un nœud rouge peut devenir temporairement un fils d'un autre nœud rouge.
3. La racine est soit noir soit x pointe vers elle.
4. Si x ne pointe pas vers la racine et pointe vers un nœud dont le parent est rouge, alors x pointe vers un nœud rouge.
5. Tous les chemins de la racine vers une feuille contiennent le même nombre de nœuds noirs. C'est la dernière condition de la définition d'un arbre rouge et noir de la section 4.1.

Dans cet exemple nous avons $\mathcal{V} = \{x\}$, $\mathcal{D} = \{red, black\}$, et donc $\Sigma = (\{\emptyset, \{x\}\} \times \{red, black\}) \cup \{null\}$. Pour simplifier nous écrivons à la place de $(\emptyset, c) \in \Sigma$, c, et c_x à la place de $(\{x\}, c) \in \Sigma$, où $c \in \{red, black\}$. Aussi, si aucune garde est spécifiée dans une règle binaire, nous supposons qu'elle est $|1| = |2|$. Soit $R = \{null \to q_b, red(q_b, q_b) \to q_r, black(q_{b/r}, q_{b/r}) \to q_b\}$. L'invariant de boucle est donné par le rTASC A_1 suivant.

$$A_1 : \; F = \{q_{rx}, q_{bx}, q'_{bx}\}, \quad \Delta = R \cup$$
$$\begin{array}{ll}
\{black_x(q_{b/r}, q_{b/r}) \to q_{bx} \; (1), & black(q_{bx/rx}, q_{b/r}) \to q'_{bx} (2) \\
black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \to q'_{bx} \; (3), \\
black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}, & red_x(q_b, q_b) \to q_{rx}, \\
red(q'_{bx}, q_b) \to q'_{rx}, & red(q_b, q'_{bx}) \to q'_{rx}, \\
red(q_{rx}, q_b) \to q'_{rx} \; (4), & red(q_b, q_{rx}) \to q'_{rx} \; (5)\}
\end{array}$$

Intuitivement, q_b étiquette des nœuds noirs et q_r les nœuds rouges qui n'ont pas de nœuds pointé par x au-dessous d'eux. q_{bx} et q_{rx} signifient la même chose sauf qu'ils étiquettent un nœud qui est pointé par x. Les versions avec apostrophe de q_{bx} et q_{rx} sont utilisées pour les nœuds qui ont un descendant pointé par x. Par la suite, cette “sémantique” intuitive va être changée par les opérations du programme. Nous faisons référence au pseudo-code de la section 4.1.

Si la condition de la boucle `x != root && x->parent->color == red` est satisfaite nous obtenons le nouvel automate A_2. Il est donné en modifiant A_1 comme suit : $F = \{q'_{bx}\}$ et les règles (1), (2), et (3) sont enlevées.

Si la condition `x->parent == x->parent->parent->left` est vraie, nous prenons A_2, changeons la règle (4) vers $red(q_{rx}, q_b) \to q''_{rx}$, la règle (5) vers $red(q_b, q_{rx}) \to q''_{rx}$ et ajoutons une règle $black(q''_{rx}, q_{b/r}) \to q'_{bx}$ (6) et obtenons A_3. Maintenant, q''_{rx} accepte le parent d'un nœud pointé par `x` et q'_{rx} son grand-parent.

Si la condition suivante `x->parent->parent->right->color == red` est vraie, alors nous obtenons l'automate A_4 qui est comme l'automate A_3 sauf pour la règle (6) qui est changée vers $black(q''_{rx}, q_r) \to q'_{bx}$.

Le pas `x->parent->color = black` qui change la couleur, fait changer quelques gardes sur les règles et amène à une propagation de ces changements à travers l'automate. Le résultat est A_5 :

$$
\begin{array}{ll}
A_5: \ F = \{q'_{bx}\}, \Delta = R \ \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red_x(q_b, q_b) \to q_{rx}, \\
black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1| + 1 = |2|} q'_{bx}, & red(q'_{bx}, q_b) \xrightarrow{|1| = |2| + 1} q'_{rx}, \\
black(q''_{rx}, q_r) \xrightarrow{|1| = |2| + 1} q'_{bx} \ (7), & red(q_b, q'_{bx}) \xrightarrow{|1| + 1 = |2|} q'_{rx}, \\
black(q_{rx}, q_b) \to q''_{rx}, & black(q_b, q_{rx}) \to q''_{rx}\}
\end{array}
$$

Après le changement de couleur `x->parent->parent->right->color = black`, nous obtenons A_6 qui est A_5 où (7) est changée vers $black(q''_{rx}, q_b) \to q'_{bx}$. Notons qu'ici une propagation n'est pas nécessaire.

Après le changement de couleur `x->parent->parent->color = red`, qui introduit des changements sur les gardes et leur propagation, nous obtenons :

$$
\begin{array}{ll}
A_7: \ F = \{q'_{bx}\}, \Delta = R \ \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \to q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \to q'_{bx}, \\
black(q_{rx}, q_b) \to q''_{rx}, & black(q_b, q_{rx}) \to q''_{rx}, \\
red_x(q_b, q_b) \to q_{rx} \ (8), & red(q'_{bx}, q_b) \to q'_{rx}, \\
red(q''_{rx}, q_r) \to q'_{bx} \ (9), & red(q_b, q'_{bx}) \to q'_{rx}\}
\end{array}
$$

Après `x = x->parent->parent`, nous obtenons A_8 obtenu de A_7 en changeant la règle (8) vers $red(q_b, q_b) \to q_{rx}$ et la règle (9) vers $red_x(q''_{rx}, q_b) \to q'_{bx}$.

Cela termine cas 1 et on peut tester que $\mathcal{L}(A_8) \subseteq \mathcal{L}(A_1)$.

Pour le cas 2, nous devons considérer l'automate A_3 et tenir compte du fait que `x->parent->parent->right->color == red` est fausse, c.-à-d. que

`x->parent->parent->right->color == black` est vraie. Le résultat est :

$$\begin{array}{ll}
A_9: \; F = \{q'_{bx}\}, \Delta = R \; \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, \\
black(q''_{rx}, q_b) \rightarrow q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rx} \; (11), \\
red(q'_{bx}, q_b) \rightarrow q'_{rx}, & red(q_b, q'_{bx}) \rightarrow q'_{rx}, \\
red(q_b, q_{rx}) \rightarrow q''_{rx} \; (12), & red(q_{rx}, q_b) \rightarrow q''_{rx} \; (10)\}
\end{array}$$

Après la condition `x == x->parent->right`, A_9 est changé vers A_{10} en enlevant la règle (10). Après `x = x->parent`, A_{10} est changé vers A_{11} en changeant règle (11) vers $red(q_b, q_b) \rightarrow q_{rx}$ et règle (12) vers $red_x(q_b, q_{rx}) \rightarrow q''_{rx}$.

Après l'opération `Left-Rotate(T,x)` introduit des nouveaux états et transitions et nous obtenons le TASC A_{12}. Remarquons qu'une propagation n'est pas nécessaire.

$$\begin{array}{ll}
A_{12}: \; F = \{q'_{bx}\}, \Delta = R \; \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, & black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, \\
black(q_{rot2}, q_b) \rightarrow q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rot1}, \\
red(q'_{bx}, q_b) \rightarrow q'_{rx}, & red(q_b, q'_{bx}) \rightarrow q'_{rx}, \\
red(q_{rot1}, q_b) \rightarrow q_{rot2}\} &
\end{array}$$

Après `x->parent->color = black` et la propagation des changements, nous obtenons :

$$\begin{array}{ll}
A_{13}: \; F = \{q'_{bx}\}, \Delta = R \; \cup & \\
\{black(q'_{bx/rx}, q_{b/r}) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red_x(q_b, q_b) \rightarrow q_{rot1}, \\
black(q_{b/r}, q'_{bx/rx}) \xrightarrow{|1| + 1 = |2|} q'_{bx}, & red(q'_{bx}, q_b) \xrightarrow{|1| = |2| + 1} q'_{rx}, \\
black(q_{rot2}, q_b) \xrightarrow{|1| = |2| + 1} q'_{bx}, & red(q_b, q'_{bx}) \xrightarrow{|1| + 1 = |2|} q'_{rx}, \\
black(q_{rot1}, q_b) \rightarrow q_{rot2}\} &
\end{array}$$

Après `x->parent->parent->color = red`, nous obtenons :

$A_{14}: \ F = \{q'_{bx}\}, \Delta = R \ \cup$
$\{black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, \qquad red_x(q_b, q_b) \rightarrow q_{rot1},$
$black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}, \qquad red(q'_{bx}, q_b) \rightarrow q'_{rx},$
$red(q_{rot2}, q_b) \xrightarrow{|1| = |2| + 1} q'_{bx}, \quad red(q_b, q'_{bx}) \rightarrow q'_{rx},$
$black(q_{rot1}, q_b) \rightarrow q_{rot2}\}$

Enfin, après `Right-Rotate(T,x->parent->parent)`, nous avons :

$A_{15}: \ F = \{q'_{bx}\}, \Delta = R \ \cup$
$\{black(q'_{bx/rx}, q_{b/r}) \rightarrow q'_{bx}, \qquad black(q_{b/r}, q'_{bx/rx}) \rightarrow q'_{bx}$
$black(q_{b/r}, q_{rot4}) \rightarrow q'_{bx}, \qquad black(q_{rot4}, q_{b/r}) \rightarrow q'_{bx},$
$black(q_{rot1}, q_{rot3}) \rightarrow q_{rot4}, \qquad red_x(q_b, q_b) \rightarrow q_{rot1},$
$red(q'_{bx}, q_b) \rightarrow q'_{rx}, \qquad red(q_b, q'_{bx}) \rightarrow q'_{rx},$
$red(q_{rot4}, q_b) \rightarrow q'_{rx}, \qquad red(q_b, q_b) \rightarrow q_{rot3},$
$red(q_b, q_{rot4}) \rightarrow q'_{rx}\}$

Il peut maintenant être testé que $\mathcal{L}(A_{15}) \subseteq \mathcal{L}(A_1)$. Le cas 3 de la procédure d'insertion est très similaire au cas 2 et il est omis.

4.6 Perspectives

Une possible extension de notre méthode est son automatisation totale. Pour cela, une méthode de génération d'invariants pour les TASC est nécessaire. Une piste possible est d'essayer d'étendre l'approche du regular model checking du chapitre 2 vers les TASC.

Chapitre 5

Programmes avec tableaux

Dans ce chapitre nous résumons nos travaux [69, 68, 33] sur la vérification de programmes et logiques sur les tableaux d'entiers. Les tableaux sont une structure de données fondamentale en informatique. Ils sont utilisés quasiment dans tous les langages de programmation impératifs modernes. Pour vérifier des logiciels qui manipulent des tableaux, il est d'abord essentiel d'avoir des logiques suffisamment puissantes pour exprimer des propriétés intéressantes qui peuvent par exemple apparaître comme des conditions de vérification. Pour permettre une procédure de décision automatique, on a besoin d'une logique décidable. Dans la section 5.2 nous introduisons la logique LIA [69] pour laquelle nous montrons sa décidabilité en passant par les automates à compteurs représentant les modèles de formules. Nous donnons ensuite une méthodologie de vérification automatique de programme avec tableaux [33] qui est basée sur une version allégé de LIA, appelé SIL (introduite dans la section 5.3.4, voir aussi [68]). Cette méthodologie est aussi basée sur les automates à compteurs. D'abord, nous donnons quelques définitions nécessaires.

5.1 Préliminaires

Pour une fonction $f : A \to B$ et un ensemble $S \subseteq A$, nous écrivons $f\downarrow_S$ pour la restriction de f sur S. Cette notation est facilement étendu aux ensembles, paires ou séquences de fonctions. Étant donnée une formule φ, nous écrivons $FV(\varphi)$ pour l'ensemble de ses variables libres. Si nous écrivons $\varphi(x_1, ..., x_n)$, nous supposons $FV(\varphi) \subseteq \{x_1, ..., x_n\}$. Pour $\varphi(x_1, ..., x_n)$, nous

écrivons $\varphi[t/x_i]$, $1 \leq i \leq n$, pour la formule dans laquelle chaque occurrence de x_i est remplacée par le terme t. Pour une formule $\varphi(x_1, \ldots, x_n)$ nous écrivons $\varphi[t_1/x_1, \ldots, t_n/x_n]$ la formule qui obtenue à partir de φ en replaçant chaque occurrence libre de $x_1, \ldots, x_n$ par les termes $t_1, \ldots, t_n$, respectivement. Nous écrivons $\varphi[t/x_1, \ldots, x_n]$ pour la formule obtenue à partir de φ en remplaçant toutes les occurrences de toutes les variables $x_1, \ldots, x_n$ par le même terme t. Étant données une formule φ et une valuation ν de ses variables libres, nous écrivons $\nu \models \varphi$ si en remplaçant chaque variable libre x de φ par $\nu(x)$ on obtient une formule valide. Nous écrivons $\models \varphi$ pour le fait que φ est valide. Par $\sigma : \mathbb{Z} \to \mathbb{Z}$, $\sigma(n) = n + 1$, nous désignons la fonction de successeurs sur les entiers. Dans la suite nous introduisons deux types de formules : Les contraintes de différences et l'arithmétique de Presburger.

Une *contraintes de différence* (DBC) est une conjonction d'inéquations de la forme $x - y \leq c$, $x \leq c$, ou $x \geq c$ où $c \in \mathbb{Z}$ est une constante. S'il n'y a pas de contrainte entre x et y, nous écrivons parfois explicitement $x - y \leq \infty$. Nous écrivons $\top$ (true) pour la DBC vide. Par la suite, $\mathbb{Z}^\infty$ dénote $\mathbb{Z} \cup \{\infty\}$. Il est bien connu que la négation d'une DBC est équivalent à une disjonction finie de DBCs (disjoints de paire à paire) car, par exemple, $\neg(x - y \leq c) \iff y - x \leq -c - 1$ et $\neg(x \leq c) \iff x \geq c + 1$. En particulier, la négation de $\top$ est la disjonction vide, notée comme $\bot$ (false). Soit $\vec{z} = \{z_1, \ldots, z_n\}$ un ensemble fixé de variables, appelées *paramètres*. Une *DBC paramétrique* est une conjonction de formules DBC avec des propositions atomiques de la forme $x \leq f(\vec{z})$ ou $x \geq f(\vec{z})$ où f est une combinaison linéaire de paramètres, c.-à-d. $f = a_0 + \sum_{i=1}^n a_i z_i$ pour certains $a_i \in \mathbb{Z}$, $0 \leq i \leq n$.

Une formule de l'*arithmétique de Presburger* (PA) est une disjonction de conjonction de contraintes linéaires de la forme $\sum_{i=1}^n a_i x_i + b \geq 0$ ou de contraintes modulo $\sum_{i=1}^n a_i x_i + b \equiv c \mod d$ où $a_i, b, c, d \in \mathbb{Z}$, $c \geq 0$ et $d > 0$, sont des constantes. Il est bien connu que chaque formule de l'arithmétique sur les entiers avec addition $\langle \mathbb{Z}, \geq, +, 0, 1 \rangle$ peut être écrite dans cette forme grâce à l'élimination de quantificateurs [104]. Clairement chaque DBC est aussi dans PA.

5.2 La logique LIA

Dans cette section, nous introduisons une logique de tableaux d'entiers indicés par des entiers. Cette logique est appelée LIA (Logique of integer arrays). Le but est d'être le plus général possible en retenant la décidabilité

du problème de satisfaisabilité. Pour éviter de devoir traiter explicitement des accès hors bornes des tableaux nous interprétons les formules sur des tableaux bi-infinis (infinis dans les deux sens). Des tableaux bornés peuvent être définis dans la logique en restreignant explicitement les indices d'être entre deux bornes données.

Les propriétés intéressantes sur les tableaux d'un programme sont typiquement des combinaisons booléennes (existentiellement quantifiées) de formules de la forme $\forall \vec{i}.G \rightarrow V$ où G est une *expression de garde* qui contient des contraintes sur les variables d'indices $\vec{i}$ universellement quantifiées (souvent entre deux bornes existentiellement quantifiées) et V est une *expression de valeur* qui contient des contraintes sur les variables tableaux. Deux types de propriétés apparaissent en particulier :

1. des propriétés relatant des éléments consécutifs d'un tableau, par exemple $\forall i \,.\, l_1 \leq i < l_2 \rightarrow a[i+1] = a[i] - 1$, qui exprime le fait que chaque valeur de a entre les deux bornes l_1 et l_2 est plus petit par 1 que son prédécesseur,
2. des propriétés exprimant des faits périodiques, par exemple $\forall i \,.\, i \equiv_2 0 \rightarrow a[i] = 0$, qui dit que tous les éléments pairs d'un tableau a sont égaux à 0.

Une logique avec ce pouvoir d'expression est indécidable sans restrictions syntaxiques. En effet, on peut montrer qu'un calcul d'une machine à deux compteurs [93] correspond aux modèles d'une certaine formule sur des tableaux. De cette réduction, on peut déduire deux restrictions qui amènent à la décidabilité. La première restriction interdit des références à $a[i]$ et $a[i+1]$ dans la même formule, ce qui est considérée dans le travail de Bradley, Manna, et Sipma [36]. La deuxième restriction considérée ici, est d'uniquement permettre des formules de la forme $\forall \vec{i}.G \rightarrow V$ dans lesquelles V ne contient pas de disjonctions.

Nous introduisons une nouvelle logique appelée **LIA** (Logic on Integer Arrays) dans le fragment $\exists^*\forall^*$ de premier ordre. LIA est essentiellement l'ensemble des combinaisons booléennes quantifiées existentiellement de la forme

1. formules de la forme $\forall \vec{i} \,.\, \varphi(\vec{k}, \vec{i}) \rightarrow \psi(\vec{k}, \vec{i}, \vec{a})$, où $\vec{i}$ est un ensemble de variables d'indice et $\vec{a}$ (resp. $\vec{k}$) est l'ensemble des variables de tableau (resp. *bornes de tableau*), φ est une formule sur les variables d'indices $\vec{i}$ (par rapport aux bornes $\vec{k}$) avec des contraintes de différences et des contraintes de périodicité et ψ est une contrainte de différence, et

2. des formules de l'arithmétique de Presburger sur les variables de bornes de tableau.

Nous montrons la décidabilité de la logique LIA en utilisons l'idée classique de la connection entre logique et automates [114] : d'une formule φ de la logique, nous construisons un automate A_φ tel que φ est satisfaisable si et seulement si le langage de A_φ n'est pas vide. La décidabilité de la logique suit ensuite de la décidabilité du problème du vide de la classe d'automate utilisée. Pour cela, nous définissons une nouvelle classe d'automate à compteur, appelé FBCA (bi-infinite Flat Büchi Counter Automata, les automates bi-infinis de Büchi à compteurs). Ce sont des automates à compteurs travaillant à l'infini vers la gauche et vers la droite avec des conditions d'acceptation à la Büchi. Pour une formule quelconque φ de LIA, nous donnons la construction d'un automate FBCA A_φ dont les calculs correspondent aux modèles de φ : la valeur du compteur x_a à un point donné i dans un calcul de A_φ corresponds à la valeur $a[i]$ dans un modèle de φ. Nous montrons la décidabilité de LIA en prouvons que le problème du vide de FBCA est décidable en étendant des résultats existants [44, 35] sur les automates à compteurs plats avec des contraintes de différences.

Toutes les preuves sont données dans [67].

