Contributions à la vérification et à la validation efficaces fondées sur des modèles

Alois Dreyfus

▶ To cite this version:

Alois Dreyfus. Contributions à la vérification et à la validation efficaces fondées sur des modèles. Systèmes et contrôle [cs.SY]. Université de Franche-Comté, 2014. Français. NNT : 2014BESA2076 . tel-01090759v2

HAL Id: tel-01090759
https://inria.hal.science/tel-01090759v2

Submitted on 19 Oct 2017

HAL is a multi-disciplinary open access archive for the deposit and dissemination of scientific research documents, whether they are published or not. The documents may come from teaching and research institutions in France or abroad, or from public or private research centers.

L'archive ouverte pluridisciplinaire **HAL**, est destinée au dépôt et à la diffusion de documents scientifiques de niveau recherche, publiés ou non, émanant des établissements d'enseignement et de recherche français ou étrangers, des laboratoires publics ou privés.

Thèse de Doctorat

école doctorale sciences pour l'ingénieur et microtechniques
UNIVERSITÉ DE FRANCHE-COMTÉ

Contributions à la vérification et à la validation efficaces fondées sur des modèles

Aloïs DREYFUS

SPIM

Thèse de Doctorat

école doctorale **sciences pour l'ingénieur et microtechniques**
UNIVERSITÉ DE FRANCHE-COMTÉ

THÈSE présentée par

Aloïs DREYFUS

pour obtenir le

Grade de Docteur de
l'Université de Franche-Comté

Spécialité : **Informatique**

Contributions à la vérification et à la validation efficaces fondées sur des modèles

Unité de Recherche :
FEMTO-ST DISC

Soutenue le 22/10/2014 devant le Jury :

Jacques JULLIAND	Président du jury	Professeur à l'Université de Franche-Comté
Pierre-Cyrille HÉAM	Directeur de thèse	Professeur à l'Université de Franche-Comté
Olga KOUCHNARENKO	Directrice de thèse	Professeur à l'Université de Franche-Comté
Ioannis PARISSIS	Rapporteur	Professeur à l'INP de Grenoble
Pierre RÉTY	Rapporteur	Maître de Conférences HDR à l'Université d'Orléans
Pascal POIZAT	Examinateur	Professeur à l'Univ. Paris Ouest Nanterre La Défense

Remerciements

J'aimerais tout d'abord remercier mes encadrants et mentors, Olga et Pierre-Cyrille, qui m'ont fait confiance, soutenu et conseillé pendant 6 ans (2 ans en tant qu'ingénieur, puis 4 ans pour mon doctorat). Lorsque mon moral et ma motivation étaient en berne, au point que l'idée folle de jeter l'éponge a parfois effleuré mon esprit fatigué, vous avez su trouver quoi dire et/ou quoi faire pour m'encourager et m'aider à rester dans la course. Je remercie vivement les membres du jury, qui ont accepté d'évaluer mon travail de thèse et plus particulièrement les rapporteurs, Ioannis Parissis et Pierre Réty, qui ont en plus accepté la tâche conséquente de lire, comprendre et critiquer ce mémoire.

Je tiens ensuite à remercier mes parents Pierrette et Laurent, ainsi que ma grande soeur Fanny et mon petit frère Chouch, pour leur sempiternel et inconditionnel soutien. J'ai de la chance de vous avoir, et je ne vous dirai jamais assez à quel point je vous aime ♡♡♡♡. Je remercie de même Nico et le loulou, pour tout le bonheur que vous apportez à notre famille depuis que vous en faites partie.

Je remercie également celles et ceux qui m'ont aidé à tenir jusqu'au bout, parfois peut-être sans même s'en rendre compte. Je pense particulièrement à Cyber, Tcharlz, Nesnours, @ngel, Fouad, Oscar, Aydeé, Richard, Justine, Lysiane et Anne-Sorya. Et je ne voudrais surtout pas oublier de remercier ma soupape de décompression dijonnaise : Cyber, Anne-Sophie, Chloé, ainsi que Féfé et Jean-Marc, pour votre hospitalité, nos délires et nos discussions, qui m'ont fait un bien fou quand j'avais le moral dans les chaussettes !