5.2.1 Travaux connexes

Dans le papier fondateur [91], les fonctions read et write de/vers un tableau et leur axiomes logiques ont été introduites. Une procédure de décision pour le fragment sans quantificateurs de la théorie des tableaux a été donnée dans [79]. Depuis, plusieurs autres logiques décidables ont été considérées, par exemple [113, 89, 74, 112, 9, 60, 61]. Certaines de ces logiques contiennent entre autres plusieurs prédicats (pour raisonner sur le caractère triée du tableau, les permutations, etc.) et des contraintes arithmétiques (typiquement de Presburger) sur les indices de tableaux et/ou les valeurs des entrées. Néanmoins, contrairement à notre travail, la plupart de ces logiques considèrent des formules sans quantificateurs. Dans ces cas, des références imbriquées (comme $a[a[i]]$) sont autorisées, ce qui n'est pas le cas de notre logique.

Dans [36], une logique dans le fragment $\exists^*\forall^*$ est introduite. Contrairement à notre procédure de décision qui est basée sur la théorie des automates, la procédure de décision de [36] est basée sur le fait que la quantification uni-

verselle peut être remplacée par une conjonction (finie). Le résultat est paramétré dans le sens d'autoriser différents types de données dans les tableaux pour lesquels une théorie décidable est supposée existée. Comparé avec notre résultat, [36] ne permet ni des contraintes modulo (permettant d'exprimer des faits périodiques), ni les contraintes de différence sur des indices universellement quantifiées (uniquement $i-j \leq 0$ est autorisée), ni de raisonner sur des cellules de tableaux a une distance fixe (comme par exemple raisonner sur $a[i]$ et $a[i+k]$ pour une constante k et un indice universellement quantifié i). Les auteurs de [36] donnent aussi un résultat d'indécidabilité intéressant pour des extensions de leur logique. Par exemple, ils montrent que relier deux valeurs de tableaux adjacentes ($a[i]$ et $a[i+1]$) ou avoir des accès imbriqués amène à l'indécidabilité.

Une forme restreinte de quantification universelle dans le fragment $\exists^*\forall^*$ est aussi autorisée dans [10], où la décidabilité est obtenu en se basant sur une propriété de modèle petit. Contrairement à [36] et à notre travail, [10] permet une forme d'accès imbriqué hiérarchique. Néanmoins, ni des contraintes modulo ni de raisonner sur des cellules de tableaux a une distance fixe est autorisé. Une restriction similaire apparaît aussi dans [25] où une logique assez général dans le fragment $\exists^*\forall^*$ sur des multi-ensemble d'éléments avec des valeurs de données associées est considérée.

5.2.2 Automates à compteurs

Un *automate à compteur* (CA) est un triple $A = \langle \vec{x}, Q, \rightarrow \rangle$ où $\vec{x}$ est l'ensemble fini de compteurs sur $\mathbb{Z}$, Q un ensemble d'état de contrôle, et $\rightarrow$ une relation de transition donnée par des règles $q \xrightarrow{\varphi(\vec{x},\vec{x'})} q'$ où φ est une formule arithmétique reliant les valeurs courantes des compteurs $\vec{x}$ à leurs valeurs futures $\vec{x'}$. Une *configuration* d'un CA A est une paire (q, ν) où $q \in Q$ est un état de contrôle et $\nu : \vec{x} \rightarrow \mathbb{Z}$ est une valuation des compteurs de $\vec{x}$. Pour une configuration $c = (q, \nu)$, nous notons $val(c) = \nu$ la valuation des compteurs dans c. Une configuration (q', ν') est un *successeur immédiat* de (q, ν) ssi A a une transition $q \xrightarrow{\varphi(\vec{x},\vec{x'})} q'$ telle que $\models \varphi(\nu(\vec{x}), \nu'(\vec{x'}))$. Une configuration c est un *successeur* d'une autre configuration c' ss'il existe une séquence de configurations $c = c_0 c_1 \ldots c_n = c'$ tel que pour tout i avec $0 \leq i < n$, c_{i+1} est un successeur immédiat de c_i. Étant donnés deux états de contrôle $q, q' \in Q$, un *calcul* de A de q vers q' est une séquence finie de

configurations $c_0c_1\dots c_n$ avec $c_0 = (q,\nu)$, $c_n = (q',\nu')$ pour des valuations $\nu,\nu' : \vec{x} \to \mathbb{Z}$, et c_{i+1} est un successeur immédiat de c_i pour tout i avec $0 \le i < n$.

Soit S un ensemble. Une *séquence bi-infinie* de S est une fonction $\beta : \mathbb{Z} \to S$.[1] Nous notons ${}^{\omega}S^{\omega}$ l'ensemble de toutes les séquences bi-infinies de S. Un *automate de Büchi bi-infini à compteurs* (BCA) est un quintuplet $A = \langle \vec{x}, Q, L, R, \to\rangle$ où $\vec{x}$ est un ensemble fini de compteurs, Q un ensemble fini d'états de contrôle, $L, R \subseteq Q$ sont les états acceptants gauches et droits, et $\to$ est une relation de transition définie de la même manière que pour les automates à compteurs.

Un *calcul* d'un BCA A est une séquence bi-infinie de configurations $\dots c_{-2}c_{-1}c_0c_1c_2\dots$ telle que, pour tout $i \in \mathbb{Z}$, c_{i+1} est un successeur immédiat de c_i. Un calcul r est *acceptant gauche* ss'il existe un état $q \in L$ et une séquence d'entiers infinie décroissante $\dots < i_2 < i_1 < 0$ telle que, pour tout $j \in \mathbb{N}$, nous avons $r(i_j) = (q, \nu_j)$ pour des valuations ν_j des compteurs de A. Symétriquement, un calcul est *acceptant droit* ss'il existe un état $q \in R$ et une séquence d'entiers infinie croissante $0 < i_0 < i_1 < i_2 < \dots$ telle que, pour tout $j \in \mathbb{N}$, nous avons $r(i_j) = (q, \nu_j)$ pour des valuations ν_j des compteurs de A. Un calcul est *acceptant* ss'il est acceptant gauche et acceptant droit. L'ensemble de tous les calculs de A est noté par $\mathcal{R}(A)$. Si $r \in \mathcal{R}(A)$ est un calcul de A, nous définissons $val(r) = \dots val(r(-1))val(r(0))val(r(1))\dots$ comme la séquence bi-infinie de valuations dans r, et $\mathcal{V}(A) = \{val(r) \mid r \in \mathcal{R}(A)\}$.

Lemma 5.1 *Pour chaque BCA A, nous avons $r \in \mathcal{R}(A)$ ssi $r \circ \sigma \in \mathcal{R}(A)$.*

Un *chemin de contrôle* du CA (ou BCA) A est une séquence finie $q_0q_1\dots q_n$ d'états de contrôle telle que, pour tout $0 \le i < n$, il existe une transition $q_i \xrightarrow{\varphi_i} q_{i+1}$. Un *cycle* est un chemin de contrôle qui commence et se termine dans le même état de contrôle. Un *cycle élémentaire* est un cycle dans lequel chaque état apparaît exactement une fois, sauf le premier qui y apparaît deux fois. Un CA (ou BCA) est *plat* ssi chaque état de contrôle apparaît dans au plus un cycle élémentaire.

[1]Dans [98], une séquence bi-infinie est définie comme la classe d'équivalence de toutes les compositions $\beta \circ \sigma^n \circ \sigma^{-m}$ pour $n, m \in \mathbb{N}$. Cela est du au fait qu'une séquence bi-infinie reste la même si elle est décalée à gauche ou à droite. Pour simplifier, nous distinguons ici les séquences bi-infinies β, $\beta \circ \sigma^n$, et $\beta \circ \sigma^{-n}$ pour $n > 0$.

5.2.2.1 Décidabilité et propriétés de fermeture des FBCA

Nous considérons ici la classe des automates de Büchi bi-infinis à compteurs qui sont plats, dont les cycles élémentaires sont étiquetés par des DBC paramétriques et dont les autres transitions sont étiquetées par de formules PA. De plus, chaque contrainte de transition force les valeurs des paramètres à rester constantes. Nous appelons cette classe FBCA. Nous montrons que le problème du vide de FBCA est décidable en utilisant des résultats de [44, 35] et leurs extensions dans [67]. Essentiellement, on peut caractériser par une formule de Presburger l'effet d'un cycle élémentaire.

Lemma 5.2 *Le problème du vide est décidable pour la classe FBCA.*

La classe FBCA est aussi fermée par union et par intersection. Nous définissons d'abord ces opérations pour CA (BCA). Pour une valuation $\nu : \vec{x} \to \mathbb{Z}$, si $\vec{z} \subseteq \vec{x}$ et un sous-ensemble des compteurs de $\vec{x}$, soit $\nu \downarrow_{\vec{z}}$ la restriction de ν au domaine $\vec{z}$. Pour un sous-ensemble $\vec{z} \subset \vec{x}$ des compteurs de A et $s \in \mathcal{V}(A)$, nous définissons l'opérateur de restriction sur des séquences comme $s \downarrow_{\vec{z}} = \ldots val(s(-1)) \downarrow_{\vec{z}} val(s(0)) \downarrow_{\vec{z}} val(s(1)) \downarrow_{\vec{z}} \ldots$, et $\mathcal{V}(A) \downarrow_{\vec{z}} = \{s \downarrow_{\vec{z}} \mid s \in \mathcal{V}(A)\}$. Symétriquement, pour $\vec{z} \supset \vec{x}$, nous définissons l'opérateur d'extension sur des séquences $\mathcal{V}(A) \uparrow_{\vec{z}} = \{v \in {}^{\omega}(\vec{z} \mapsto \mathbb{Z})^{\omega} \mid v \downarrow_{\vec{x}} \in \mathcal{V}(A)\}$.

Nous appelons une classe d'automate à compteurs *fermée* par union et intersection, s'ils existent des opérations $\uplus$ et $\otimes$ telles que, pour deux FBCA $A_i = \langle \vec{x_i}, Q_i, L_i, R_i, \to_i \rangle$, $i = 1, 2$, nous avons $\mathcal{V}(A_1 \uplus A_2) = \mathcal{V}(A_1) \uparrow_{\vec{x_1} \cup \vec{x_2}} \cup \mathcal{V}(A_2) \uparrow_{\vec{x_1} \cup \vec{x_2}}$ et $\mathcal{V}(A_1 \otimes A_2) = \mathcal{V}(A_1) \uparrow_{\vec{x_1} \cup \vec{x_2}} \cap \mathcal{V}(A_2) \uparrow_{\vec{x_1} \cup \vec{x_2}}$, respectivement. La classe est appelé *effectivement* fermée par union et intersection si ces opérateurs sont effectivement calculable.

Proposition 5.3 *Soit $A = \langle \vec{x}, Q, L, R, \to \rangle$ un FBCA. Soit $A^c = \langle \vec{x}, Q, L^c, R^c, \to \rangle$ le FBCA tel que (1) pour tout $q \in L$ et $q' \in Q$, q' appartient au même cycle élémentaire que q ssi $q' \in L^c$, (2) pour tout $q \in R$ et $q' \in Q$, q' appartient au même cycle élémentaire que q ssi $q' \in R^c$. Alors, $\mathcal{R}(A) = \mathcal{R}(A^c)$.*

Supposons sans perte de généralité que $Q_1 \cap Q_2 \neq \emptyset$. L'union est définie comme $A_1 \uplus A_2 = \langle \vec{x_1} \cup \vec{x_2}, Q_1 \cup Q_2, L_1 \cup L_2, R_1 \cup R_2, \to_1 \cup \to_2 \rangle$. Le produit est défini comme $A_1 \otimes A_2 = \langle \vec{x_1} \cup \vec{x_2}, Q_1 \times Q_2, L_1^c \times L_2^c, R_1^c \times R_2^c, \to \rangle$ où $\to$ est donné par : $(q_1, q_1') \xrightarrow{\varphi_1 \wedge \varphi_2} (q_2, q_2')$ ssi $q_1 \xrightarrow{\varphi_1} q_2$ est une transition de A_1 et $q_1' \xrightarrow{\varphi_2} q_2'$ est une transition de A_2. Ici, L_i^c et R_i^c (pour $i = 1, 2$) dénotent les états acceptants gauches et droits de la proposition 5.3.

Lemma 5.4 *La classe FBCA est effectivement fermé par union et intersection.*

5.2.3 Définition de LIA

Dans cette section nous définissons la logique LIA.

5.2.3.1 Syntaxe

Nous considérons trois types de variables. Les *variables de bornes de tableau* (k, l) qui apparaissent dans les termes de bornes de tableau. Ces termes sont utilisés pour définir des intervals d'indices et des références statiques à l'intérieur des tableaux. Les *variables d'indice* (i, j) et les *variables de tableau* (a, b) sont utilisées pour construire les termes de tableaux. La figure 5.1 donne la syntaxe de LIA. Nous utilisons le symbole $\top$ pour la valeur booléenne *true*. Par la suite, nous utilisons $f \leq i \leq g$ à la place de $f \leq i \ \wedge \ i \leq g$, $i < f$ à la place de $i \leq f - 1$, et $i = f$ à la place de $f \leq i \leq f$. Intuitivement, notre logique est l'ensemble des combinaisons booléennes existentiellement quantifiées de :

1. Des formules de tableaux de la forme : $\forall \vec{i}\ .\ \varphi(\vec{k}, \vec{i}) \rightarrow \psi(\vec{k}, \vec{i}, \vec{a})$ où $\vec{k}$ est un ensemble de variables de bornes de tableau, $\vec{i}$ est un ensemble de variables d'indice, $\vec{a}$ est un ensemble de variables de tableau, φ est une formule arithmétique sur les variables d'indice et ψ est une formule arithmétique sur les termes de tableaux. En particulier, ψ est une contrainte DBC, et φ est composée de propositions atomiques de la forme $f \leq i$, $i \leq f$, $i - j \leq n$, ou $i \equiv_s t$ où f est une combinaison linéaire de variables de bornes de tableau, $n \in \mathbb{Z}$, et $0 \leq t < s$. Les variables $\vec{k}$ et $\vec{a}$ sont libres dans une formule de tableau, mais elles peuvent être existentiellement quantifiées au niveau le plus élevé.
2. Des formules PA sur les variables de bornes de tableau.

5.2.3.2 Exemples

Nous donnons ici quelques exemples de propriétés intéressantes sur des tableaux exprimées dans notre logique. Par exemple, un tableau a strictement ordonné jusqu'à une certaine borne est défini par

$$\exists k\ \forall i\ .\ 0 \leq i < k \rightarrow a[i] - a[i+1] \leq -1$$

$n, m, s, t \ldots$	$\in$	$\mathbb{Z}$	constantes ($0 \leq t < s$)
$k, l, \ldots$	$\in$	$BVar$	variables de bornes de tableau
$i, j, \ldots$	$\in$	$IVar$	variables d'indice
$a, b, \ldots$	$\in$	$AVar$	variables de tableau
B	$:=$	$n \mid k \mid B + B \mid B - B$	termes de bornes de tableau
I	$:=$	$i \mid I + n$	termes d'indice
A	$:=$	$a[I] \mid a[B]$	termes de tableau
G	$:=$	$B \leq I \mid I \leq B \mid I - I \leq n \mid$	
		$I \equiv_s t \mid G \vee G \mid G \wedge G$	gardes
V	$:=$	$A \leq B \mid B \leq A \mid$	
		$A - A \leq n \mid V \wedge V$	expressions de valeur
C	$:=$	$B \leq n \mid B \equiv_s t$	contraintes de bornes de tableau
P	$:=$	$\top \rightarrow V \mid G \rightarrow V \mid \forall i \,.\, P$	propriétés de tableau
U	$:=$	$P \mid C \mid \neg U \mid U \vee U \mid U \wedge U$	formules universelles
F	$:=$	$U \mid \exists k \,.\, F \mid \exists a \,.\, F$	formules LIA

FIG. 5.1 – Syntaxe de la logique LIA

Le fait que les premiers k éléments d'un tableau a sont inférieurs d'au moins 5 des premiers l éléments d'un tableau b est défini par

$$\exists k, l \; \forall i, j \;.\; 0 \leq i < k \;\wedge\; 0 \leq j < l \rightarrow a[i] - b[j] \leq -5$$

L'égalité de deux tableaux jusqu'à une certaine borne est exprimé par

$$\exists n \forall i \;.\; 0 \leq i < n \rightarrow a[i] = b[i]$$

L'utilisation des contraintes modulo permet d'exprimer des propriétés périodiques, par exemple

$$\forall i, j \;.\; i \equiv_2 0 \;\wedge\; j \equiv_2 1 \rightarrow a[i] \leq a[j]$$

qui dit que chaque valeur à une position paire est plus petite ou égale à toutes les valeurs des positions impaires. Ce type de formule est nécessaire pour montrer par exemple la validité d'un programme qui fusionne deux tableaux (voir [67]).

5.2.3.3 Sémantique

La logique LIA est interprétée sur des *tableaux bi-infinis*. Cela permet de traiter assez simplement des accès de tableaux hors bornes qui peuvent apparaître dans des programmes avec tableaux. On peut prévenir et/ou tester les accès hors bornes en introduisant explicitement des variables de bornes de tableau existentiellement quantifiées pour les variables de tableaux. Soit $\varphi(\vec{k}, \vec{a})$ une formule LIA. Une *valuation* est une paire de fonction partielle [2] $\langle \iota, \mu \rangle$ avec $\iota : BVar \cup IVar \rightarrow \mathbb{Z}_\perp$ qui associe un entier à chaque variable d'entier libre et $\mu : AVar \rightarrow {}^\omega\mathbb{Z}_\perp^\omega$ qui associe une séquence bi-infinie d'entiers à chaque tableau $a \in \vec{a}$. La valuation ι est étendue d'une façon standard vers des termes de bornes de tableau ($\iota(B)$) et des termes d'indices ($\iota(I)$). $\mathcal{I}_{\iota,\mu}(A)$ dénote la valeur d'un terme de tableau A donnée par une valuation $\langle \iota, \mu \rangle$. La sémantique d'une formule φ est définie en utilisant la relation $\models$ comme suit :

$$
\begin{array}{ccc}
\mathcal{I}_{\iota,\mu}(a[I]) & = & \mu(a)(\iota(I)) \\
\mathcal{I}_{\iota,\mu}(a[B]) & = & \mu(a)(\iota(B)) \\
\\
\langle \iota, \mu \rangle \models A \leq B & \iff & \mathcal{I}_{\iota,\mu}(A) \leq \iota(B) \\
\langle \iota, \mu \rangle \models A_1 - A_2 \leq n & \iff & \mathcal{I}_{\iota,\mu}(A_1) - \mathcal{I}_{\iota,\mu}(A_2) \leq n \\
\langle \iota, \mu \rangle \models \forall i \ . \ G \rightarrow V & \iff & \forall \ n \in \mathbb{Z} \ . \ \langle \iota[i \leftarrow n], \mu \rangle \models G \rightarrow V \\
\langle \iota, \mu \rangle \models \exists a \ . \ \psi & \iff & \exists \ \beta \in {}^\omega\mathbb{Z}^\omega \ . \ \langle \iota, \mu[a \leftarrow \beta] \rangle \models \psi
\end{array}
$$

Les règles manquantes sont standards. Un *modèle* de $\varphi(\vec{k}, \vec{a})$ est une valuation $\langle \iota, \mu \rangle$ telle que la formule obtenue en interprétant chaque variable $k \in \vec{k}$ par $\iota(k)$ et chaque tableau $a \in \vec{a}$ par $\mu(a)$ est valide : $\langle \iota, \mu \rangle \models \varphi$. Nous définissons $[\![\varphi]\!] = \{\langle \iota, \mu \rangle \mid \langle \iota, \mu \rangle \models \varphi\}$. Une formule est *satisfaisable* ssi $[\![\varphi]\!] \neq \emptyset$.