Puis j'aimerais remercier mes compagnons de galère : Jibouille, Éric, Abdallah, Hamida, Hadrien, Doudou, P.C., Kalou, Jérôme, Oscar, Séb, Mathias, Roméo, Kerkouette, Sékou, Alexandru, Julien, Lionel, Ivan, Youssou, Bety, Nicolas, Jean-Marie, Fouad, Alexander, Vincent H., Rami, John, Guillaume, Lemia, David, Valérie, Naima, Paco, Sarga, Alban, Aydeé, Yanbo, Régis, Lamiel, Elena et Alexandre, pour l'ambiance agréable et conviviale qui rend les levers du matin plus faciles (ou moins difficiles, c'est selon). Je souhaite bon courage et bon vent à ceux qui rament en ce moment même.

Il reste encore tellement de monde (amis, cousins, collègues, ...) que j'apprécie vraiment et qui méritent que je les remercie, qu'il est clair que je ne pourrai pas être exhaustif. Donc j'espère que celles et ceux que je n'ai pas cités ne m'en tiendront pas rigueur.

Sommaire

I Introduction **15**

1 Prologue **17**

1.1 Contexte 17
1.2 Contributions 18
1.3 Plan 18
1.4 Publications 18

2 Préliminaires théoriques et notations **21**

2.1 Automates et langages réguliers 21
2.1.1 Mots et langages 21
2.1.2 Automates finis et expressions régulières 21
2.1.3 Produit d'automates 23
2.1.4 Isomorphisme d'automates 23
2.1.5 Réduction d'un automate 24
2.1.6 Transducteurs lettre à lettre finis 26
2.1.7 Appliquer un transducteur à un automate 27
2.1.8 Appliquer un transducteur à un langage 28
2.2 Grammaires algébriques et automates à pile 28
2.2.1 Grammaires algébriques 28
2.2.2 Automates à pile 29
2.2.3 Génération aléatoire uniforme d'un arbre de dérivation dans une grammaire 31
2.2.4 D'un NPDA vers une grammaire algébrique 33

II Approximations régulières pour l'analyse d'accessibilité **35**

3 Contexte et problématiques **39**

3.1 Introduction et énoncé du problème 39
3.2 Technique proposée 42
3.3 Calcul de la sur-approximation 43

3.4 Raffinement d'approximations . . . 45

4 Contribution : Critères de fusion syntaxiques 49

4.1 In et Out . . . 49

4.2 Left et Right . . . 50

4.3 Combinaisons de fonctions d'approximation . . . 53

4.4 Expérimentations . . . 53

4.5 Conclusion . . . 54

5 Contribution : Fonction d'approximation qui utilise les états du transducteur 55

5.1 Définitions supplémentaires . . . 55

5.2 Critère de fusion qui utilise les états du transducteur . . . 58

5.3 Raffiner les approximations qui utilisent les états du transducteur . . . 61

5.4 Expérimentations . . . 64

5.5 Conclusion . . . 66

6 Conclusion de la partie II 67

III Combiner génération aléatoire et critère de couverture, à partir d'une grammaire algébrique ou d'un automate à pile 69

7 Contexte et problématiques 73

7.1 Le test logiciel . . . 73

7.1.1 Test automatique . . . 73

7.1.2 Test à partir de modèles (MBT) . . . 74

7.1.3 Critères de couverture . . . 74

7.1.4 Test aléatoire . . . 75

7.1.5 Génération aléatoire-uniforme d'un chemin dans un graphe fini . . . 75

7.1.6 Test à partir de modèles à pile . . . 76

7.2 Où se situe-t-on ? . . . 77

7.2.1 Génération aléatoire de chemins contre parcours aléatoire . . . 77

7.2.2 Combiner génération aléatoire et critères de couverture . . . 78

7.2.3 Modèles réguliers (sans pile) contre modèles algébriques (à pile) . . . 80

7.2.4 Contribution : Combiner génération aléatoire et critères de couverture, dans des modèles à pile . . . 82

8 Contribution : Application sur les grammaires 83

8.1 Calculer $p_{X,n}$ et $p_{X,Y,n}$ . . . 84
8.2 Calculer $|E_{X,n}(G)|$ et $|E_{X,Y,n}(G)|$ . . . 84
8.3 Expérimentations . . . 87

9 Contribution : Application sur les automates à piles **91**
9.1 Génération aléatoire-uniforme de DFA-Traces cohérentes . . . 93
9.1.1 Génération aléatoire de DFA-traces cohérentes . . . 93
9.2 Critères de couverture d'un PDA . . . 94
9.2.1 Le critère *Tous les états* . . . 94
9.2.2 Le critère *Toutes les transitions* . . . 95
9.2.3 Combiner test aléatoire et critères de couverture . . . 97
9.3 Expérimentations . . . 97
9.3.1 Information technique . . . 98
9.3.2 Exemple illustratif . . . 98
9.3.3 Exemple illustratif pour les critères de couverture . . . 101
9.3.4 Temps d'exécution sur plus d'exemples . . . 103
9.4 Cas d'étude : l'algorithme Shunting-Yard . . . 106
9.4.1 Description . . . 106
9.4.2 Automate sous-jacent pour l'algorithme Shunting-Yard . . . 108
9.4.3 Expérimentations . . . 109
9.5 Discussion . . . 109

10 Conclusion de la partie III **115**

IV Conclusions et perspectives **117**

11 Conclusions et perspectives **119**

Table des figures

2.1 $\mathcal{A}_{ex}$ (Exemple d'automate) 22

2.2 $\mathcal{A}_{ex2}$ (Deuxième exemple d'automate) 22

2.3 $\mathcal{A}_1 \times \mathcal{A}_2$ 23

2.4 $\mathcal{A}_{ex} \times \mathcal{A}_{ex2}$ 24

2.5 Automates isomorphes 24

2.6 $\mathcal{A}_{ex/\sim}$ 25

2.7 $\mathcal{T}(\mathcal{A}_1)/_{\sim_{\text{exe}}}$ 25

2.8 $\mathfrak{In}(\mathcal{A}_{ex})$ 26

2.9 $\mathcal{T}_{ex}$ (Exemple de transducteur) 26

2.10 $\mathcal{T}_{ex}(\mathcal{A}_{ex})$ 27

2.11 $\mathcal{T}(\mathcal{A}_1)$ 28

2.12 Token ring 28

2.13 Arbre de dérivation correspondant à la séquence $S, aSb, aTbb, abb$ 29

2.14 $\mathcal{A}_{\text{power}}$, exemple de NPDA 30

2.15 Algorithme Génération aléatoire 32

2.16 Exemple d'une grammaire qui génère des DFA-traces cohérentes 33

3.1 Approche par sur-approximation 42

3.2 Sur-approximations successives 43

3.3 Calcul de $C_{\mathfrak{Right}}(\mathcal{A})$ 44

3.4 Semi-algorithme `PointFixe` 44

3.5 Token ring : semi-algorithme `PointFixe` avec $\mathfrak{F} = \mathfrak{Left}$ 45

3.6 Situation nécessitant un raffinement de l'approximation 46

3.7 Algorithme `Analyse en arrière` 46

3.8 Principe du raffinement 47

4.1 $\mathfrak{In}(\mathcal{A}_{ex})$ 50

4.2 $\mathfrak{Out}(\mathcal{A}_{ex})$ 50

4.3 $\mathcal{A}_{ex}^{Left,2}$ 51

4.4 $\mathcal{A}_{ex}^{Right,3}$ 51

4.5 $\mathfrak{Left}(\mathcal{A}_{\mathrm{tr1}})$ 52
4.6 $\mathfrak{Right}(\mathcal{A}_{ex})$ 52
4.7 Résultats avec des critères syntaxiques 54