5.2.3.4 Un résultat d'indécidabilité

La raison pour laquelle les termes de tableaux ne peuvent pas apparaître dans des disjonctions d'expressions de valeur (voir la figure 5.1) est que sans cette restriction la logique devient indécidable. Essentiellement, cela peut être prouvé en montrant qu'une formule de tableaux $\forall \vec{i}.G \rightarrow V_1 \vee \ldots \vee V_n$, pour

[2] Le symbole $\perp$ est utilisé ici pour dénoter qu'une fonction partielle est indéfinie à un point donné.

$n > 1$, correspond grosso modo à n boucles imbriquées dans un automate à compteurs. L'indécidabilité est montrée par une réduction du problème d'arrêt d'une machine à deux compteurs [93].

Lemma 5.5 *La logique LIA étendue avec des disjonctions dans les expressions de valeur est indécidable.*

Remarquons qu'avoir plus qu'une boucle imbriquée est une condition nécessaire pour l'indécidabilité d'une machine à deux compteurs, car une machine à deux compteurs plate est dans la classe des automates à compteurs décidables de [44, 35].

5.2.4 Décidabilité du problème de satisfaisabilité

L'idée principale pour décider le problème de satisfaisabilité de LIA est que pour chaque formule φ de LIA il existe un FBCA A_φ tel que φ a un modèle, si et seulement si A_φ a un calcul acceptant. Plus précisément, chaque variable de tableau dans φ a un compteur correspondant dans A_φ, et étant donné un modèle de φ qui associe des entiers à chaque position des tableaux, A_φ a un calcul tel que les valeurs des compteurs aux différents positions du calcul correspondent aux valeurs des tableaux du modèle aux indices respectifs. Puisque le problème du vide est décidable pour les FBCA (lemme 5.2), cela entraîne la décidabilité de LIA.

Pour construire un automate à partir d'une formule LIA nous la normalisons d'abord vers une formule existentiellement quantifiée qui est une combinaison booléenne de propriétés simples de tableaux (voir la figure 5.1). Ensuite, chaque formule comme cela est traduit vers un FBCA. L'automate final A_φ est défini récursivement sur la structure de la formule normalisé en utilisant $\uplus$ et $\otimes$ correspondant aux connecteurs $\vee$ et $\wedge$, respectivement.

5.2.4.1 La normalisation des formules

Le but de cette étape est de transformer chaque formule de LIA dans une formule de la forme normale suivante :

$$\exists \vec{k} \exists \vec{a} \ . \ \bigvee_c \Big(\bigwedge_d \phi_{cd}(\vec{a}, \vec{k}) \Big) \wedge \theta_c(\vec{k}) \tag{NF}$$

où $\vec{a}$ est un ensemble de variables de tableaux, $\vec{k}$ est un ensemble de variables d'entiers, et

– θ_d est une conjonction de termes de la forme (i) $g(\vec{k}) \geq 0$ ou (ii) $g(\vec{k}) \equiv_s t$ avec g une combinaison linéaire de variables de $\vec{k}$ et $0 \leq t < s$,
– ϕ_{cd} est une formule des formes suivantes où $\sim \in \{\leq, \geq\}, m \in \mathbb{N}$, $0 \leq t < s$, $0 \leq v < u$, $q \in \mathbb{Z}$, et f_k, g_l, f_k^1, g_l^1, f_k^2, g_l^2 sont des combinaisons linéaires de variables de bornes de tableau :

$$\forall i \ . \ \bigwedge_{k=1}^{K} f_k \leq i \ \wedge \ \bigwedge_{l=1}^{L} i \leq g_l \ \wedge \ i \equiv_s t \rightarrow a[i] \sim h(\vec{k}) \qquad \text{(F1)}$$

Les formules (F1) lient chaque valeur de a dans un interval par une combinaison linéaire h de variables dans $\vec{k}$.

$$\forall i \ . \ \bigwedge_{k=1}^{K} f_k \leq i \ \wedge \ \bigwedge_{l=1}^{L} i \leq g_l \ \wedge \ i \equiv_s t \rightarrow a[i] - b[i+p] \sim q \qquad \text{(F2)}$$

Ici, $p \in \mathbb{Z}$. Les formules (F2) relient toutes les valeurs de a et b dans le même interval tel que la différence entre les indices de a et b est constante.

$$\forall i, j \ . \ \bigwedge_{k=1}^{K_1} f_k^1 \leq i \ \wedge \ \bigwedge_{l=1}^{L_1} i \leq g_l^1 \ \wedge \ \bigwedge_{k=1}^{K_2} f_k^2 \leq j \ \wedge \ \bigwedge_{l=1}^{L_2} j \leq g_l^2 \ \wedge \ i - j \leq p \ \wedge \ i \equiv_s t \ \wedge \ j \equiv_u v \rightarrow a[i] - b[j] \sim q \qquad \text{(F3)}$$

Ici, $p \in \mathbb{Z}^{\infty}$. Les formules (F3) relient toutes les valeurs de a avec les valeurs de b dans deux (possiblement les mêmes) intervals. Le cas où $p = \infty$ correspond à la situation où aucune contrainte $i - j \leq p$ avec $p \in \mathbb{Z}$ est utilisée.

Lemma 5.6 *Pour chaque formule de LIA il existe une formule équivalente en forme (NF).*

Par la suite nous appelons la matrice de φ la formule obtenu en enlevant les quantificateurs existentiels de la forme (NF) de φ.

5.2.4.2 Les formules et les graphes de contraintes

Dans [44, 35], l'ensemble des calculs d'un automate à compteurs plat est représenté par un graphe de contraintes de taille infinie. Ici, nous voyons les modèles d'une formule comme un graphe de contrainte infini (à gauche et à droite). Ces graphes de contraintes sont ensuite vus comme des exécutions de FBCA, reliant ainsi des modèles de formules aux calculs des automates.

Soit $\varphi(\vec{k}, \vec{a})$ une formule de type (F1)-(F3), et $\iota : \vec{k} \to \mathbb{Z}$ une valuation de ses variables de bornes de tableau $\vec{k}$. Pour la suite de la section nous fixons la valuation ι, et nous écrivons φ_ι pour la formule obtenue à partir de φ en remplaçant chaque occurrence de $k \in \vec{k}$ par la valeur $\iota(k)$.

La formule φ_ι peut être représentée par un graphe dirigé pondéré $G_{\iota,\varphi}$ dans lequel chaque sommet (a, n) représente la valeur $a[n]$ pour un $a \in \vec{a}$ et un $n \in \mathbb{Z}$, et il y a un chemin de poids w entre sommets (a, n) et (b, m) ssi la contrainte $a[n]-b[m] \leq w$ est impliqué par φ_ι. Dans la section suivante, nous montrons que ces graphes sont en correspondance avec les calculs acceptants d'un FBCA.

Pour construire le graphe de contrainte d'une formule, nous devons faire attention au problème suivant. Considérons, par exemple la formule

$$\forall i, j.i - j \leq 3 \wedge i \equiv_2 0 \wedge j \equiv_2 1 \to a[i] - b[j] \leq 5$$

Le graphe de contrainte de cette formule doit avoir un chemin de poids 5 entre, $a[0]$ et $b[1]$, $a[0]$ et $b[3]$, $a[0]$ et $b[5]$, etc. Remarquons que si on représente chaque chemin par un lien, la distance entre deux points liés est non-bornée. Puisque nous voulons que ce graphe représente un calcul d'un automate à compteurs plat, il est important de le définir comme une séquence composée d'une répétition (possiblement infinie) d'un nombre *fini* de sous-graphe (voir par exemple la figure 5.2(a) ou la figure 5.2(b)). Pour cela, nous introduisons des sommets intermédiaires qui sont connectés entre eux par des arcs 0 de sorte que pour chaque contrainte non-locale de la forme $a[n] - b[m] \leq w$ où $|n - m|$ peut être arbitrairement grand, il existe exactement un chemin de poids w à travers ces sommets. Par exemple, dans la figure 5.2(a), il y a un chemin $(a, 0) \xrightarrow{5} (t_\varphi, -3) \xrightarrow{0} \ldots \xrightarrow{0} (t_\varphi, 1) \xrightarrow{0} (b, 1)$ pour la contrainte $a[0] - b[1] \leq 5$, un autre chemin $(a, 0) \xrightarrow{5} (t_\varphi, -3) \xrightarrow{0} \ldots \xrightarrow{0} (t_\varphi, 3) \xrightarrow{0} (b, 3)$ pour la contrainte $a[0] - b[3] \leq 5$, etc.

Formellement, le graphe de contrainte $G_{\iota,\varphi} = \langle V, E\rangle$ d'une formule φ de type (F1)-(F3) est défini comme suit : L'ensemble des sommets est $V = (\mathcal{A} \cup \mathcal{T} \cup \{\zeta\}) \times \mathbb{Z}$. Ici, $\mathcal{A} = \{a\}$ pour des formules (F1), et $\mathcal{A} = \{a, b\}$ pour des formules (F2)-(F3), avec a ou a, b étant les tableaux qui apparaissent dans φ de type (F1) ou (F2)-(F3), respectivement. Ensuite, $\mathcal{T} = \emptyset$ pour les formules (F1)-(F2), et $\mathcal{T} = \{t_\varphi\}$ pour les formules (F3) où t_φ est un symbole auxiliaire unique associé à chaque formule φ de type (F3). Finalement, ζ est un symbole spécial (trace zéro). L'ensemble des arêtes E est défini basé sur

le type (F1)-(F3) de φ. Ici, nous donnons uniquement les définitions pour les formules de type (F3). Nous avons :

$$E \supset \{(\zeta, k) \xrightarrow{0} (\zeta, k+1) \mid k \in \mathbb{Z}\} \ \cup \ \{(\zeta, k+1) \xrightarrow{0} (\zeta, k) \mid k \in \mathbb{Z}\}$$

c'est-à-dire, la valeur de la trace zéro reste constante.

Graphe de contrainte pour les formules (F3) Soit φ la formule ci-dessous où $0 \leq s < t$, $0 \leq u < v$, $p \in \mathbb{Z}^\infty$, $q \in \mathbb{Z}$, et f_k^1, g_l^1, f_k^2, g_l^2 sont des combinaisons linéaires de variables de bornes de tableau :

$$\forall i, j \,.\, \underbrace{\bigwedge_{k=1}^{K_1} f_k^1 \leq i \ \wedge \ \bigwedge_{l=1}^{L_1} i \leq g_l^1 \ \wedge \ i \equiv_s t}_{\phi^1} \wedge \underbrace{\bigwedge_{k=1}^{K_2} f_k^2 \leq j \ \wedge \ \bigwedge_{l=1}^{L_2} j \leq g_l^2 \ \wedge \ j \equiv_u v}_{\phi^2}$$

$$\wedge \ i - j \leq p \rightarrow a[i] - b[j] \sim q$$

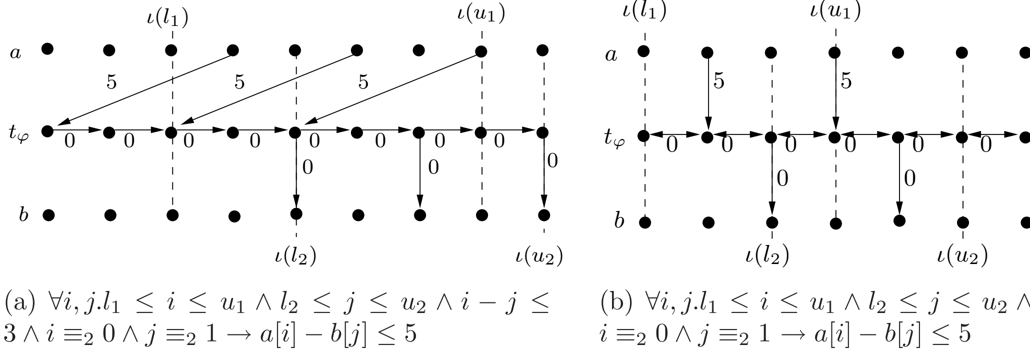
Soient $\phi^1(i, \vec{k})$ et $\phi^2(j, \vec{k})$ les sous-formules définissant respectivement les domaines de i et j, et $\mathcal{P}_\iota^1 = \{n \in \mathbb{Z} \mid \ \models \phi_\iota^1[n/i]\}$ et $\mathcal{P}_\iota^2 = \{n \in \mathbb{Z} \mid \ \models \phi_\iota^2[n/j]\}$ ces domaines sous la valuation ι. Soit $T_\leq = \{(t_\varphi, k) \xrightarrow{0} (t_\varphi, k+1) \mid k \in \mathbb{Z} \wedge \exists n \in \mathcal{P}_\iota^1 \ \exists m \in \mathcal{P}_\iota^2 \ . \ n - m \leq p\}$ et $T_\geq = \{(t_\varphi, k) \xrightarrow{0} (t_\varphi, k-1) \mid k \in \mathbb{Z} \wedge \exists n \in \mathcal{P}_\iota^1 \ \exists m \in \mathcal{P}_\iota^2 \ . \ n - m \geq p\}$. Remarquons que $T_\leq$ et $T_\geq$ sont vides si la précondition de φ n'est pas satisfaisable. L'ensemble des arêtes E est défini par :

1. Si $p < \infty$, alors il y a deux cas basés sur la direction de la contrainte $a[i] - b[j] \sim q$:
   (a) pour $a[i] - b[j] \leq q$, nous avons (Figure 5.2(a)) :
   $E \supset \{(a, k) \xrightarrow{q} (t_\varphi, k - p) \mid k \in \mathcal{P}_\iota^1\} \ \cup \ \{(t_\varphi, k) \xrightarrow{0} (b, k) \mid k \in \mathcal{P}_\iota^2\} \cup \ T_\leq$
   (b) Pour $a[i] - b[j] \geq q$, nous avons :
   $E \supset \{(b, k) \xrightarrow{-q} (t_\varphi, k + p) \mid k \in \mathcal{P}_\iota^2\} \ \cup \ \{(t_\varphi, k) \xrightarrow{0} (a, k) \mid k \in \mathcal{P}_\iota^1\} \cup \ T_\geq$
2. Si $p = \infty$, nous considérons encore une fois deux cas basés sur la direction de la contrainte $a[i] - b[j] \sim q$:

(a) Pour $a[i] - b[j] \leq q$, nous avons (Figure 5.2(b)) :
$E \supset \{(a,k) \xrightarrow{q} (t_\varphi, k) \mid k \in \mathcal{P}^1_\iota\} \cup \{(t_\varphi, k) \xrightarrow{0} (b,k) \mid k \in \mathcal{P}^2_\iota\} \cup T_\leq \cup T_\geq$

(b) Pour $a[i] - b[j] \geq q$, nous avons :
$E \supset \{(b,k) \xrightarrow{-q} (t_\varphi, k) \mid k \in \mathcal{P}^2_\iota\} \cup \{(t_\varphi, k) \xrightarrow{0} (a,k) \mid k \in \mathcal{P}^1_\iota\} \cup T_\leq \cup T_\geq$

Aucune autre arête est dans E.



(a) $\forall i, j. l_1 \leq i \leq u_1 \wedge l_2 \leq j \leq u_2 \wedge i - j \leq 3 \wedge i \equiv_2 0 \wedge j \equiv_2 1 \rightarrow a[i] - b[j] \leq 5$

(b) $\forall i, j. l_1 \leq i \leq u_1 \wedge l_2 \leq j \leq u_2 \wedge i \equiv_2 0 \wedge j \equiv_2 1 \rightarrow a[i] - b[j] \leq 5$

Fig. 5.2 – Deux exemples de graphes des contraintes pour des formules (F3)

Relier les graphes des contraintes et les modèles de formules Nous pouvons prouver une correspondance entre les graphes de contraintes et les modèles de formules de la forme (F1)-(F3). Plus précisément, nous pouvons montrer que si les arêtes d'un graphe de contrainte pour une formule φ peuvent être étiquetées d'une façon consistante, alors à partir des étiquettes on peut extraire un modèle de φ, et vice versa. Cela formalise la correction de la construction des graphes de contraintes avec des traces supplémentaires.

Soit $\varphi(\vec{k}, \vec{a})$ une formule de la forme (F1)-(F3), $\iota : \vec{k} \rightarrow \mathbb{Z}$ une valuation des variables de bornes de tableau de φ, et $G_{\iota,\varphi} = (V, E)$ le graphe de contrainte correspondant. Un *étiquetage* $Lab : V \rightarrow \mathbb{Z}$ de $G_{\iota,\varphi}$ est appelé *consistant* ssi (1) pour toute arête $v_1 \xrightarrow{k} v_2 \in E$, on a $Lab(v_1) - Lab(v_2) \leq k$ et (2) $Lab((\zeta, n)) = 0$ pour tout $n \in \mathbb{Z}$.

Lemma 5.7 *Soit $\varphi(\vec{k}, \vec{a})$ une formule de la forme (F1)-(F3). Alors, pour toute valuation $\iota : \vec{k} \rightarrow \mathbb{Z}$ et $\mu : \vec{a} \rightarrow {}^\omega\mathbb{Z}^\omega$, on a $\langle \iota, \mu \rangle \models \varphi$ ss'il existe*

un étiquetage consistant Lab de $G_{\iota,\varphi}$ *tel que* $\mu(a)(i) = Lab((a, i))$ *pour tout* $a \in \vec{a}$ *et* $i \in \mathbb{Z}$.

5.2.4.3 D'une formule vers un automate à compteurs

Dans cette section, nous donnons la construction d'un FBCA A_φ qui correspond à une formule φ de sorte que (1) chaque calcul de A_φ correspond à un modèle de φ, et (2) pour chaque modèle de φ, A_φ a au moins un calcul correspondant. De cette façon, nous reduisons le problème de satisfaisabilité de LIA au problème du vide pour FBCA.

La construction du FBCA est par induction sur la structure des formules. Pour la suite de la section, soit φ une formule, $\vec{k}$ l'ensemble des variables de bornes de tableaux de φ, et $\vec{a}$ l'ensemble des variables de tableaux de φ, c.-à-d. $FV(\varphi) = \vec{k} \cup \vec{a}$. Supposons que φ est la matrice d'une formule en forme normale (NF), c.-à-d. $\varphi : \bigvee_{i \in I} \theta_i(\vec{k}) \wedge \bigwedge_{j \in J} \psi_{ij}(\vec{k}, \vec{a})$ où θ_i sont des contraintes PA et ψ_{ij} des formules de type (F1)-(F3). L'automate A_φ est défini comme $\biguplus_{i \in I} A_{\theta_i} \otimes \bigotimes_{j \in J} A_{\psi_{ij}}$ où $\uplus$ et $\otimes$ sont les opérations d'union et intersection de FBCA. La construction de l'automate à compteurs $A_{\psi_{ij}}$ pour les formules ψ_{ij} de type (F1)-(F3) est basée sur la définition des graphes de contraintes dans la section 5.2.4.2. Notamment, chaque calcul acceptant de $A_{\psi_{ij}}$ donne une valuation consistante du graphe de contraintes de ψ_{ij}.

Les gabarits d'automates à compteurs Pour simplifier la définition des automates à compteurs, nous remarquons que chaque graphe de contrainte pour les formules basiques de type (F1)-(F3) est composé d'arêtes *horizontales*, *verticales*, et *diagonales*, qui sont définies essentiellement de la même façon pour tous les types de formules. Pour cette raison, nous définissons trois types de *gabarits* d'automates à compteurs qui sont par la suite utilisés pour définir les automates pour les formules basiques.[3] Plus précisément, les automates pour les formules (F1)-(F3) sont définis comme des $\otimes$-produits d'instances des gabarits d'automates pour les arêtes horizentales, verticales et diagonales des graphes de contraintes respectifs. Dans la suite nous supposons l'existence d'un compteur spécial x_τ (tic-tac) incrémenté par chaque transition, c.-à-d. nous supposons que la contrainte $x'_\tau = x_\tau + 1$ est implicitement ajoutée en conjonction avec chaque formule étiquetant une transition. Intuitivement, le rôle du compteur x_τ est de synchroniser tous les automates

[3] Par un gabarit, nous entendons une classe d'automates à compteurs qui utilisent la même structure.

composés par les $\otimes$-produits sur une position commune.