5.1 $x \cdot \mathcal{A}_{ex}$ 56
5.2 $\mathcal{A}_{ex3}$ 56
5.3 $\mathcal{A}_{ex4}$ 56
5.4 Token ring : Appliquer un transducteur $\mathcal{T}$ à un $Q_C \times Q_{\mathcal{T}}^*$-automate (1) 57
5.5 Token ring : Appliquer un transducteur $\mathcal{T}$ à un $Q_C \times Q_{\mathcal{T}}^*$-automate (2) 58
5.6 Token Ring : fusion en fonction des états du transducteur (1) 59
5.7 Token ring : fusion en fonction des états du transducteur (2) 59
5.8 Semi-algorithme `PointFixeT` 61
5.9 Situation nécessitant un raffinement de l'approximation 62
5.10 Algorithme `Séparer` 63
5.11 Algorithme `Raffiner` 63
5.12 Exemples pour les algorithmes `Séparer` et `Raffiner` 63
5.13 Semi-algorithme `Vérif-avec-raffinement` 64
5.14 Types d'automates C 65
5.15 Résultats avec la méthode reposant sur les états du transducteur comme critère de fusion 66

7.1 Classification des différents types de tests 74
7.2 Exemple de graphe fini 75
7.3 Exemple 1 77
7.4 Exemple 2 78
7.5 Génération aléatoire - Algorithme 1 79
7.6 Génération aléatoire - Algorithme 2 79
7.7 Génération aléatoire - Algorithme 3 80
7.8 Calcul des probabilités π_e, afin d'optimiser la probabilité de couvrir C 80
7.9 Programme en C, qui calcule x^n 81
7.10 Graphe de flot de contrôle de x^n 81

8.1 Arbre de dérivation de la grammaire G de l'exemple 8.3 86
8.2 Arbre de dérivation de la grammaire G_X de l'exemple 8.3 86

9.1 Méthode de test aléatoire d'un programme 92
9.2 Génération aléatoire de DFA-traces cohérentes 93

9.3 Automate sous-jacent de $\mathcal{A}^{6}_{exe}$ 95
9.4 Automate sous-jacent de $\mathcal{A}^{(6,h,9)}_{exe}$ 96
9.5 Calcul des probabilités π_q, afin d'optimiser la probabilité de couvrir Q 97
9.6 Fonctions mutuellement récursives 98
9.7 $\mathcal{A}_{\text{modulo}}$: NPDA de l'exemple 99
9.8 Automate $\mathcal{A}_{\text{xpath}}$ 101
9.9 Algorithme Shunting-Yard 107
9.10 Automate sous-jacent pour l'algorithme Shunting-Yard 108
9.11 Implémentation en C de l'algorithme Shunting-Yard, que nous avons testée 1/2 . 110
9.12 Implémentation en C de l'algorithme Shunting-Yard, que nous avons testée 2/2 . 111

I

Introduction

1

Prologue

1.1/ Contexte

Le développement de programmes sûrs, sécurisés et exempts de bogues est un des problèmes les plus difficiles de l'informatique moderne. Les logiciels sont omniprésents dans notre quotidien, et on leur confie des tâches de plus en plus sensibles, comme le transfert de nos données bancaires lorsque l'on commande sur internet. On parle de *logiciels critiques* lorsque les conséquences d'un dysfonctionnement peuvent être particulièrement graves, comme lorsque des vies ou lorsque d'importantes sommes d'argent (millions/milliards d'euros) sont en jeux. Sont concernés par exemple les logiciels utilisés pour les systèmes de transport (avions, trains, voitures, ...), l'énergie (réseau de distribution, centrales, ...), la santé (gestion des hôpitaux, appareils médicaux, ...), la finance (bourse, paiement en ligne ou par carte, ...), ou encore dans le domaine militaire (missiles, drones, communications cryptées, ...) ou dans l'aérospatiale (fusées, satellites, ...). Il est donc indispensable de pouvoir s'assurer de la qualité de ces logiciels. Dans ce contexte, on peut distinguer deux techniques complémentaires pour évaluer la qualité d'un système informatique : la *vérification* et le *test*.

La vérification consiste à prouver mathématiquement qu'un code ou un modèle d'application répond aux attentes (ou propriétés) décrites dans sa spécification. Elle peut se faire avec un assistant de preuve (par exemple Coq) qui aide l'ingénieur à construire des preuves formelles, pouvant même démontrer certains lemmes de façon automatique. La vérification peut aussi se faire par *évaluation de modèle* – on utilise couramment le terme anglais *model-checking* –, qui est une méthode qui tente de vérifier si d'après le modèle du système, l'ensemble de ses états atteignables satisfait (ou pas) des propriétés. Un des principaux inconvénients de la vérification est la complexité algorithmique des techniques existantes, qui les rend difficilement applicables à des systèmes de taille importante.

Contrairement à la vérification, le test ne fournit pas de preuve formelle en amont. Mais les techniques de test sont applicables à des systèmes de grande taille, et en pratique elles sont pertinentes dans le développement de logiciels de qualité. Le test occupe une part de plus en plus importante du développement de logiciels, au point que se développent des méthodes de *développement piloté par les tests* – abrégé en *TDD*, pour l'anglais Test Driven Development –, qui préconisent d'écrire les tests avant d'écrire le code source. Au cours de la dernière décennie, afin de faire face à cette importance croissante de la phase de test, de nombreux travaux ont été réalisés dans l'objectif de passer des méthodes de test manuelles (fondées sur l'expérience) à des méthodes automatiques ou semi-automatiques fondées sur des méthodes formelles.

1.2/ CONTRIBUTIONS

Cette thèse présente des contributions à la vérification et à la validation à base de modèles.

Notre contribution dans le cadre de la vérification, consiste à définir deux nouvelles techniques d'approximation pour le model-checking régulier, dans le but de fournir des (semi-)algorithmes efficaces. On calcule des sur-approximations de l'ensemble des états accessibles, avec l'objectif d'assurer la terminaison de l'exploration de l'espace d'états. L'ensemble des états accessibles (ou les sur-approximations de cet ensemble) est représenté par un langage régulier, lui même codé par un automate fini. La première technique consiste à sur-approximer l'ensemble des états atteignables en fusionnant des états des automates, en fonction de critères syntaxiques simples, ou d'une combinaison de ces critères (*cf.* section 4). La seconde technique d'approximation consiste aussi à fusionner des états des automates, mais cette fois-ci à l'aide d'un transducteur (*cf.* section 5.2). De plus, pour cette seconde technique, nous développons une nouvelle approche pour raffiner les approximations (*cf.* section 5.3), qui s'inspire du paradigme CEGAR (*cf.* section 3.4). Ces deux techniques d'approximation pour le model-checking régulier ont été publiées dans [DHK13a].

Pour le test, notre contribution est la définition d'une technique qui permet de combiner la génération aléatoire avec un critère de couverture, à partir d'une grammaire algébrique ou d'un automate à pile. Dans [DGG+12], il est expliqué comment biaiser une approche de test aléatoire uniforme dans un graphe fini, en utilisant des contraintes données par un critère de couverture, afin d'optimiser la probabilité de satisfaire ce critère. Mais un graphe fini représente souvent une abstraction forte du système sous test, il y a donc un risque que de nombreux tests abstraits ne soient pas concrétisables (c'est-à-dire jouables sur l'implémentation). Nous avons enrichi cette approche afin de l'appliquer sur des modèles algébriques (des grammaires algébriques dans le chapitre 8, et des automates à pile dans le chapitre 9), ce qui permet de réduire le degré d'abstraction du modèle et donc de générer moins de tests non concrétisables. Notre technique qui permet de combiner la génération aléatoire avec un critère de couverture à partir d'une grammaire algébrique a été publiée dans [DHK13b], et celle sur les automates à pile a été publiée dans [DHKM14].

1.3/ PLAN

Dans le chapitre 2, nous introduisons toutes les définitions, notations, propositions et théorèmes indispensables à la lecture de cette thèse. Nous présentons tout d'abord les notions relatives aux automates et langages réguliers, puis les notions relatives aux grammaires algébriques et aux automates à pile.

La partie II présente nos contributions dans le domaine de la vérification : Le chapitre 3 introduit le model-checking régulier et le raffinement d'approximations. Dans le chapitre 4, nous présentons une technique pour le model-checking régulier, qui consiste à sur-approximer l'ensemble des états atteignables en fusionnant des états des automates, en fonction de critères syntaxiques simples, ou d'une combinaison de ces critères. Le chapitre 5 présente une technique utilisant un nouveau critère de fusion, qui exploite la relation de transition du système.