Le gabarit pour les arêtes horizontales Soit a un symbole de tableau, $dir \in \{\texttt{left}, \texttt{right}, \texttt{bi}\}$ un paramètre de *direction*, et ϕ une formule sur les variables de bornes de tableaux. Soit $\vec{x_k}$ l'ensemble $\{x_k \mid k \in FV(\phi)\}$. Nous définissions le gabarit $H(a, dir, \phi) = \langle \vec{x}, Q, L, R, \rightarrow \rangle$ où :

- $\vec{x} = \{x_a\} \cup \vec{x_k}$. Ces compteurs auront les mêmes noms dans toutes les instances de H.
- $Q = \{q_L, q_R, p_L, p_R\}$. Les états de contrôle doivent avoir des noms "fraîchement" choisis pour chaque instance de H. $L = \{q_L, p_L\}$ et $R = \{q_R, p_R\}$.
- $q_L \xrightarrow{\xi} q_L$, $q_R \xrightarrow{\xi} q_R$, $q_L \xrightarrow{\phi(\vec{x_k}) \ \wedge \ \xi} q_R$, $p_L \xrightarrow{\top} p_L$, $p_R \xrightarrow{\top} p_R$, et $p_L \xrightarrow{\neg\phi(\vec{x_k})} p_R$.

Ci-dessus, $\phi(\vec{x_k})$ est la formule obtenue en remplaçant chaque occurrence d'un variable de borne de tableaux $k \in FV(\phi)$ par le compteur x_k correspondant. La formule $\xi(x_a, x'_a)$ est $x_a - x'_a \leq 0$ si $dir = \texttt{right}$, $x'_a - x_a \leq 0$ si $dir = \texttt{left}$, et $x'_a = x_a$ si $dir = \texttt{bi}$. Nous supposons aussi, que pour chaque transition, $\bigwedge_{k \in FV(\phi)} x'_k = x_k$ est ajouté implicitement comme conjonction à chaque étiquette (formule), c.-à-d. la valeur d'un compteur x_k reste constant dans un calcul.

Si la formule ϕ est vraie pour une valuation donnée des paramètres $\vec{x_k}$, alors chaque calcul acceptant d'une instance de H visite q_L infiniment souvent à gauche et visite q_R infiniment souvent à droite. Sinon, si ϕ n'est pas vraie pour une valuation donnée de $\vec{x_k}$, l'automate (l'instance) a un calcul qui visite infiniment souvent p_L à gauche et p_R à droite. Dans ce cas, l'automate n'impose pas de contraintes sur x_a.

Le gabarit pour les arêtes diagonales Soient a, b deux symboles de tableaux, $q \in \mathbb{Z}$, $p, s \in \mathbb{N}^+$, $t \in [0, s-1]$, et $dir \in \{\texttt{left}, \texttt{right}\}$ un paramètre de direction. Par la suite nous appelons $\mathbb{L} = \{l_1, \ldots, l_K\}$ et $\mathbb{U} = \{u_1, \ldots, u_L\}$ respectivement les bornes inférieures et supérieures, où l_i et u_j sont des combinaisons linéaires de variables de bornes de tableaux. Soit $\vec{x_k} = \{x_k \mid k \in \bigcup_{i=1}^{K} FV(l_i) \ \cup \ \bigcup_{j=1}^{L} FV(u_j)\}$. Nous supposons aussi, que $\mathbb{L} \cup \mathbb{U} \neq \emptyset$ et nous traitons le cas $\mathbb{L} \cup \mathbb{U} = \emptyset$ plus tard. Nous définissons le gabarit $D(a, b, p, q, s, t, \mathbb{L}, \mathbb{U}, dir) = \langle \vec{x}, Q, L, R, \rightarrow \rangle$ où :

- $\vec{x} = \{x_a, x_b\} \cup \vec{x_k} \cup \{x_i \mid 1 \leq i < p\}$. Les compteurs x_a, x_b, et $\vec{x_k}$ auront les mêmes noms dans toutes les instances de D. De l'autre coté, les compteurs x_i, $1 \leq i < p$, ont des noms fraîchement choisis

pour chaque instance de D. Les compteurs x_i sont utilisés pour scinder les arêtes diagonales qui enjambent plus qu'une position en plusieurs arêtes connectant uniquement des positions adjacentes. Par exemple, la contrainte $a[i] - b[i+3] \leq 5$ peut être scindée en $a[i] - x_1[i+1] \leq 5$, $x_1[i+1] - x_2[i+2] \leq 0$, et $x_2[i+2] - b[i+3] \leq 0$. Les contraintes sur les valeurs des tableaux à deux indices voisins peuvent ensuite être exprimées en utilisant les valeurs actuelles et futures des compteurs correspondants (par exemple, ici, $x_a - x_1' \leq 5$, $x_1 - x_2' \leq 0$, et $x_2 - x_b' \leq 0$, qui apparaissent sur des transitions qui se suivent dans le FBCA).

- $Q = \{q_L, q_R\} \cup \{q_i \mid 0 \leq i < s\} \cup \{q_i^j \mid 0 \leq j < s,\ j+1 \leq i < j+p\}$. Les états de contrôle doivent avoir des noms "fraîchement" choisis pour chaque D. Soient $L = \{q_L\} \cup \{q_i \mid 0 \leq i < s\}$ et $R = \{q_R\} \cup \{q_i \mid 0 \leq i < s\}$.
- $q_L \xrightarrow{\top} q_L$, $q_R \xrightarrow{\top} q_R$, et $q_L \xrightarrow{\neg(\exists i\ .\ \bigwedge_{l \in \mathbb{L}} i \geq l(\vec{x_k})\ \wedge\ \bigwedge_{u \in \mathbb{U}} i \leq u(\vec{x_k})\ \wedge\ i \equiv_s t)} q_R$.
- $q_L \xrightarrow{\bigwedge_{l \in \mathbb{L}} x_\tau \geq l(\vec{x_k}) - 1\ \wedge\ (\bigvee_{l \in \mathbb{L}} x_\tau = l(\vec{x_k}) - 1)\ \wedge\ x_\tau + 1 \equiv_s i} q_i$ pour i avec $0 \leq i < s$.
- $q_i \xrightarrow{\bigwedge_{l \in \mathbb{L}} x_\tau \geq l(\vec{x_k})\ \wedge\ \bigwedge_{u \in \mathbb{U}} x_\tau < u(\vec{x_k})\ \wedge\ \xi_i[x_a/x_0, x_b/x_p]} q_{(i+1) \bmod s}$ pour tout i avec $0 \leq i < s$.
- $q_i \xrightarrow{\bigvee_{u \in \mathbb{U}} x_\tau = u(\vec{x_k})\ \wedge\ x_\tau \equiv_s i\ \wedge\ \xi_i[x_a/x_0, x_b/x_p]} q_{i+1}^i$ pour tout i avec $0 \leq i < s$.
- $q_i \xrightarrow{\bigvee_{u \in \mathbb{U}} x_\tau = u(\vec{x_k})\ \wedge\ x_\tau \equiv_s i\ \wedge\ \xi_i[x_a/x_0, x_b/x_p]} q_R$ pour tout i avec $0 \leq i < s$ si $p = 1$.
- $q_i^j \xrightarrow{\xi_i[x_a/x_0, x_b/x_p]} q_{i+1}^j$ pour tout j avec $0 \leq j < s$, $j < i < j+p-1$.
- $q_{j+p-1}^j \xrightarrow{\xi_i[x_a/x_0, x_b/x_p]} q_R$ pour tout j avec $0 \leq j < s$ si $p > 1$.

Ci-dessus, $l(\vec{x_k})$ et $u(\vec{x_k})$ dénotent les expressions l et u dans lesquelles chaque occurrence d'une variable de borne de tableaux k est remplacée par le compteur (paramètre) x_k correspondant. Comme avant, pour chaque transition nous supposons que $\bigwedge_{k \in FV(\phi)} x_k' = x_k$ est ajoutée implicitement comme conjonction à chaque étiquette (formule), c.-à-d. la valeur d'un compteur x_k reste constant dans un calcul. Les formules ξ_i sont définies comme suit :

- si $dir = \texttt{right}$, $\xi_i = \bigwedge_{k \in K_i} x_k - x_{k+1}' \leq \alpha_k$ pour $K_i = \{k \mid 0 \leq k < p,\ i \equiv_s k + t\}$, $\alpha_0 = q$, et $\alpha_k = 0$, $k > 0$,
- si $dir = \texttt{left}$, $\xi_i = \bigwedge_{k \in K_i} x_{k-1}' - x_k \leq \alpha_k$, $K_i = \{k \mid 1 \leq k \leq p,\ k + i \equiv_s t\}$, $\alpha_1 = q$, et $\alpha_k = 0$, $k > 1$.

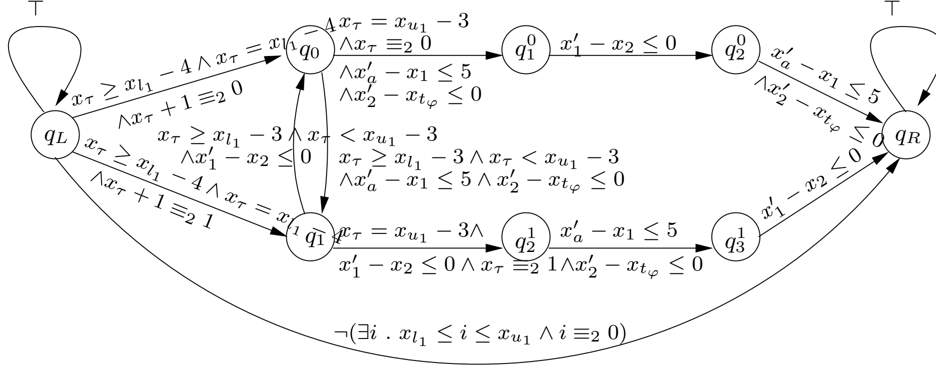


FIG. 5.3 – Le FBCA pour les arêtes diagonales de la formula $\varphi : \forall i, j. l_1 \leq i \leq u_1 \wedge l_2 \leq j \leq u_2 \wedge i - j \leq 3 \wedge i \equiv_2 0 \wedge j \equiv_2 1 \rightarrow a[i] - b[j] \leq 5$ de la figure 5.2(a) obtenue comme $D(a, t_\varphi, 3, 5, 2, 0-3, \{l_1 - 3\}, \{u_1 - 3\}, \texttt{left})$. Pour comprendre la formule ξ_0 sur la transition de q_0 vers q_1, notons que la contrainte $i \equiv_s k + t$ dans la définition de l'ensemble K_0 est instantiée à $0 \equiv_2 k - 3$, et donc $K_0 = \{1, 3\}$. Un raisonnement similaire s'applique aux autres transitions.

Finalement, pour le cas $\mathbb{L} = \mathbb{U} = \emptyset$, nous définissons chaque instance de $D(a, b, p, q, s, t, \emptyset, \emptyset, dir)$ d'être $A_1 \otimes A_2$ où A_1 est une instance de $D(a, b, p, q, s, t, \emptyset, \{0\}, dir)$ et A_2 une instance de $D(a, b, p, q, s, t, \{0\}, \emptyset, dir)$.

La construction peut être expliquée en considérant un calcul acceptant d'une instance de D. Considérons le cas où il existe une valeur i entre les bornes qui satisfait aussi la contrainte modulo. Si cela n'est pas le cas, il y a un calcul acceptant qui prend la transitions $q_L \xrightarrow{\neg(\exists i \;.\; \bigwedge_{l \in \mathbb{L}} i \geq l(\vec{x_k}) \;\wedge\; \bigwedge_{u \in \mathbb{U}} i \leq u(\vec{x_k}) \;\wedge\; i \equiv_s t)} q_R$ exactement une fois.

Puisque le calcul est acceptant, il doit visiter infiniment souvent un état de L à gauche et un état de R à droite. Il y a trois cas : (1) $\mathbb{L} \neq \emptyset$ et $\mathbb{U} \neq \emptyset$, (2) $\mathbb{L} = \emptyset$ et $\mathbb{U} \neq \emptyset$, et (3) $\mathbb{L} \neq \emptyset$ et $\mathbb{U} = \emptyset$. Dans le cas (1), un calcul bi-infini visite q_L infiniment souvent à gauche et q_R infiniment souvent à droite. Notons que le calcul ne peut pas visiter la boucle $q_0 \rightarrow \ldots \rightarrow q_{s-1}$ infiniment souvent grâce aux bornes inférieures et supérieures sur x_τ. Dans le cas (2), le calcul ne peut prendre aucune des transitions $q_L \rightarrow q_i$, $0 \leq i < s$, à cause du fait que $\mathbb{L}$ est vide, ce qui rend la garde insatisfaisable. La seule possibilité pour un calcul bi-infini acceptant est donc de visiter les états $q_0 \rightarrow \ldots \rightarrow q_{s-1}$ infiniment souvent à gauche. Grâce à la borne supérieure sur x_τ, le calcul ne

peut pas rester infiniment souvent dans cette boucle et doit sortir par une des transitions $q_i \rightarrow q^i_{i+1}$ (ou $q_i \rightarrow q_R$ pour $p = 1$) en étant ensuite enfermé dans q_R à droite. Le cas (3) est symétrique au cas (2).

Notons que dans tous les cas, grâce aux tests modulo sur x_τ dans l'entrée et la sortie de la boucle principale $q_0 \rightarrow \ldots \rightarrow q_{s-1}$ la valeur du compteur x_τ doit être égale à i modulo s quand un état q_i, $0 \leq i < s$ est visité dans chaque calcul acceptant. Les états q^j_i sont utilisés pour décrire les contraintes qui correspondent à des arêtes qui commencent à l'intérieur des bornes et vont au delà de la borne supérieure (et vice-versa, c.-à-d. les arêtes inversées) Le nombre de ces arêtes est borné. Cette construction n'est pas nécessaire pour les bornes inférieures car les gabarits sont utilisés uniquement de sorte qu'aucune arête va au-dessous d'une borne inférieure.

Le gabarit pour les arêtes verticales Soient a, b deux symboles de tableaux, q une combinaison de variables de bornes de tableaux, $p, s \in \mathbb{N}^+$, et $t \in [0, s-1]$. Par la suite nous appelons $\mathbb{L} = \{l_1, \ldots, l_K\}$ et $\mathbb{U} = \{u_1, \ldots, u_L\}$ respectivement les bornes inférieurs et supérieurs, où l_i et u_j sont des combinaisons linéaires de variables de bornes de tableaux. Soit aussi $\vec{x_k} = \{x_k \mid k \in \bigcup_{i=1}^K FV(l_i) \;\cup\; \bigcup_{j=1}^L FV(u_j)\}$. Par ailleurs, nous supposons $\mathbb{L} \cup \mathbb{U} \neq \emptyset$ et nous traitons le cas $\mathbb{L} \cup \mathbb{U} = \emptyset$ plus tard. Nous définissons le gabarit $V(a, b, p, q, s, t, \mathbb{L}, \mathbb{U}) = \langle \vec{x}, Q, L, R, \rightarrow \rangle$ où :

- $\vec{x} = \{x_a, x_b\} \cup \vec{x_k}$. Ces compteurs auront les mêmes noms dans toutes les instances de V.
- $Q = \{q_L, q_R\} \cup \{q_i \mid 0 \leq i < s\}$. Les états de contrôle doivent avoir des noms "fraîchement" choisis pour chaque instance de V. $L = \{q_L\} \cup \{q_i \mid 0 \leq i < s\}$ et $R = \{q_R\} \cup \{q_i \mid 0 \leq i \leq s\}$.
- $q_L \xrightarrow{\top} q_L$, $q_R \xrightarrow{\top} q_R$, et $q_L \xrightarrow{\neg(\exists i \;.\; \bigwedge_{l \in \mathbb{L}} i \geq l(\vec{x_k}) \;\wedge\; \bigwedge_{u \in \mathbb{U}} i \leq u(\vec{x_k}) \;\wedge\; i \equiv_s t)} q_R$.
- $q_L \xrightarrow{\bigwedge_{l \in \mathbb{L}} x_\tau \geq l(\vec{x_k}) - 1 \;\wedge\; \bigvee_{l \in \mathbb{L}} x_\tau + 1 = l(\vec{x_k}) \;\wedge\; x_\tau + 1 \equiv_s i} q_i$, $0 \leq i < s$.
- $q_i \xrightarrow{\bigwedge_{l \in \mathbb{L}} x_\tau \geq l(\vec{x_k}) \;\wedge\; \bigwedge_{u \in \mathbb{U}} x_\tau < u(\vec{x_k}) \;\wedge\; x_a - x_b \leq q(\vec{x_k})} q_{(i+1) \bmod s}$, $0 \leq i < s$ et $i \equiv_s t$.
- $q_i \xrightarrow{\bigwedge_{l \in \mathbb{L}} x_\tau \geq l(\vec{x_k}) \;\wedge\; \bigwedge_{u \in \mathbb{U}} x_\tau < u(\vec{x_k})} q_{(i+1) \bmod s}$, $0 \leq i < s$ et $i \not\equiv_s t$.
- $q_i \xrightarrow{\bigvee_{u \in \mathbb{U}} x_\tau = u(\vec{x_k}) \;\wedge\; x_\tau \equiv_s i \;\wedge\; x_a - x_b \leq q(\vec{x_k})} q_R$, $0 \leq i < s$ et $i \equiv_s t$.
- $q_i \xrightarrow{\bigvee_{u \in \mathbb{U}} x_\tau = u(\vec{x_k}) \;\wedge\; x_\tau \equiv_s i} q_R$, $0 \leq i < s$ et $i \not\equiv_s t$.

Ci-dessus, $l(\vec{x_k})$, $u(\vec{x_k})$, et $q(\vec{x_k})$ dénotent les expressions l, u, et q où chaque occurrence d'une variable de bornes de tableaux k est remplacée par le paramètre x_k. Comme avant, nous supposons, que $\bigwedge_{k \in FV(\phi)} x'_k = x_k$ est ajouté implicitement comme conjonction à chaque étiquette (formule), c.-à-d. la valeur d'un compteur x_k reste constant dans un calcul. Finalement, si $\mathbb{L} = \mathbb{U} = \emptyset$, nous définissons chaque instance de $V(a, b, p, q, s, t, \emptyset, \emptyset)$ comme $A_1 \otimes A_2$ où A_1 est une instance de $V(a, b, p, q, s, t, \emptyset, \{0\})$ et A_2 est une instance de $V(a, b, p, q, s, t, \{0\}, \emptyset)$. L'intuition derrière la construction de V est très similaire que celle pour D.

5.2.4.4 Automates à compteurs pour les formules basiques

Nous donnons ici la construction d'un FBCA pour les formules basiques. Cela est fait en composant des instances des gabarits en utilisant l'opérateur $\otimes$ pour l'intersection (voir la section 5.2.2.1). Nous donnons ici la construction du FBCA pour des formules (F3). Les formules de type (F1), (F2), et les contraintes PA sur des variables de bornes de tableaux sont données dans [67]. Soit φ une formule (F3)

$$\forall i, j \,.\, \underbrace{\bigwedge_{k=1}^{K_1} f_k^1 \leq i \;\wedge\; \bigwedge_{l=1}^{L_1} i \leq g_l^1 \;\wedge\; \bigwedge_{k=1}^{K_2} f_k^2 \leq j \;\wedge\; \bigwedge_{l=1}^{L_2} j \leq g_l^2 \;\wedge\; i - j \leq p \;\wedge\; i \equiv_s t \;\wedge\; j \equiv_u v}_{\phi} \rightarrow a[i] - b[j] \sim q$$

où $0 \leq s < t$ et $0 \leq u < v$. Soit $\mathbb{L}_i = \{f_1^i, \ldots, f_{K_i}^i\}$ et $\mathbb{U}_i = \{g_1^i, \ldots, g_{L_i}^i\}$ pour $i = 1, 2$. Nous écrivons ϕ pour la précondition de φ. L'automate A_φ est défini comme $A_\varphi = A_1 \otimes A_2 \otimes A_3$ où A_1, A_2, A_3 sont instantiés par rapport au tableau 5.1.

5.2.4.5 Assembler les automates pour des formules normalisées

Étant donnée une formule $\varphi(\vec{k}, \vec{a})$ qui est une combinaison booléenne positive de formules de type (F1)-(F3) et des contraintes PA sur des variables de bornes de tableaux $\vec{k}$, soit A_φ l'automate défini inductivement sur la structure de φ comme suit :

- Si φ est de type (F1)-(F3), ou une contrainte PA sur $\vec{k}$, alors A_φ est tel que défini dans la section 5.2.4.4,
- si $\varphi = \psi_1 \wedge \psi_2$, alors $A_\varphi = A_{\psi_1} \otimes A_{\psi_2}$,
- si $\varphi = \psi_1 \vee \psi_2$, alors $A_\varphi = A_{\psi_1} \uplus A_{\psi_2}$.

p	$\sim$	A_1	A_2	A_3
∞	$\leq$	$V(a, t_\varphi, q, s, t, \mathbb{L}_1, \mathbb{U}_1)$	$H(t_\varphi, \texttt{bi}, \exists i, j.\phi)$	$V(t_\varphi, b, 0, u, v, \mathbb{L}_2, \mathbb{U}_2)$
∞	$\geq$	$V(b, t_\varphi, -q, u, v, \mathbb{L}_2, \mathbb{U}_2)$	$H(t_\varphi, \texttt{bi}, \exists i, j.\phi)$	$V(t_\varphi, a, 0, s, t, \mathbb{L}_1, \mathbb{U}_1)$
0	$\leq$	$V(a, t_\varphi, q, s, t, \mathbb{L}_1, \mathbb{U}_1)$	$H(t_\varphi, \texttt{right}, \exists i, j.\phi)$	$V(t_\varphi, b, 0, u, v, \mathbb{L}_2, \mathbb{U}_2)$
0	$\geq$	$V(b, t_\varphi, -q, u, v, \mathbb{L}_2, \mathbb{U}_2)$	$H(t_\varphi, \texttt{left}, \exists i, j.\phi)$	$V(t_\varphi, a, 0, s, t, \mathbb{L}_1, \mathbb{U}_1)$
> 0	$\leq$	$D(a, t_\varphi, p, q, s, t-p, \mathbb{L}_1-p, \mathbb{U}_1-p, \texttt{left})$	$H(t_\varphi, \texttt{right}, \exists i, j.\phi)$	$V(t_\varphi, b, 0, u, v, \mathbb{L}_2, \mathbb{U}_2)$
> 0	$\geq$	$D(b, t_\varphi, p, -q, u, v, \mathbb{L}_2, \mathbb{U}_2, \texttt{right})$	$H(t_\varphi, \texttt{left}, \exists i, j.\phi)$	$V(t_\varphi, a, 0, s, t, \mathbb{L}_1, \mathbb{U}_1)$
< 0	$\leq$	$D(a, t_\varphi, -p, q, s, t, \mathbb{L}_1, \mathbb{U}_1, \texttt{right})$	$H(t_\varphi, \texttt{right}, \exists i, j.\phi)$	$V(t_\varphi, b, 0, u, v, \mathbb{L}_2, \mathbb{U}_2)$
< 0	$\geq$	$D(b, t_\varphi, -p, -q, u, v+p, \mathbb{L}_2+p, \mathbb{U}_2+p, \texttt{left})$	$H(t_\varphi, \texttt{left}, \exists i, j.\phi)$	$V(t_\varphi, a, 0, s, t, \mathbb{L}_1, \mathbb{U}_1)$

TAB. 5.1 – La table d'instantiation de formules (F3). Dans quelques lignes nous transposons les bornes originales apparaissant dans les formules pour pouvoir utiliser les gabarits qui ne traitent pas explicitement les arêtes qui partent à l'intérieur des bornes et qui arrivent au-dessous. La sémantique des formules est préservée grâce à la façon dont les gabarits sont construits, au lieu d'arêtes qui vont au-dessous d'une borne inférieure pour un interval nous obtenons les mêmes arêtes qui vont juste au-dessus de la borne inférieur de l'interval transposé. Étant donné un ensemble d'entiers S et un entier p, nous utilisons la notation $S + p$ pour $\{s + p \mid s \in S\}$.

Soit $r \in \mathcal{R}(A_\varphi)$ un calcul acceptant de A_φ et $\delta(r) = val(r(0))(x_\tau)$ la valeur du compteur x_τ (tic-tac) à la position 0 de r. Nous appelons $\eta(r) = r \circ \sigma^{-\delta(r)}$ le *calcul centré* obtenu de r en le transposant tel que la valeur de x_τ à la position 0 et aussi 0. En utilisant le lemme 5.1, r est aussi un calcul acceptant de A_φ ssi $\eta(r)$ l'est. Notons que r induit les valuations suivantes sur $\vec{k}$ et $\vec{a}$, respectivement : $\iota_r(k) = val(\eta(r)(0))(x_k)$ pour tout $k \in \vec{k}$, et $\mu_r(a)(i) = val(\eta(r)(i))(x_a)$ pour tout $a \in \vec{a}$ et $i \in \mathbb{Z}$.

Pour une valuation quelconque $\nu \in \mathcal{V}(A_\varphi)$, il existe $r \in \mathcal{R}(A_\varphi)$ tel que $\nu = val(r)$. Soit $M_\varphi(\nu) = \langle \iota_r, \mu_r \rangle$ la valuation des variables libres de φ qui correspond à r. Alors, on voit que M_φ définit une fonction $M_\varphi : \mathcal{V}(A_\varphi) \rightarrow (\vec{k} \mapsto \mathbb{Z}) \times (\vec{a} \mapsto {}^\omega\mathbb{Z}^\omega)$.

Theorème 5.8 *Soit $\varphi(\vec{k}, \vec{a})$ une combinaison booléenne positive de formules de type (F1)-(F3) et des contraintes PA sur les variables de bornes de tableaux $\vec{k}$, et A_φ l'automate défini ci-dessus Alors, $M_\varphi(\mathcal{V}(A_\varphi)) = [\![\varphi]\!]$.*

La preuve est par induction sur la structure de φ. Pour le cas de base, nous utilisons la correspondence entre modèles et graphes de contraintes des formules (F1)-(F3) (Lemme 5.7). Le pas inductif suit comme une conséquence du

fait que la classe des FBCA est fermée par union et intersection (Lemme 5.4). Le résultat principal de cette section est le suivant :

Corollaire 5.9 *La logique* **LIA** *est décidable.*

La preuve utilise la normalisation (voir le lemme 5.6) pour réécrire une formule de LIA en forme normale (NF) et applique ensuite le théorème 5.8 à la matrice de la formule (c.-à-d. la formule obtenu en enlevant les quantificateur existentielle).

5.3 Vérification automatique de programme avec tableaux

Dans cette section nous donnons une approche de vérification pour des programmes utilisant des tableaux d'entiers avec affectations, conditions et des boucles while *non-imbriquées*. Notre technique de vérification est basée sur une combinaison de la logique SIL (une sous-classe de LIA) présentée dans la section 5.3.4 utilisée pour exprimer des pré- et postconditions et les *automates et transducteurs à compteurs (CA)* utilisés pour traduire les formules SIL et les boucles du programme permettant de calculer l'effet des boucles et de vérifier l'implication.

Par exemple, la formule $(\forall i.0 \leq i \leq n_1 - 1 \rightarrow b[i] \geq 0) \ \wedge \ (\forall i.0 \leq i \leq n_2 - 1 \rightarrow c[i] < 0)$ décrit une postcondition d'un programme qui partitionne un tableau a dans un tableau b contenant ses éléments positifs et un tableau c contenant ses éléments négatifs.

Les formules SIL sont interprétées sur des *configurations* de programmes qui associent des entiers aux variables scalaires et des séquences finies d'entiers aux variables de tableaux. Comme montré dans la section précédente et dans [68] (où une preuve plus directe est donnée), l'ensemble des modèles d'une formule de $\exists^*\forall^*$-SIL correspond à l'ensemble de traces d'un automate à compteurs plat avec des contraintes DBM. Cela entraîne la décidabilité du problème de satisfaisabilité pour $\exists^*\forall^*$-SIL.

Dans cette section nous prenons un autre regard sur la connection entre $\exists^*\forall^*$-SIL et les automates à compteurs CA, qui nous permet de bénéficier des avantages des deux formalismes. En effet, la logique est utile pour exprimer des pré/postconditions lisibles de programmes et leurs parties et pour calculer

symboliquement l'effet de commandes (sans boucles). De l'autre côté, les automates sont adaptés pour exprimer l'effet de boucles de programmes.

En particulier, étant donnée une formule $\exists^*\forall^*$-SIL, nous pouvons facilement calculer la plus forte postcondition, qui est dans le même fragment, d'une affectation ou d'un conditionnel. En atteignant une boucle du programme, nous traduisons la formule $\exists^*\forall^*$-SIL φ qui décrit l'ensemble des configurations au début de la boucle vers un automate à compteurs A_φ qui code l'ensemble de ses modèles. Par la suite, pour caractériser l'effet d'une boucle L, nous la traduisons (d'une façon purement syntaxique) dans un *transducteur* T_L, c.-à-d. un automate à compteurs qui décrit la relation entrée/sortie sur les variables scalaires et les éléments des tableaux implémentée par L. La postcondition de L est ensuite obtenue en composant T_L avec A_φ. Le résultat de la composition est un automate à compteurs CA $B_{\varphi,L}$ représentant l'ensemble exact de configurations après un nombre quelconque d'itérations de L. À la fin, nous traduisons $B_{\varphi,L}$ vers $\exists^*\forall^*$-SIL et nous obtenons une postcondition de L par rapport à φ. Cependant, ce pas contient une abstraction (raffinable), puisque les automates à compteurs sont plus expressifs que $\exists^*\forall^*$-SIL. Nous générons d'abord un automate à compteurs plat, qui sur-approxime l'ensemble des traces de $B_{\varphi,L}$, et nous le traduisons ensuite vers $\exists^*\forall^*$-SIL.

Notre approche permet donc de générer des postconditions "lisibles" pour chaque boucle du programme permettant à l'utilisateur de comprendre ce que fait le programme. Puisque ces postconditions sont exprimées dans une logique décidable, cela permet en plus de tester automatiquement si elles impliquent les postconditions définies par l'utilisateur dans la même logique. Nous avons validé notre approche en vérifiant plusieurs programmes manipulant les tableaux, comme la partition d'un tableaux, la rotation d'un tableaux, etc.

Tous les détails se trouvent dans [33] et le rapport technique correspondant [34].

5.3.1 Travaux connexes

Le domaine de la vérification automatique de programmes avec tableaux et/ou la synthèse d'invariants de boucles est très étudié récemment. Par exemple, [57, 83, 17, 18, 75, 63] utilisent des gabarits d'invariants de boucles universellement quantifiés et/ou des prédicats définis par l'utilisateur. La forme des invariants recherchés est basée sur ces gabarits. Inférer les inva-

riants est abordé par plusieurs approches, telles que l'abstraction par prédicats avec de constantes de Skolem [57], la synthèse d'invariants basée sur les contraintes [17, 18], ou l'abstraction par prédicats combinée avec l'interpolation [75].

Dans [92], un prouveur par saturation et interpolation est utilisé pour dériver des invariants à partir de dépliages finis de boucles. Dans le travail récent [82], des invariants de boucles sont synthétisés en dérivant d'abord des invariants scalaires, en les combinant ensuite avec des axiomes de premier ordre sur les tableaux prédéfinis et enfin en utilisant un prouveur par saturation pour générer des invariants de boucles sur les tableaux. Cette approche peut générer des invariants contenant des alternations de quantificateurs. Un désavantage est que, contrairement à notre approche, la méthode ne tient pas compte de préconditions de boucles, qui sont parfois nécessaires pour trouver des invariants raisonnables. Cette approche ne génère pas en général des invariants dans un fragment logique décidable.

Une autre approche, basée sur l'interprétation abstraite, est utilisée dans [62]. Les tableaux sont partitionnés et des propriétés de "résumé" sont traquées. Le partitionnement est basé sur des heuristiques dépendant des valeurs des variables d'indices. Ces heuristiques échouent parfois et l'intervention de l'utilisateur devient nécessaire. Cette approche a été récemment amélioré dans [71] en utilisant des meilleurs heuristiques de partition et des meilleurs domaines relationnels abstraits pour garder la trace de relation entre deux parties de tableaux.

Les travaux sur les logiques de tableaux [36, 112, 60, 25, 61] ne donnent pas une méthode directe pour traiter automatiquement les boucles de programmes.

5.3.2 Préliminaires

Pour une séquence $\sigma \in A^*$, nous écrivons $|\sigma|$ pour sa longueur et σ_i pour l'élément à la position i pour $0 \leq i < |\sigma|$. Contrairement aux automates à compteurs pour LIA nous utilisons ici des automates à compteurs plus simples avec des calculs finis. Pour cette raison nous définissons les automates à compteurs munis explicitement d'un état initial et des états finaux. Un *automate à compteurs fini* (FCA) est un quintuple $A = \langle X, Q, I, \rightarrow, F \rangle$, où : X est un ensemble fini de compteurs sur $\mathbb{Z}$, Q est un ensemble fini d'états de contrôle, $I \subseteq Q$ est un ensemble d'états initiaux, $\rightarrow$ est une relation de

transition donné par un ensemble de règles $q \xrightarrow{\varphi(X,X')} q'$ où φ est une formule arithmétique reliant les valeurs actuelles des compteurs X à leur valeurs futures $X' = \{x' \mid x \in X\}$, et $F \subseteq Q$ est un ensemble fini d'états finaux.

Les notions de *configuration* et *successeur immédiat* sont les mêmes que celles la section 5.2.2. Étant donnés deux états de contrôle $q, q' \in Q$, un *calcul* de A de q à q' est une séquence finie de configurations $c_1 c_2 \ldots c_n$ avec $c_1 = \langle q, \nu \rangle$, $c_n = \langle q', \nu' \rangle$ pour des valuations $\nu, \nu' : X \to \mathbb{Z}$, et c_{i+1} est un successeur immédiat de c_i, pour tout i avec $1 \leq i < n$. Soit $\mathcal{R}(A)$ l'ensemble de calculs de A d'un état initial $q_0 \in I$ vers un état final $q_f \in F$ et $Tr(A) = \{val(c_1)val(c_2)\ldots val(c_n) \mid c_1 c_2 \ldots c_n \in \mathcal{R}(A)\}$ son ensemble de *traces*.

Pour deux automates à compteurs finis $A_i = \langle X_i, Q_i, I_i, \rightarrow_i, F_i \rangle,\ i = 1, 2$ nous définissons l'*automate produit* comme $A_1 \otimes A_2 = \langle X_1 \cup X_2, Q_1 \times Q_2, I_1 \times I_2, \rightarrow, F_1 \times F_2 \rangle$, où $\langle q_1, q_2 \rangle \xrightarrow{\varphi} \langle q_1', q_2' \rangle$ ssi $q_1 \xrightarrow{\varphi_1}_1 q_1'$, $q_2 \xrightarrow{\varphi_2}_2 q_2'$ et $\models \varphi \leftrightarrow \varphi_1 \wedge \varphi_2$. Nous avons pour toute séquence $\sigma \in Tr(A_1 \otimes A_2)$, $\sigma\downarrow_{X_1} \in Tr(A_1)$ et $\sigma\downarrow_{X_2} \in Tr(A_2)$, et vice-versa.

5.3.3 Automates à compteurs pour reconnaître les configurations et les transitions

Pour le reste de la section, soit $\vec{a} = \{a_1, a_2, \ldots, a_k\}$ un ensemble de *variables de tableau*, et $\vec{b} = \{b_1, b_2, \ldots, b_m\}$ un ensemble de *variables scalaires*. Une *configuration* $\langle \alpha, \iota \rangle$ est une paire de valuations $\alpha : \vec{a} \to \mathbb{Z}^*$, et $\iota : \vec{b} \to \mathbb{Z}$. Pour simplifier, nous supposons que $|\alpha(a_1)| = |\alpha(a_2)| = \ldots = |\alpha(a_k)| > 0$, et notons $|\alpha|$ la taille des tableaux dans la configuration.

Par la suite, soit X un ensemble de compteurs partitionné dans des *compteurs de valeurs* $\vec{x} = \{x_1, x_2, \ldots, x_k\}$, *compteurs d'indices* $\vec{i} = \{i_1, i_2, \ldots, i_k\}$, *paramètres* $\vec{p} = \{p_1, p_2, \ldots, p_m\}$, et *compteurs de travail* $\vec{w}$. Notons que $\vec{a}$ est en correspondance directe avec $\vec{x}$ et $\vec{i}$, et que $\vec{b}$ est en correspondance avec $\vec{p}$.

Definition 5.10 *Soit $\langle \alpha, \iota \rangle$ une configuration. Une séquence $\sigma \in (X \to \mathbb{Z})^*$ est appelé* consistante *avec $\langle \alpha, \iota \rangle$, noté $\sigma \vdash \langle \alpha, \iota \rangle$ ssi, pour tout $1 \leq p \leq k$, et tout $1 \leq r \leq m$:*

1. *pour tout $q \in \mathbb{N}$ avec $0 \leq q < |\sigma|$, nous avons $0 \leq \sigma_q(i_p) \leq |\alpha|$,*
2. *pour tout $q, r \in \mathbb{N}$ avec $0 \leq q < r < |\sigma|$, nous avons $\sigma_q(i_p) \leq \sigma_r(i_p)$,*

3. *pour tout* $s \in \mathbb{N}$ *avec* $0 \leq s \leq |\alpha|$, *il existe* q *avec* $0 \leq q < |\sigma|$ *tel que* $\sigma_q(i_p) = s$,
4. *pour tout* $q \in \mathbb{N}$ *avec* $0 \leq q < |\sigma|$, *si* $\sigma_q(i_p) = s < |\alpha|$, *alors* $\sigma_q(x_p) = \alpha(a_p)_s$,
5. *pour tout* $q \in \mathbb{N}$ *avec* $0 \leq q < |\sigma|$, *nous avons* $\sigma_q(p_r) = \iota(b_r)$.

Intuitivement, un calcul d'un FCA représente le contenu d'un tableau en traversant toutes ses entrées de gauche à droite. Le contenu de plusieurs tableaux est représenté en entrelaçant d'une façon arbitraire le parcours des différents tableaux (contrairement à la représentation des tableaux dans LIA). De ce point de vue, pour qu'un calcul d'un FCA corresponde à une configuration (c.-à-d. soit consistant avec elle), il est nécessaire que chaque compteur d'indice garde sa valeur ou incrémente à chaque point du calcul (voir point 2 de la définition 5.10) en visitant chaque entrée du tableau (points 1 et 3 de la définition 5.10). [4] La valeur d'une certaine entrée d'un tableau a_p est codée par la valeur du compteur de tableaux x_p quand le compteur d'indice contient la position de l'entrée donnée (point 4 de la définition 5.10). Enfin, les valeurs des variables scalaires sont codées par les valeurs des paramètres correspondants qui restent constants pendant le calcul (point 5 de la définition 5.10).