La partie III présente nos contributions dans le domaine du test : Le chapitre 7 présente le contexte de nos travaux. Nous combinons ensuite de la génération aléatoire avec un critère de couverture, dans une grammaire (chapitre 8), puis dans un automate à pile (chapitre 9).

1.4/ Publications

Cette thèse s'appuie sur des travaux ayant fait l'objet de publications. Voici une liste de celles auxquelles j'ai participé :

[DHK13a] contient deux méthodes de model-checking régulier : La première méthode utilise des critères syntaxiques que l'on peut combiner avec des opérateurs logiques (*cf.* chapitre 4), et la seconde exploite la relation de transition du système (*cf.* chapitre 5).

[DHK13b] expose notre méthode pour combiner de la génération aléatoire avec un critère de couverture, dans le cadre du test à partir d'une grammaire (*cf.* chapitre 8).

[DHKM14] détaille comment combiner de la génération aléatoire avec un critère de couverture, à partir d'un automate à pile (*cf.* chapitre 9).

2

Préliminaires théoriques et notations

Cette section présente les notations et définitions nécessaires à la compréhension du document.

2.1/ Automates et langages réguliers

2.1.1/ Mots et langages

Un *alphabet* est un ensemble fini d'éléments appelés *lettres*. Un mot u sur un alphabet Σ est une juxtaposition de lettres de Σ. Le nombre de lettres de u (comptées avec multiplicité) est appelé *longueur* de u. L'ensemble des mots sur un alphabet Σ est noté Σ^*. On appelle *langage* sur un alphabet Σ tout sous-ensemble de Σ^*. Le mot vide (le mot de longueur nulle) est noté ε. L'ensemble $\Sigma^* \setminus \{\varepsilon\}$ est noté Σ^+.

2.1.2/ Automates finis et expressions régulières

Un *automate* fini $\mathcal{A}$ sur un alphabet Σ est un 5-uplet (Q, Σ, E, I, F) où :

- Q est l'ensemble fini des *états*,
- $E \subseteq Q \times \Sigma \times Q$ est l'ensemble fini des *transitions*,
- $I \subseteq Q$ est l'ensemble fini des *états initiaux*,
- $F \subseteq Q$ est l'ensemble fini des *états finaux*.

Dans le cadre de nos recherches nous avons uniquement travaillé sur des automates finis, l'utilisation par la suite du terme "automate" désignera donc un automate fini. Le terme "automate fini" est abrégé en *NFA*, pour l'anglais Nondeterministic Finite Automaton.

On définit la *taille* de $\mathcal{A}$ par $|\mathcal{A}| = |Q| + |E|$. Un automate est *déterministe* si I est un singleton et pour tout $(p, a) \in Q \times \Sigma$ il existe au plus un $q \in Q$ tel que $(p, a, q) \in E$. Le terme "automate déterministe" est abrégé en *DFA*, pour l'anglais Deterministic Finite Automaton. Un automate est *complet* si pour tout $(p, a) \in Q \times \Sigma$ il existe au moins un $q \in Q$ tel que $(p, a, q) \in E$.

Pour une transition de la forme (p, a, q), p est l'*état de départ*, a est l'*étiquette*, et q est l'*état d'arrivée*. Deux transitions (p_1, a_1, q_1) et (p_2, a_2, q_2) sont dites *consécutives* si $q_1 = p_2$. Un *chemin* de $\mathcal{A}$ est une suite finie (éventuellement vide) de transitions $(p_1, a_1, q_1) \dots (p_n, a_n, q_n)$ consécutives. Le nombre entier n est la *longueur* du chemin, et le mot $a_1 \dots a_n$ est son *étiquette*. On dit qu'un chemin est *réussi* lorsque p_1 est un état initial, et q_n un état final. Un mot u est *reconnu* par $\mathcal{A}$ si u est l'étiquette d'un chemin réussi de $\mathcal{A}$. On appelle *langage reconnu* par $\mathcal{A}$ l'ensemble des mots reconnus par l'automate $\mathcal{A}$, et on le note $L(\mathcal{A})$.