Un FCA est appelée *conf-consistant* ssi pour chaque trace $\sigma \in Tr(A)$, il existe une configuration (qui est unique) $\langle \alpha, \iota \rangle$ telle que $\sigma \vdash \langle \alpha, \iota \rangle$. Nous notons $\Sigma(A) = \{\langle \alpha, \iota \rangle \mid \exists\, \sigma \in Tr(A) \,.\, \sigma \vdash \langle \alpha, \iota \rangle\}$ l'ensemble des configurations reconnu par un FCA.

Une conséquence de la définition 5.10 est qu'entre deux positions adjacentes d'une trace d'un FCA conf-consistant les compteurs d'indice n'incrémentent jamais de plus qu'un. Par conséquent, chaque transition dont la relation est non-déterministe par rapport à un compteur d'indice peut être séparée en deux transitions : un *idle* (pas de changement) et un *tic-tac* (incrément par un). Par la suite, nous supposons donc, que chaque transition d'un FCA conf-consistant et soit un idle soit un tic-tac par rapport à chaque compteur d'indice.

[4] En fait, chaque compteur d'indice atteint la valeur $|\alpha|$ qui est un de plus que nécessaire pour traverser un tableau avec des entrées $0 \ldots |\alpha| - 1$. La raison est technique, liée à la composition avec les transducteurs représentant les boucles du programme. Ils produisent des entrée de tableaux avec un délai d'un pas. Notons que l'entrée à la position $|\alpha|$ n'est pas contrainte.

Pour chaque ensemble $U = \{u_1, ..., u_n\}$, nous notons $U^i = \{u_1^i, ..., u_n^i\}$ et $U^o = \{u_1^o, ..., u_n^o\}$. Si $s = \langle \alpha, \iota \rangle$ et $t = \langle \beta, \kappa \rangle$ sont deux configurations tels que $|\alpha| = |\beta|$, la paire $\langle s, t \rangle$ est appelée une *transition*. Un FCA $T = \langle X, Q, I, \rightarrow, F \rangle$ est appelé un *transducteur* ssi son ensemble de compteurs X est partitionné dans : *compteurs d'entrée* $\vec{x}^i$ et *compteurs de sortie* $\vec{x}^o$, où $\vec{x} = \{x_1, x_2, \ldots, x_k\}$, *compteurs d'indices* $\vec{i} = \{i_1, i_2, \ldots, i_k\}$, *paramètres d'entrée* $\vec{p}^i$ et *paramètres de sortie* $\vec{p}^o$, où $\vec{p} = \{p_1, p_2, \ldots, p_m\}$, et *compteurs de travail* $\vec{w}$.

Definition 5.11 *Une séquence $\sigma \in (X \rightarrow \mathbb{Z})^*$ est appelée* consistante *avec une transition $\langle s, t \rangle$, où $s = \langle \alpha, \iota \rangle$ et $t = \langle \beta, \kappa \rangle$, notée $\sigma \vdash \langle s, t \rangle$ ssi, pour tout p avec $1 \leq p \leq k$ et tout r avec $1 \leq r \leq m$:*

1. *pour tout $q \in \mathbb{N}$ avec $0 \leq q < |\sigma|$, nous avons $0 \leq \sigma_q(i_p) \leq |\alpha|$,*
2. *pour tout $q, r \in \mathbb{N}$ avec $0 \leq q < r < |\sigma|$, nous avons $\sigma_q(i_p) \leq \sigma_r(i_p)$,*
3. *pour tout $s \in \mathbb{N}$ avec $0 \leq s \leq |\alpha|$, il existe p avec $0 \leq q < |\sigma|$ tel que $\sigma_q(i_p) = s$,*
4. *pour tout $q \in \mathbb{N}$ avec $0 \leq q < |\sigma|$, si $\sigma_q(i_p) = s < |\alpha|$, alors $\sigma_q(x_p^i) = \alpha(a_p)_s$,*
5. *pour tout $q \in \mathbb{N}$ avec $0 \leq q < |\sigma|$, si $\sigma_q(i_p) = s > 0$, alors $\sigma_q(x_p^o) = \beta(a_p)_{s-1}$,*
6. *pour tout $q \in \mathbb{N}$ avec $0 \leq q < |\sigma|$, nous avons $\sigma_q(p_r^i) = \iota(b_r)$ et $\sigma(p_r^o) = \kappa(b_r)$.*

L'intuition derrière la représentation des transitions d'un programme avec des tableaux par des transducteurs est la même que celle pour la représentation des configurations par des automates à compteurs. Les transducteurs décrivent les configurations entrée et sortie en utilisant les compteurs d'entrée et de sortie respectivement. Notons que les transducteurs sont définis de sorte que les valeurs de sortie apparaissent avec un délai d'un pas par rapport aux valeurs d'entrée correspondantes (voir point 5 de la définition 5.11). [5]

Un transducteur T est appelé *trans-consistant* ssi pour chaque trace $\sigma \in Tr(T)$ il existe une transition $\langle s, t \rangle$ telle que $\sigma \vdash \langle s, t \rangle$. Nous notons $\Theta(T) = \{\langle s, t \rangle \mid \exists\, \sigma \in Tr(T) \,.\, \sigma \vdash \langle s, t \rangle\}$ l'ensemble des transitions reconnu par un transducteur.

[5] Intuitivement, le transducteur à besoin d'un pas pour calculer la valeur de sortie. Il est possible de définir des transducteurs synchrones. Nous préférons cette définition pour des raisons techniques.

5.3.3.1 Dépendances entre les compteurs d'indice

Un automate à compteurs conf-consistant peut représenter un tableau de plusieurs façons. Par exemple, le tableau $a = 4, 3, 2$ est codé par les deux calculs $(0, 4), (0, 4), (1, 3), (2, 2), (2, 2)$ et $(0, 4), (1, 3), (1, 3), (2, 2)$, où les premiers éléments des paires sont les valeurs du compteur d'indice et les deuxièmes les valeurs du compteur de valeur correspondant à a. Deux ou plus de compteurs d'indices sont appelés *dépendants* s'ils avancent toujours à la même vitesse ensemble. Cette notion est utilisée pour donner des conditions suffisantes permettant la composition d'automate à compteurs avec des transducteurs.

Soit $X \subset \vec{i}$ un ensemble fixé de compteurs d'indice. Une *dépendance* δ est une conjonction d'égalités entre des éléments appartenant à X. Pour une séquence de valuations $\sigma \in (X \rightarrow \mathbb{Z})^*$, nous notons $\sigma \models \delta$ ssi $\sigma_l \models \delta$, pour tout $0 \leq l < |\sigma|$.

Pour une dépendance δ, nous notons $[\![\delta]\!] = \{\sigma \in (X \rightarrow \mathbb{Z})^* \mid$ il existe une configuration s tel que $\sigma \vdash s$ et $\sigma \models \delta\}$, c.-à-d., l'ensemble de toutes les séquences qui correspondent à un tableau et qui satisfont δ. Une dépendance δ_1 est appelée *plus forte* qu'une autre dépendance δ_2, notée $\delta_1 \rightarrow \delta_2$, ssi l'implication de premier ordre entre δ_1 et δ_2 est valide. Notons que $\delta_1 \rightarrow \delta_2$ ssi $[\![\delta_1]\!] \subseteq [\![\delta_2]\!]$. Si $\delta_1 \rightarrow \delta_2$ et $\delta_2 \rightarrow \delta_1$, nous écrivons $\delta_1 \leftrightarrow \delta_2$. Pour un automate à compteurs conf-consistant (resp. trans-consistant) A, nous notons par $\Delta(A)$ la dépendance la plus forte δ telle que $Tr(A) \subseteq [\![\delta]\!]$.

Definition 5.12 *Un FCA $A = \langle \vec{x}, Q, I, \rightarrow, F \rangle$, où $\vec{x} \subseteq X$, est appelé* conf-complet *ssi pour toute configuration $s \in \Sigma(A)$, et toute séquence $\sigma \in (X \rightarrow \mathbb{Z})^*$, telle que $\sigma \vdash s$ et $\sigma \models \Delta(A)$, nous avons $\sigma \in Tr(A)$.*

Intuitivement, un automate A est conf-complet s'il représente chaque configuration $s \in \Sigma(A)$ de toutes les façons possibles par rapport à la dépendance la plus forte sur ses compteurs d'indices.

5.3.3.2 Composer les automates à compteurs avec les transducteurs

Pour un automate à compteurs A et un transducteur T, $\Sigma(A)$ représente un ensemble de configurations, tandis que $\Theta(T)$ est la relation de transition. Une question naturelle est de savoir, si l'image de $\Sigma(A)$ par la relation $\Theta(T)$ peut être représentée par un FCA, et si cet automate peut être construit à partir de A et de T.

Theorème 5.13 *Si A est un automate à compteurs conf-consistant et conf-complet avec des compteurs de valeurs $\vec{x} = \{x_1, ..., x_k\}$, compteurs d'indice $\vec{i} = \{i_1, ..., i_k\}$, et paramètres $\vec{p} = \{p_1, ..., p_m\}$, et T est un transducteur avec compteurs d'entrée (sortie) $\vec{x}^i$ ($\vec{x}^o$), compteurs d'indice $\vec{i}$, et paramètres d'entrée (sortie) $\vec{p}^i$ ($\vec{p}^o$) tel que $\Delta(T)[\vec{x}/\vec{x}^i] \rightarrow \Delta(A)$, alors on peut construire un automate à compteurs conf-consistant B, tel que $\Sigma(B) = \{t \mid \exists\ s \in \Sigma(A)\ .\ \langle s, t\rangle \in \Theta(T)\}$, et de plus, $\Delta(B) \rightarrow \Delta(T)[\vec{x}/\vec{x}^i]$.*

5.3.4 La logique SIL

Dans cette section nous introduisons SIL (Singly Index Logique) qui est une version allégée et légèrement modifiée de LIA, notamment un seul indice est permis dans les propriétés de tableaux et les contraintes modulo sur les indices ne sont pas autorisées. Comme pour LIA nous avons trois types de variables. Les *variables scalaires* $b, b_1, b_2, ... \in BVar$ apparaissent dans les bornes qui définissent les intervals et dans les contraintes sur les variables qui ne sont pas des tableaux. Les *variables d'indices* $i, i_1, i_2, ... \in IVar$ et les *variables de tableaux* $a, a_1, a_2, ... \in AVar$ sont utilisées dans les termes de tableaux. Les ensembles $BVar$, $IVar$, et $AVar$ sont disjoints.

$n, m, p \ldots$	$\in$	$\mathbb{Z}$	constantes entières
$b, b_1, b_2, \ldots$	$\in$	$BVar$	variables scalaires
ϕ			contraintes de Presburger
$i, j, i_1, i_2, \ldots$	$\in$	$IVar$	variables d'indices
$a, a_1, a_2, \ldots$	$\in$	$AVar$	variables de tableaux
$\sim$	$\in$	$\{\leq, \geq\}$	

B	$:=$	$n \mid b + n$	termes de bornes de tableaux
G	$:=$	$\top \mid B \leq i \leq B \mid G \wedge G \mid G \vee G$	expressions de garde
V	$:=$	$a[i+n] \sim B \mid a_1[i+n] - a_2[i+m] \sim p \mid$	
		$i - a[i+n] \sim m \mid V \wedge V$	expressions de valeur
F	$:=$	$\forall i\ .\ G \rightarrow V \mid \phi(B_1, B_2, \ldots, B_n) \mid$	
		$\neg F \mid F\ \wedge\ F$	formules

FIG. 5.4 – Syntaxe de SIL

La figure 5.4 montre la syntaxe de SIL. Nous autorisons des comparaisons

entre i et $a[i+n]$ qui ne sont pas explicitement autorisées en LIA.

Comme dans LIA, nous écrivons $i < f$ à la place de $i \leq f - 1$, $i = f$ à la place de $f \leq i \leq f$, $\varphi_1 \vee \varphi_2$ à la place de $\neg(\neg\varphi_1 \wedge \neg\varphi_2)$, et $\forall i \,.\, \upsilon(i)$ à la place de $\forall i \,.\, \top \rightarrow \upsilon(i)$. Si $B_1(b_1), ..., B_n(b_n)$ sont des termes de bornes avec des variables libres $b_1, ..., b_n \in BVar$, nous écrivons chaque formule de Presburger φ sur les termes $a_1[B_1], ..., a_n[B_n]$ comme une abréviation pour $(\bigwedge_{k=1}^{n} \forall j \,.\, j = B_k \rightarrow a_k[j] = b'_k) \wedge \varphi[b'_1/a_1[B_1], ..., b'_n/a_n[B_n]]$, où $b'_1, ..., b'_n$ sont des variable fraîches de scalaires.

La sémantique d'une formule φ est définie en utilisant la relation $\langle\alpha, \iota\rangle \models \varphi$ entre configurations et formules En particulier, $\langle\alpha, \iota\rangle \models \forall i \,.\, \gamma(i, \vec{b}) \rightarrow \upsilon(i, \vec{a}, \vec{b})$ ssi, pour toute valeur de n dans l'ensemble $\bigcap\{[-m, |\alpha| - m - 1] \mid a[i+m]$ apparaît dans $\upsilon\}$, si $\iota \models \gamma[n/i]$, alors aussi $\iota \cup \alpha \models \upsilon[n/i]$. Intuitivement, γ est considéré uniquement pour les indices qui ne génèrent pas des accès de tableaux hors bornes.

Nous notons $[\![\varphi]\!] = \{\langle\alpha, \iota\rangle \mid \langle\alpha, \iota\rangle \models \varphi\}$. Alors le *problème de satisfaisabilité* demande, si pour une formule φ, $[\![\varphi]\!] \stackrel{?}{=} \emptyset$. Nous disons qu'un automate A et une formule SIL se correspondent ssi $\Sigma(A) = [\![\varphi]\!]$.

Le fragment $\exists^*\forall^*$ de SIL est l'ensemble des formules SIL qui écrit en forme normale prenexe ont un préfixe de quantificateur de la forme $\exists i_1 \ldots \exists i_n \forall j_1 \ldots \forall j_m$. Comme montré dans [68] le fragment $\exists^*\forall^*$ de SIL est équivalent à l'ensemble de combinaisons booléennes existentiellement quantifiés de (1) contraintes PA sur les variables scalaires $\vec{b}$ et (2) propriétés de tableaux de la forme $\forall i \,.\, \gamma(i, \vec{b}) \rightarrow \upsilon(i, \vec{b}, \vec{a})$.

Theorème 5.14 ([68]) *Le problème de satisfaisabilité est décidable pour le fragment $\exists^*\forall^*$ de SIL.*

Nous donnons ci-dessous une connexion automate logique entre $\exists^*\forall^*$-SIL et automates à compteurs. Nous montrons comment des préconditions de boucles écrites en $\exists^*\forall^*$-SIL peuvent être traduite vers des automates à compteurs CA de sorte qu'on peut les composer avec des transducteurs correspondant aux boucles (pour cette raison la traduction décrite dans [33] est différente de celle de [68]). Ensuite nous montrons comment, des formules de $\exists^*\forall^*$-SIL peuvent être obtenues à partir d'un automate à compteurs qui est obtenue comme un produit entre l'automate à compteurs d'une précondition et le transducteur d'une boucle.

5.3.4.1 De $\exists^*\forall^*$-SIL aux automates à compteurs

Étant donnée une précondition φ en $\exists^*\forall^*$-SIL, nous pouvons construire un automate A_φ avec $\Sigma(A) = [\![\varphi]\!]$. Nous supposons que la précondition est satisfaisable (ce qui peut être testé, voir théorème 5.14). L'idée de la construction (donnée dans [33]) est similaire aux automates construits pour LIA.

La plus forte dépendance $\Delta(A_\varphi)$ entre les compteurs d'indice de A_φ peut être déterminé de considérant la formule φ. Nous pouvons aussi montrer que A_φ est conf-complet (voir la définition 5.12). Puisque A_φ est construit inductivement sur la structure de φ, $\Delta(A_\varphi)$ peut aussi être construit inductivement. Soit $\delta(\varphi)$ la formule définie comme suit :

- $\delta(\varphi) = \top$ si φ est une contrainte de Presburger sur $\vec{b}$,
- pour $\varphi \equiv \forall i \ . \ f \leq i \leq g \rightarrow v$,
  $\delta(\varphi) \triangleq \delta(\neg\varphi) \triangleq \begin{cases} \top & \text{si } v \text{ est } a_p[i] \sim B \text{ ou } i - a_p[i] \sim n, \\ i_p = i_q & \text{si } v \text{ est } a_p[i] - a_q[i+1] \sim n, \end{cases}$
- $\delta(\varphi_1 \wedge \varphi_2) = \delta(\varphi_1) \wedge \delta(\varphi_2)$.

Theorème 5.15 *Étant donnée une formule satisfaisable φ de $\exists^*\forall^*$-SIL, nous avons pour le FCA A_φ : (1) A_φ est conf-consistant, (2) A_φ est conf-complet, (3) A_φ et φ correspondent, et (4) $\delta(A_\varphi) \leftrightarrow \Delta(A_\varphi)$.*

5.3.4.2 Des automates à compteurs à $\exists^*\forall^*$-SIL

En général les automates à compteurs sont plus expressifs que $\exists^*\forall^*$-SIL. Pour donner une formule qui correspond à un automate nous donnons d'abord une construction qui permet d'obtenir à partir d'un FCA A un ensemble de FCA restreints $\mathcal{A}_1^K, \mathcal{A}_2^K, \ldots, \mathcal{A}_n^K$, tel que $\Sigma(A) \subseteq \bigcap_{i=1}^n \Sigma(\mathcal{A}_i^K)$, et pour chaque $\mathcal{A}_i^K$, $1 \leq i \leq n$, nous pouvons donner une formule $\exists^*\forall^*$-SIL φ_i qui lui correspond. Nous obtenons donc une formule $\varphi_A = \bigwedge_{i=1}^n \varphi_i$ telle que $\Sigma(A) \subseteq [\![\varphi_A]\!]$ (la formule est donc une sur-approximation de l'automate). Intuitivement, les FCA restreints sont des automates plats dont les transitions sont des DBMs. L'abstraction consiste à aplatir (d'une façon raffinable) les automates en tenant compte des dépendances entre les compteurs d'indices. Chaque automate de $\mathcal{A}_1^K, \mathcal{A}_2^K, \ldots, \mathcal{A}_n^K$ ne concerne que des compteurs d'indice dépendants.

$b \in BVar$, $a \in AVar$, $i \in IVar$, $n \in \mathbb{Z}$, $c \in \mathbb{N}$

$$\begin{array}{lll} ASGN & ::= & LHS = RHS \\ LHS & ::= & b \mid a[i+c] \\ TRM & ::= & LHS \mid i \\ RHS & ::= & TRM \mid \texttt{-}TRM \mid TRM\texttt{+}n \\ CND & ::= & CND \;\texttt{\&\&}\; CND \mid RHS \leq RHS \end{array}$$

Fig. 5.5 – Affectations et conditions

5.3.5 Les programmes qui manipulent les tableaux

Nous considérons des programmes avec des affectations, des conditions et des boucles while non-imbriquées montrées dans la figure 5.6, qui travaillent sur des tableaux $AVar$ et des variables scalaires $BVar$ (for une syntaxe formel, voir [34]). Dans une boucle, nous supposons une correspondance 1 :1 entre l'ensemble des tableaux $AVar$ et l'ensemble des indices $IVar$. Autrement dit, à chaque tableau est associé une variable d'indice. Chaque indice $i \in IVar$ est initialisé au début de la boucle en utilisant une expression de la forme $b+n$ où $b \in BVar$ et $n \in \mathbb{Z}$. Les indices sont locaux à la boucle. Le corps S_1^l ;... ;$S_{n_l}^l$; de chaque branche de boucle consiste en zéro ou plusieurs affectations suivies par une commande d'incrémentation des indices $\texttt{incr}(I)$, $I \subseteq IVar$. La syntaxe des affectations et des expressions booléennes est donnée dans la figure 5.5.

Une *configuration d'un programme* est une paire $\langle l, s \rangle$ où l est une ligne du programme et s est une configuration $\langle \alpha, \iota \rangle$ définie comme dans la section 5.3.3. La sémantique des commandes est standard (voir par exemple [87]). Nous supposons qu'il n'y a pas d'accès hors bornes des tableaux (ces situations sont traitées dans [34]).

Pour les commandes données dans la figure 5.5, nous avons développé un calcul de plus fortes postconditions pour $\exists^*\forall^*$-SIL. Ce calcul permet d'exprimer la sémantique des affectations et des conditions et peut être utilisé pour traiter les parties séquentielles du programme (les blocs de commandes en dehors des boucles). On peut montrer que $\exists^*\forall^*$-SIL est fermée par les plus fortes postconditions (voir [34]).

5.3.6 Des boucles aux automates à compteurs

Étant donnée une boucle L qui commence à la ligne l, tel que l' est la ligne qui suit directement L, nous notons $\Theta_L = \{\langle s,t\rangle \mid$ il y a un calcul de L de $\langle l,s\rangle$ vers $\langle l',t\rangle\}$ la relation de transition induite par L. Nous définissons la *dépendance de boucle* δ_L comme la conjonction des égalités $i_p = i_q,\ i_p, i_q \in IVar$, où (1) $e_p \equiv e_q$ où e_1 et e_2 sont les expressions initialisant i_p et i_q et (2) pour chaque branche de L finit par un incrément d'indice $\mathtt{incr}(I)$, $i_p \in I \iff i_q \in I$. La relation d'équivalence $\simeq_{\delta_L}$ sur les compteurs d'indice est définie comme avant : $i_p \simeq_{\delta_L} i_q$ ssi $\models \delta_L \rightarrow i_p = i_q$.

$while_{a_1:i_1=e_1,\ldots,a_k:i_k=e_k}\ (C)$
  $if\ (C_1)\ S_1^1\ ;\ldots;S_{n_1}^1\ ;$
  $else\ if\ (C_2)\ S_1^2\ ;\ldots;S_{n_2}^2\ ;$
  ...
  $else\ if\ (C_{h-1})\ S_1^{h-1}\ ;\ldots;S_{n_{h-1}}^{h-1}\ ;$
  $else\ S_1^h\ ;\ldots;S_{n_h}^h\ ;$

FIG. 5.6 – Une boucle while

Nous supposons avoir une boucle L comme dans la figure 5.6 où $AVar = \{a_1,\ldots,a_k\}$, $IVar = \{i_1,\ldots,i_k\}$, et $BVar = \{b_1,\ldots,b_m\}$ sont respectivement les ensemble de variables de tableaux, indices, et scalaires. Soient $I_1, I_2,\ldots,I_n \subseteq IVar$ une partition de $IVar$ en classes d'équivalence induites par $\simeq_{\delta_L}$. Pour une condition, affectation, incrément d'indices ou une boucle entière E nous définissons $d_E : AVar \rightarrow \mathbb{N} \cup \{\perp\}$ as $d_E(a) = max\{c \mid a[i+c]$ apparaît dans $E\}$ si a est utilisé dans E, et $d_E(a) = \perp$ sinon. Le transducteur $T_L = \langle X, Q, \{q_0\}, \rightarrow, \{q_{fin}\}\rangle$, qui correspond à la boucle L, est défini ci-dessous :

- $X = \{x_r^i, x_r^o, i_r \mid 1 \leq r \leq k\} \cup \{w_{r,l}^i \mid 1 \leq r \leq k, 1 \leq l \leq d_L(a_r)\} \cup \{w_{r,l}^o \mid 1 \leq r \leq k,\ 0 \leq l \leq d_L(a_r)\} \cup \{p_r^i, p_r^o, w_r \mid 1 \leq r \leq m\} \cup \{w_N\}$ où $x_r^{i/o}$, $1 \leq r \leq k$, sont les compteurs de tableaux entrée/sortie, $p_r^{i/o}$, $1 \leq r \leq k$, les paramètres mémorisant les scalaires entrée/sortie, et w_r, $1 \leq r \leq m$, sont des compteurs de travail utilisés pour manipuler les tableaux et les scalaires (w_N mémorise la longueur commune des tableaux).
- $Q = \{q_0, q_{pre}, q_{loop}, q_{suf}, q_{fin}\} \cup \{q_l^r \mid 1 \leq r \leq h, 0 \leq l < n_r\}$.

- Les règles de transition de T_L sont les suivantes. Nous supposons une contrainte implicite $x' = x$ pour chaque compteur $x \in X$ tel que x' n'apparaît par explicitement :
  - $q_0 \xrightarrow{\varphi} q_{pre}$, $\varphi = \bigwedge_{1 \le r \le m}(w_r = p^i_r) \wedge w_N > 0 \wedge \bigwedge_{1 \le r \le k}(i_r = 0 \wedge x^i_r = w^o_{r,0}) \wedge \bigwedge_{\substack{1 \le r \le k \\ 1 \le l \le d_L(a_r)}}(w^i_{r,l} = w^o_{r,l})$ (les compteurs sont initialisés).
  - Pour chaque classe d'équivalence I_j de $\simeq_{\delta_L}$, $1 \le j \le n$, $q_{pre} \xrightarrow{\varphi} q_{pre}$ avec $\varphi = \bigwedge_{1 \le r \le k}(i_r < \xi(e_r)) \wedge \xi(incr(I))$ (T_L copie la partie initiale des tableaux non modifiée par L).
  - $q_{pre} \xrightarrow{\varphi} q_{loop}$, $\varphi = \bigwedge_{1 \le r \le k} i_r = \xi(e_r)$ (T_L commence la simulation de L).
  - Pour chaque $1 \le l \le h$, $q_{loop} \xrightarrow{\varphi} q^l_0$, $\varphi = \xi(C) \wedge \bigwedge_{1 \le r < l}(\neg \xi(C_r)) \wedge \xi(C_l)$ où $C_h = \top$ (T_L choisit la branche de la boucle à simuler).
  - Pour chaque $1 \le l \le h$, $1 \le r \le n_l$, $q^l_{r-1} \xrightarrow{\xi(S^l_r)} q$ où $q = q^l_r$ si $r < n_l$, et $q = q_{loop}$ sinon (l'automate simule une branche de la boucle).
  - $q_{loop} \xrightarrow{\varphi} q_{suf}$, $\varphi = \neg \xi(C) \wedge \bigwedge_{1 \le r \le m}(w_r = p^o_r)$ (T_L a terminé la simulation d'une exécution de L).
  - Pour chaque classe d'équivalence I_j de $\simeq_{\delta_L}$, $1 \le j \le n$, et $i_r \in I_j$, $q_{suf} \xrightarrow{\varphi} q_{suf}$, $\varphi = i_r < w_N \;\wedge\; \xi(incr(I_j))$ (copie les suffixes des tableaux non-modifiés par la boucle)
  - $q_{suf} \xrightarrow{\varphi} q_{fin}$, $\varphi = \bigwedge_{1 \le r \le k} i_r = w_N$ (tous les tableaux sont complètement traités).

La transformation syntaxique ξ d'affectations et de conditions préserve la structure de ces expression, mais remplace chaque b_r par le compteur w_r et chaque $a_r[i_r + c]$ par $w^o_{r,c}$ pour $b_r \in BVar$, $a_r \in AVar$, $i_r \in IVar$, et $c \in \mathbb{N}$. Dans les parties gauches des affectations, les valeurs futures des compteurs sont utilisées (voir [34]). Pour les incréments nous définissons, pour chaque $i_r \in IVar$:

- $\xi(incr(i_r))$: $x^{i\,\prime}_r = w^i_{r,1} \wedge \bigwedge_{1 < l \le d_L(a_r)} w^{i\prime}_{r,l-1} = w^i_{r,l} \wedge x^{o\prime}_r = w^o_{r,0} \wedge \bigwedge_{0 < l \le d_L(a_r)} w^{o\prime}_{r,l-1} = w^o_{r,l} \wedge w^{i\prime}_{r,d_L(a_r)} = w^{o\prime}_{r,d_L(a_r)} \wedge i'_r = i_r + 1$, si $d_L(a_r) > 0$,
- $\xi(incr(i_r))$: $x^{i\,\prime}_r = w^{o\prime}_{r,0} \wedge x^{o\prime}_r = w^o_{r,0} \wedge i'_r = i_r + 1$, si $d_L(a_r) = 0$.

Pour l'incrément d'un ensemble d'indices cette définition est étendue “point par point”.

L'idée principale de la construction est la suivante. T_L préserve les séquences exactes des opérations faites sur les tableaux et les scalaires dans L, mais les exécute sur des compteurs convenablement choisis. Nous exploitons le fait que le programme accède aux tableaux uniquement à travers d'une "fenêtre" de taille bornée, qui bouge de gauche à droite. Le contenu de cette fenêtre est mémorisé dans les compteurs de travail dont la sémantique change à chaque pas d'incrément. En particulier, la valeur initiale d'une cellule de tableau $a_r[l]$ est mémorisée dans $w^o_{r,d_L(a_r)}$ pour $d_L(a_r) > 0$ (le cas de $d_L(a_r) = 0$ est un peu plus simple). Cette valeur peut être accédé et/ou modifié ensuite via $w^o_{r,q}$ où $q \in \{d_L(a_r), ..., 0\}$ dans les itérations $l - d_L(a_r), ..., l$ respectives, car $w^o_{r,q}$ est copié dans $w^o_{r,q-1}$ pendant la simulation de $incr(i_r)$ pour $q > 0$. En même temps, la valeur initiale de $a_r[l]$ est mémorisée dans $w^i_{r,d_L(a_r)}$, qui est ensuite copié dans $w^i_{r,q}$ pour $q \in \{d_L(a_r) - 1, ..., 1\}$ et finalement dans x^i_r, ce qui arrive exactement quand i_r atteint la valeur l. Pendant la simulation du prochain $incr(i_r)$, la valeur finale de $a_r[l]$ apparaît dans x^o_r, ce qui est en accord avec la façon dont un transducteur exprime un changement dans une certaine cellule de tableau (voir la définition 5.11). Les valeurs d'entrée des scalaires sont mémorisées dans les paramètres p^i_r et initialement copiées dans les compteurs de travail w_r. Ces compteurs sont modifiés tout au long des calculs de T_L et finalement copié dans les paramètres de sortie p^o_r.

La correction de la traduction est donné par le théorème suivant.

Theorème 5.16 *Étant donnée une boucle de programme L, nous avons :*

1. *T_L est un transducteur trans-consistant*
2. *$\Theta(L) = \Theta(T_L)$, et*
3. *$\Delta(T_L) \rightarrow \delta_L$.*

Le dernier point assure que δ_L est une sur-approximation des dépendances de compteurs d'indice de T_L. Cette sur-approximation est utilisé dans le théorème 5.13 pour tester si l'image d'un automate d'une précondition A peut être calculé, en testant $\delta_T \rightarrow \Delta(A)$. Pour satisfaire les exigences du théorème 5.13, on peut étendre T_L d'une façon simple pour copier de l'entrée vers la sortie toutes les valeurs des variables qui apparaissent dans le programme mais pas dans L.

Tab. 5.2 – Exemples

programme	états de contrôle	compteurs
`init`	4	8
`partition`	4	24
`insertion`	7	19
`rotate`	4	15

5.3.7 Exemples

Pour valider notre approche, nous avons fait des expériences avec plusieurs programmes qui manipulent des tableaux d'entiers La table 5.2 donne la taille des automates après la boucle principale du programme en nombre d'états de contrôle et en nombre de compteurs. Les automates ont été légèrement optimisés en utilisant des techniques d'analyse statique (élimination de compteurs inutiles, etc.) Ces résultats montre que les automates obtenu sont assez simples. Plus de détails se trouvent dans [34].

L'exemple `init` est l'initialisation d'un tableau avec des zéros. L'exemple `partition` copie les éléments positifs d'un tableau a dans un autre tableau b, et les éléments négatifs dans c. L'exemple `insertion` exemple insère un élément dans sa position correspondante dans un tableau trié. L'exemple `rotation` prend un tableau et applique une rotation par une position vers la gauche. Pour tous ces exemples, une postcondition est générée qui décrit l'effet attendu d'un programme.

5.4 Perspectives

Une implémentation de l'approche de vérification est en cours. Nos premières expériences sont encourageantes. Un des problèmes à résoudre dans le futur est l'explosion du nombre de compteurs nécessaires dans les automates produits. Cette explosion est due à la généralité de la traduction. Nous envisageons d'utiliser des analyses statiques pour réduire ce nombre en éliminant des compteurs inutiles.

Chapitre 6

Autres travaux

Pendant ces cinq dernières années j'ai réalisé plusieurs autres travaux qui ne sont pas détaillés dans ce mémoire. Quelques travaux sont résumés ici.

Dans [30] nous considérons le problème de vérification d'une classe de systèmes de processus qui rivalisent pour l'accès à des ressources communs. Nous supposons l'accès aux ressources est contrôlé selon une politique de FIFO avec la possibilité de distinguer entre des requêtes de haute ou basse priorité. Nous proposons un modèle pour ces systèmes basé sur des automates avec files d'attente. Pour ce modèle nous considérons le problème de vérification de propriétés exprimés en LTL/X enrichi avec quantification universelle de processus et interprété sur des comportements finis ainsi qu'équitables. Nous examinons aussi la vérification paramétrée d'interblocage de processus. En réduisant le problème de vérification paramétrée vers le model-checking de système fini, nous montrons plusieurs résultats de décidabilité pour plusieurs classes des propriétés et systèmes considérés. Nous montrons que la vérification de formules avec la quantification locale de processus est indécidable.

Dans [12] nous considérons des automates à multi-pile qui sont une extension d'automate à pile classique à plusieurs piles avec une restriction sur les opérations sur la pile : une opération dépiler peut uniquement être fait sur la première pile non-vide (les piles sont ordonnées). Nous montrons que le problème du vide pour ce modèle est 2ETIME-complet par rapport au nombre de piles. La borne supérieure est montrer en traduisant un automate à multi-pile vers une grammaire pour laquelle le vide de son langage peut être décidé facilement. La borne inférieure est obtenu en simulant une machine alternante de Turing qui travaille avec une espace exponentielle.

Chapitre 7

Conclusion

À la fin de chaque chapitre nous avons donnés les perspectives concernant le sujet précis traité à l'intérieur du chapitre. Ici, nous donnons quelques perspectives plus générales. Concernant le regular model-checking nous pouvons donner trois principaux axes de recherche future.

Il y a encore un grand potentiel d'amélioration des outils basés sur les automates de mots (ou d'arbre) finis. En effet, nous avons montré dans le chapitre 2 l'intérêt d'utiliser les automates non-déterministes (NFA) à la place des automates déterministes (DFA) dans le regular model-checking et/ou l'apprentissage. Nos expériences indiquent que mêmes des implémentations du regular model-checking utilisant les automates non-déterministes faites rapidement et sans trop chercher toutes les optimisations possibles peuvent rivaliser avec des implémentations basées sur les automates déterministes hautement optimisées qui sont basées sur l'outil MONA. MONA utilise des BDD pour représenter les DFA tandis que les implémentations basées sur les NFA utilisent des codages simples. Par ailleurs, les performances des implémentations basées sur les NFA sont bien meilleurs que celles des implémentations basées sur les DFA en utilisant le même type de représentation. Il est donc très probable qu'une implémentation des techniques utilisant les NFA basées sur les BDD permettra d'améliorer encore significativement les performances du regular model-checking. De la même façon l'utilisation de l'algorithme d'apprentissage pour les NFA que nous avons présenté combinée avec l'utilisation des représentations efficaces devrait améliorer considérablement les performances des algorithmes de vérification basés sur l'apprentissage.

Une deuxième approche à explorer est celle d'étendre les techniques de

regular model-checking vers les automates à compteurs, c.-à-d. utiliser des techniques similaires à celles utiliser dans le regular model-checking pour abstraire par exemple des automates à compteurs. Cela n'est pas simple car le vocabulaire des transitions (contraintes linéaires sur les compteurs) est potentiellement infini contrairement aux automates finis sur un alphabet fini comme utilisé jusqu'à présent. Pour ce type d'automates de nouvelles abstraction doivent être conçues. Il est aussi intéressant d'étudier l'extension des techniques du regular model-checking vers d'autres types d'automates comme les automates étudiés dans le chapitre 4. Une première tentative a été faite dans [56] pour une classe particulière d'automates à pile.

Une troisième approche à explorer est d'étudier des cas spécifiques dans le regular model-checking où uniquement des parties de la configuration sont codées comme des mots et le reste autrement. Cela permet d'utiliser des méthodes similaires aux méthodes dites de raisonnement local qui sont appliquées avec succès pour les programmes avec pointeurs dans le cadre de l'utilisation de la logique de séparation. Les abstractions que nous avons proposées dans le chapitre 3 vont déjà dans ce sens.

Une autre direction de travail prometteuse est d'étendre les méthodes d'apprentissage pour la vérification. En effet, ces dernières années ces méthodes se sont beaucoup dévéloppées. Elles sont basées en majorité sur l'utilisation des automates finis (voir aussi nos travaux dans le chapitre 2). Il serait intéressant d'étudier des modèles plus puissants permettant par exemple d'apprendre des invariants qui ne sont pas forcément réguliers. De manièrement plus générale, des méthodes déjà dévéloppés en apprentissage pourraient être exploitées dans un cadre de vérification en suivant les premiers succès obtenus ces dernières années.

Bibliographie

[1] P. Abdulla, L. Boasson, and A. Bouajjani. Effective Lossy Queue Languages. In *Proc. of ICALP'01*, volume 2076 of *LNCS*. Springer, 2001.

[2] P. Abdulla, A. Bouajjani, and B. Jonsson. On-the-fly Analysis of Systems with Unbounded, Lossy Fifo Channels. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[3] P. Abdulla, J. d'Orso, B. Jonsson, and M. Nilsson. Algorithmic improvements in regular model checking. In *Proc. of CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

[4] P. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular Tree Model Checking. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[5] P. Abdulla, A. Legay, J. d'Orso, and A.Rezine. Simulation-Based Iteration of Tree Transducers. In *Proc. of TACAS'05*, volume 3440 of *LNCS*. Springer, 2005.

[6] P. A. Abdulla, A. Bouajjani, L. Holík, L. Kaati, and T. Vojnar. Composed bisimulation for tree automata. In *CIAA*, volume 5148 of *Lecture Notes in Computer Science*, pages 212–222. Springer, 2008.

[7] R. Alur and P. Madhusudan. Visibly Pushdown Languages. In *Proceedings of STOC'04*. ACM Press, 2004.

[8] D. Angluin. Learning regular sets from queries and counterexamples. *Inf. Comput.*, 75(2) :87–106, 1987.

[9] A. Armando, S. Ranise, and M. Rusinowitch. Uniform Derivation of Decision Procedures by Superposition. In *Proc of CSL'01*, volume 2142 of *LNCS*, 2001.

[10] T. Arons, A. Pnueli, S. Ruah, J. Xu, and L.Zuck. Parameterized Verification with Automatically Computed Inductive Assertions. In *CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[11] Artmc - abstract regular tree model checking tool. available at http ://www.fit.vutbr.cz/research/groups/verifit/tools/artmc/.

[12] M. F. Atig, B. Bollig, and P. Habermehl. Emptiness of multi-pushdown automata is 2etime-complete. In *Developments in Language Theory*, volume 5257 of *Lecture Notes in Computer Science*, pages 121–133. Springer Verlag, 2008.

[13] P. Baldan, A. Corradini, J. Esparza, T. Heindel, B. König, and V. Kozioura. Verifying Red-Black Trees. In *Proc. of COSMICAH'05*, 2005.

[14] T. Ball and S. K. Rajamani. The SLAM toolkit. In *Proc. CAV'01*, LNCS. Springer, 2001.

[15] S. Bardin, A. Finkel, É. Lozes, and A. Sangnier. From pointer systems to counter systems using shape analysis. In R. Bharadwaj, editor, *Proceedings of the 5th International Workshop on Automated Verification of Infinite-State Systems (AVIS'06)*, Vienna, Austria, Apr. 2006.

[16] S. Bensalem, Y. Lakhnech, and S. Owre. Computing abstractions of infinite state systems compositionally and automatically. In *Proc. of CAV'98*, LNCS. Springer, 1998.

[17] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Invariant Synthesis for Combined Theories. In *In Proc. VMCAI'07*, volume 4349 of *LNCS*. Springer, 2007.

[18] D. Beyer, T. A. Henzinger, R. Majumdar, and A. Rybalchenko. Path Invariants. In *Proc. of PLDI'07, ACM SIGPLAN*, 2007.

[19] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In *Proc. CAV'03*, volume 2725 of *LNCS*. Springer, 2003.

[20] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-style learning of NFA. Research rep., LSV, ENS Cachan, 2008. www.lsv.ens-cachan.fr/Publis/index.php.

[21] B. Bollig, P. Habermehl, C. Kern, and M. Leucker. Angluin-Style Learning of NFA. In C. Boutilier, editor, *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 1004–1009, Pasadena, CA, USA, July 2009. AAAI Press.

[22] A. Bouajjani, M. Bozga, P. Habermehl, R. Iosif, P. Moro, and T. Vojnar. Programs with lists are counter automata. In *Proceedings of the 18th International Conference on Computer Aided Verification (CAV'06)*, volume 4144 of *Lecture Notes in Computer Science*, pages 517–531, Seattle, Washington, USA, 2006. Springer.

[23] A. Bouajjani, J. Esparza, and O. Maler. Reachability Analysis of Pushdown Automata : Application to Model-Checking. In *Proceedings of CONCUR '97*, volume 1243 of *LNCS*. Springer, 1997.

[24] A. Bouajjani, P. Habermehl, L. Holík, T. Touili, and T. Vojnar. Antichain-based universality and inclusion testing over nondeterministic finite tree automata. In *CIAA*, volume 5148 of *Lecture Notes in Computer Science*, pages 57–67. Springer, 2008.

[25] A. Bouajjani, P. Habermehl, Y. Jurski, and M. Sighireanu. Rewriting systems with data – A framework for reasoning about systems with unbounded structures over infinite data domains. In *Proceedings of the 16th International Symposium on Fundamentals of Computation Theory (FCT'07)*, volume 4639 of *Lecture Notes in Computer Science*, pages 1–22, Budapest, Hungary, 2007. Springer. invited paper.

[26] A. Bouajjani, P. Habermehl, P. Moro, and T. Vojnar. Verifying Programs with Dynamic 1-Selector-Linked Structures in Regular Model Checking. In *Proc. 11th Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2005), Edinburgh, Scotland, UK, Apr. 2005*, volume 3440 of *Lecture Notes in Computer Science*, pages 13–29. Springer Verlag, 2005.

[27] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking. In *Proceedings of the 7th International Workshop on Verification of Infinite State Systems (INFINITY'05)*, volume 149 of *Electronic Notes in Theoretical Computer Science*, pages 37–48, San Francisco, CA, USA, 2006. Elsevier Science Publishers.

[28] A. Bouajjani, P. Habermehl, A. Rogalewicz, and T. Vojnar. Abstract regular tree model checking of complex dynamic data structures. In *Proceedings of the 13th International Symposium Static Analysis (SAS'06)*, volume 4134 of *Lecture Notes in Computer Science*, pages 52–70, Seoul, Korea, 2006. Springer.

[29] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract Regular Model Checking. In *Proc. 16th Int. Conf. Computer Aided Verification (CAV 2004), Boston, MA, USA, July 2004*, volume 3114 of *Lecture Notes in Computer Science*, pages 372–386. Springer Verlag, 2004.

[30] A. Bouajjani, P. Habermehl, and T. Vojnar. Verification of parametric concurrent systems with prioritized fifo resource management. *Formal Methods in System Design*, 32(2) :129–172, 2008.

[31] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In *Proc. of CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[32] A. Bouajjani and T. Touili. Extrapolating Tree Transformations. In *Proc. of CAV'02*, volume 2404 of *LNCS*. Springer, 2002.

[33] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic verification of integer array programs. In *Proceedings of CAV 09*, volume 5643 of *Lecture Notes in Computer Science*, pages 157–172. Springer Verlag, 2009.

[34] M. Bozga, P. Habermehl, R. Iosif, F. Konečný, and T. Vojnar. Automatic Verification of Integer Array Programs. Technical Report TR-2009-2, Verimag, Grenoble, France, 2009.

[35] M. Bozga, R. Iosif, and Y. Lakhnech. Flat Parametric Counter Automata. In *Proc. of ICALP'06*, volume 4052 of *LNCS*. Springer, 2006.

[36] A. Bradley, Z. Manna, and H. Sipma. What 's Decidable About Arrays ? In *Proc. of VMCAI'06*, volume 3855 of *LNCS*. Springer, 2006.

[37] A. Bruggemann-Klein, M. Murata, and D. Wood. Regular Tree and Regular Hedge Languages over Unranked Alphabets : Version 1. Technical Report HKUST-TCSC-2001-0, The Hongkong University of Science and Technology, 2001.

[38] C. Calcagno, P. Gardner, and U. Zarfaty. Context Logic and Tree Update. In *Proceedings of POPL'05*. ACM Press, 2005.

[39] 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.

[40] S. Chaki, E. Clarke, A. Groce, J. Ouaknine, O. Strichman, and K. Yorav. Efficient verification of sequential and concurrent c programs. *Formal Methods in System Design*, 2004.

[41] E. M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In *Proc. CAV'00*, volume 1855 of *LNCS*. Springer, 2000.

[42] H. Comon and V. Cortier. Tree Automata with One Memory, Set Constraints and Cryptographic Protocols. *Theoretical Computer Science*, 331, 2005.

[43] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and appli-

cations. Available on : `http ://www.grappa.univ-lille3.fr/tata`, 2007. release October, 12th 2007.

[44] H. Comon and Y. Jurski. Multiple Counters Automata, Safety Analysis and Presburger Arithmetic. In *Proc. of CAV'98*, volume 1427 of *LNCS*. Springer, 1998.

[45] H. Comon-Lundh, F. Jacquemard, and N. Perrin. Visibly tree automata with memory and constraints. *Logical Methods in Computer Science*, 4(2), 2008.

[46] B. Cook, A. Podelski, and A. Rybalchenko. Abstraction Refinement for Termination. In *SAS*, volume 3672 of *LNCS*, 2005.

[47] T. Cormen, C. Leiserson, and R. Rivest. *Introduction to Algorithms*. The MIT Press, 1990.

[48] S. Dal Zilio and D. Lugiez. Multitrees Automata, Presburger's Constraints and Tree Logics. Technical Report 08-2002, LIF, 2002.

[49] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In *Proc. CAV'01*, volume 2102 of *LNCS*. Springer, 2001.

[50] S. Das and D. Dill. Counter-example based predicate discovery in predicate abstraction. In *Formal Methods in Computer-Aided Design*, 2002.

[51] F. Denis, A. Lemay, and A. Terlutte. Residual finite state automata. *Fundamenta Informaticae*, 51(4) :339–368, 2002.

[52] F. Denis, A. Lemay, and A. Terlutte. Learning regular languages using RFSAs. *Theoretical Comput. Sci.*, 313(2) :267–294, 2004.

[53] D. Distefano, P. O'Hearn, and H. Yang. A Local Shape Analysis Based on Separation Logic. In *Proc. of TACAS'06*, volume 3920 of *LNCS*. Springer, 2006.

[54] P. Dupont. Incremental Regular Inference. In *Grammatical Inference : Learning Syntax from Sentences*, volume 1147 of *LNAI*, 1996.

[55] J. Esparza. Grammars as Processes. In *Formal and Natural Computing*, volume 2300 of *LNCS*. Springer, 2002.

[56] D. Fisman and A. Pnueli. Beyond Regular Model Checking. In *Proc. of FSTTCS'01*, volume 2245 of *LNCS*. Springer, 2001.

[57] C. Flanagan and S. Qadeer. Predicate Abstraction for Software Verification. In *Proc. of POPL'02*. ACM, 2002.

[58] L. Fribourg and H. Olsen. Reachability Sets of Parametrized Rings as Regular Languages. In *INFINITY workshop*, volume 9 of *Electronic Notes in Theoretical Computer Science*. Elsevier Science, 1997.

[59] D. Geidmanis. Unsolvability of the Emptiness Problem for Alternating 1-way Multi-head and Multi-tape Finite Automata over Single-letter Alphabet. In *Computers and Artificial Intelligence*, volume 10, 1991.

[60] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Decision Procedures for Extensions of the Theory of Arrays. *Annals of Mathematics and Artificial Intelligence*, 50, 2007.

[61] S. Ghilardi, E. Nicolini, S. Ranise, and D. Zucchelli. Towards SMT Model Checking of Array-based Systems. In *Proc. of IJCAR'08*, volume 5195 of *LNCS*. Springer, 2008.

[62] D. Gopan, T. W. Reps, and S. Sagiv. A Framework for Numeric Analysis of Aarray Operations. In *POPL'05*. ACM, 2005.

[63] S. Gulwani, B. McCloskey, and A. Tiwari. Lifting Abstract Interpreters to Quantified Logical Domains. In *POPL'08*. ACM, 2008.

[64] P. Habermehl, R. Iosif, A. Rogalewicz, and T. Vojnar. Proving termination of tree manipulating programs. In *Proceedings of the 5th International Symposium on Automated Technology for Verification and Analysis (ATVA'07)*, volume 4762 of *Lecture Notes in Computer Science*, pages 145–161, Tokyo, Japan, 2007. Springer.

[65] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based Verification of Programs with Tree Updates. Technical Report TR-2005-16, Verimag, 2005. soumis à Acta Informatica.

[66] P. Habermehl, R. Iosif, and T. Vojnar. Automata-based verification of programs with tree updates. In *Proceedings of the 12th International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS'06)*, volume 3920 of *Lecture Notes in Computer Science*, pages 350–364, Vienna, Austria, 2006. Springer.

[67] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays ? Technical Report TR-2007-8, Verimag, 2007.

[68] P. Habermehl, R. Iosif, and T. Vojnar. A Logic of Singly Indexed Arrays. In *Proc. of LPAR'08*, volume 5330 of *LNAI*. Springer Verlag, 2008.

[69] P. Habermehl, R. Iosif, and T. Vojnar. What else is decidable about integer arrays ? In *Proceedings of the 11th International Conference on Foundations of Software Science and Computation Structures (FoSSaCS'08)*, volume 4962 of *Lecture Notes in Computer Science*, pages 474–489. Springer Verlag, 2008.

[70] P. Habermehl and T. Vojnar. Regular model checking using inference of regular languages. In J. Bradfield and F. Moller, editors, *Proceedings of the 6th International Workshop on Verification of Infinite State Systems (INFINITY'04)*, volume 138 of *Electronic Notes in Theoretical Computer Science*, pages 21–36, London, UK, Dec. 2005. Elsevier Science Publishers.

[71] N. Halbwachs and M. Péron. Discovering Properties about Arrays in Simple Programs. In *Proc. of PLDI'08*. ACM, 2008.

[72] T. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Lazy abstraction. In *Proc. of POPL'02*. ACM Press, 2002.

[73] T. A. Henzinger, R. Jhala, R. Majumdar, and G. Sutre. Software verification with Blast. In *Proc. of 10th SPIN Workshop*, volume 2648 of *LNCS*. Springer, 2003.

[74] J. Jaffar. Presburger Arithmetic with Array Segments. *Information Processing Letters*, 12, 1981.

[75] R. Jhala and K. L. McMillan. Array Abstractions from Proofs. In *CAV'07*, volume 4590 of *LNCS*. Springer, 2007.

[76] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. In *Proc. of TACAS'00*, volume 1785 of *LNCS*. Springer, 2000.

[77] C. Kern. *Learning Communicating and Nondeterministic Automata*. PhD-Thesis, RWTH Aachen, 2009.

[78] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. *Theoretical Computer Science*, 256(1–2), 2001.

[79] J. King. *A Program Verifier*. PhD thesis, Carnegie Mellon University, 1969.

[80] N. Klarlund and A. Møller. MONA Version 1.4 User Manual, 2001. BRICS, Department of Computer Science, University of Aarhus, Denmark.

[81] N. Klarlund and M. Schwartzbach. Graph Types. In *Proc. of POPL'93*. ACM Press, 1993.

[82] L. Kovacs and A. Voronkov. Finding Loop Invariants for Programs over Arrays Using a Theorem Prover. In *Proc. of FASE'09*, LNCS. Springer, 2009.

[83] S. K. Lahiri and R. E. Bryant. Indexed Predicate Discovery for Unbounded System Verification. In *CAV'04*, volume 3114 of *LNCS*. Springer, 2004.

[84] K. J. Lang. Random DFA's can be Approximately Learned from Sparse Uniform Examples. In *Proc. of the 5th ACM Workshop on Computational Learning Theory*. ACM Press, 1992.

[85] D. Lugiez. Multitree automata that count. *Theor. Comput. Sci.*, 333(1-2) :225–263, 2005.

[86] O. Maler and A. Pnueli. On the learnability of infinitary regular sets. *Inf. Comput.*, 118(2) :316–326, 1995.

[87] Z. Manna and A. Pnueli. *The Temporal Logic of Reactive and Concurrent Systems*. Springer-Verlag, 1992.

[88] Z. Manna, H. Sipma, and T. Zhang. Verifying Balanced Trees. In *Proceedings of the Symposium on Logical Foundations of Computer Science (LFCS 2007)*, volume 4514 of *Lecture Notes in Computer Science*. Springer, 2007.

[89] P. Mateti. A Decision Procedure for the Correctness of a Class of Programs. *Journal of the ACM*, 28(2), 1980.

[90] R. Mayr. Undecidable problems in unreliable computations. *Theor. Comput. Sci.*, 297(1-3) :337–354, 2003.

[91] J. McCarthy. Towards a Mathematical Science of Computation. In *IFIP Congress*, 1962.

[92] K. McMillan. Quantified Invariant Generation Using an Interpolating Saturation Prover. In *Proc. of TACAS'08*, volume 4963 of *LNCS*. Springer, 2008.

[93] M. Minsky. *Computation : Finite and Infinite Machines*. Prentice-Hall, Inc., 1967.

[94] A. Moeller and M. Schwartzbach. The Pointer Assertion Logic Engine. In *Proceeedings of PLDI'01*. ACM Press, 2001.

[95] A. Muscholl, T. Schwentick, H. Seidl, and P. Habermehl. Counting in trees for free. In *Proc. 31st Intern. Coll. on Automata, Languages and Programming (ICALP)*, volume 3142 of *Lecture Notes in Computer Science*. Springer Verlag, 2004.

[96] A. Nerode. Linear Automata Transformation. *American Math. Society*, 9 :541–544, 1958.

[97] H. Nguyen, C. David, S. Qin, and W. Chin. Automated Verification of Shape and Size Properties via Separation Logic. In *Proceedings of VMCAI'07*, volume 4349 of *Lecture Notes in Computer Science*. Springer, 2007.

[98] M. Nivat and D. Perrin. Ensembles reconnaissables de mots biinfinis. *Canad. J. Math.*, 38 :513–537, 1986.

[99] J. Oncina and P. Garcia. Inferring Regular Languages in Polynomial Update Time. In *Pattern Recognition and Image Analysis*, 1992.

[100] J. Pachl. Protocol Description and Analysis based on a State Transition Model with Channel Expressions. In *Protocol Verification, Specification, and Testing VII*, 1987.

[101] S. Parduhn. Algorithm Animation Using Shape Analysis with Special Regard to Binary Trees. Technical report, Universität des Saarlandes, 2005.

[102] H. Petersen. Alternation in Simple Devices. In *Proceedings of ICALP'95*, volume 944 of *LNCS*. Springer, 1995.

[103] A. Podelski, A. Rybalchenko, and T. Wies. Heap assumptions on demand. In *CAV*, volume 5123 of *Lecture Notes in Computer Science*, pages 314–327. Springer, 2008.

[104] M. Presburger. Über die Vollständigkeit eines gewissen Systems der Arithmetik ganzer Zahlen, in welchem die Addition als einzige Operation hervortritt. In *Comptes Rendus du Premier Congrès des Mathématiciens des Pays Slaves*, pages 92–101, Warsaw, Poland, 1929.

[105] M. Rabin. Decidability of Second Order Theories and Automata on Infinite Trees. *Transactions of American Mathematical Society*, 141, 1969.

[106] J. Reynolds. Separation Logic : A Logic for Shared Mutable Data Structures. In *Proc. of LICS'02*. IEEE CS Press, 2002.

[107] R. Rugina. Quantitative Shape Analysis. In *Proceedings of SAS'04*, volume 3148 of *LNCS*. Springer, 2004.

[108] S. Sagiv, T. Reps, and R. Wilhelm. Parametric shape analysis via 3-valued logic. *TOPLAS*, 24(3), 2002.

[109] H. Saïdi. Model checking guided abstraction and analysis. In *Proc. of SAS'00*, volume 1824 of *LNCS*. Springer, 2000.

[110] H. Saïdi and S. Graf. Construction of abstract state graphs with PVS. In *Proc. of CAV'97*, volume 1254 of *LNCS*. Springer, 1997.

[111] E. Shahar and A. Pnueli. Acceleration in Verification of Parameterized Tree Networks. Technical Report MCS02-12, Faculty of Mathematics and Computer Science, The Weizmann Institute of Science, Rehovot, Israel, 2002.

[112] A. Stump, C. Barrett, D. Dill, and J. Levitt. A Decision Procedure for an Extensional Theory of Arrays. In *Proc. of LICS'01*, 2001.

[113] N. Suzuki and D. Jefferson. Verification Decidability of Presburger Array Programs. *Journal of the ACM*, 27(1), 1980.

[114] W. Thomas. Automata on Infinite Objects. In *Handbook of Theoretical Computer Science, volume B : Formal Models and Semantics*. Elsevier, 1990.

[115] Timbuk tool. available at http ://www.irisa.fr/lande/genet/timbuk/.

[116] T. Touili. Widening techniques for regular model checking. *ENTCS*, 50, 2001.

[117] B. A. Trakhtenbrot and Y. A. Barzdin. *Finite Automata : Behavior and Synthesis*. North-Holland, 1973.

[118] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Learning to verify safety properties. In *ICFEM*, volume 3308 of *Lecture Notes in Computer Science*, pages 274–289. Springer, 2004.

[119] A. Vardhan, K. Sen, M. Viswanathan, and G. Agha. Using language inference to verify omega-regular properties. In *TACAS*, volume 3440 of *Lecture Notes in Computer Science*, pages 45–60. Springer, 2005.

[120] A. Vardhan and M. Viswanathan. Lever : A tool for learning based verification. In *CAV*, volume 4144 of *Lecture Notes in Computer Science*, pages 471–474. Springer, 2006.

[121] A. Vardhan and M. Viswanathan. Learning to verify branching time properties. *Formal Methods in System Design*, 31(1) :35–61, 2007.

[122] M. D. Wulf, L. Doyen, T. A. Henzinger, and J.-F. Raskin. Antichains : A new algorithm for checking universality of finite automata. In *CAV'06*, 2006.