Si $\mathcal{A}$ est déterministe et complet, pour tout état p et tout mot u, il existe un unique état de $\mathcal{A}$, noté $p \cdot_{\mathcal{A}} u$ accessible en lisant à partir de l'état p, un chemin étiqueté par u. S'il n'y a pas d'ambiguïté sur l'automate $\mathcal{A}$, on écrit plus simplement $p \cdot u$. Par convention, soit ε le mot vide, $p \cdot \varepsilon = p$. Un état p est *accessible* s'il existe un chemin d'un état initial vers p, et *co-accessible* s'il existe un

chemin de p vers un état final.

Un automate dont les états sont tous accessibles et co-accessibles est dit *élagué*. Si $\mathcal{A}$ n'est pas élagué, supprimer tous ses états qui ne sont pas à la fois accessibles et co-accessibles, ainsi que toutes les transitions liées, produit un automate élagué qui reconnaît le même langage. Dans les figures, les automates sont généralement donnés sous forme élaguée afin d'en faciliter la compréhension.

Une *expression régulière* sur un alphabet Σ est une expression *reg* définie par la grammaire suivante :

$$reg := \ B \mid (reg \cup reg) \mid (reg \cdot reg) \mid reg^*,$$

où B est un ensemble fini de mots. Par exemple $(\{ab, b\}^* \cup (\{aa\} \cdot \{a\}^*))$ est une expression régulière sur $\{a, b\}$. Pour simplifier, si u est un mot et *reg* une expression régulière, les expressions du type $\{u\}^*$, $\{u\} \cdot L$ et $L \cdot \{u\}$ sont respectivement écrites u^*, $u \cdot L$ (ou même uL) et $L \cdot \{u\}$ (et même Lu). De plus, on omet parfois les parenthèses, l'étoile étant prioritaire sur le produit, lui-même prioritaire sur l'union. L'expression $(\{ab, b\}^* \cup (\{aa\} \cdot \{a\}^*))$ sera simplement notée $\{ab, b\}^* \cup aaa^*$. On confondra dans la suite une expression régulière avec le langage qu'elle reconnaît, la sémantique étant naturellement celle de l'union, produit et de l'étoile sur les langages.

Exemple 2.1. *La figure 2.1 et la figure 2.2 représentent $\mathcal{A}_{ex}$ et $\mathcal{A}_{ex2}$, qui sont deux exemples d'automates finis. L'automate $\mathcal{A}_{ex}$ (figure 2.1) reconnaît les mots de la forme a^*ba, ou a^*bb, ou a^*bba, ou a^*bbc. L'automate $\mathcal{A}_{ex2} = (Q_{ex2}, \Sigma_{ex2}, E_{ex2}, I_{ex2}, F_{ex2})$ (figure 2.2) reconnaît les mots de la forme b^*c, et on a $Q_{ex2} = \{1, 2\}$, $\Sigma_{ex2} = \{b, c\}$, $E_{ex2} = \{(1, b, 1), (1, c, 2)\}$, $I_{ex2} = \{1\}$ et $F_{ex2} = \{2\}$.*

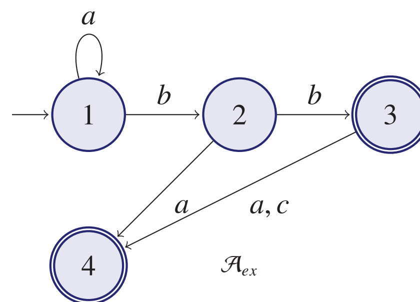


FIGURE 2.1 – $\mathcal{A}_{ex}$ (Exemple d'automate)

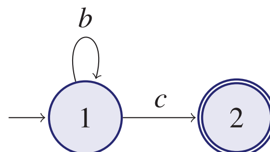


FIGURE 2.2 – $\mathcal{A}_{ex2}$ (Deuxième exemple d'automate)

Définition 2.1. Langage régulier

Un langage est dit *régulier* s'il appartient à la plus petite famille de langages contenant les langages finis et close par union, produit et étoile.

D'après le théorème de Kleene, un langage est régulier si et seulement s'il est reconnu par un automate fini. De plus, un langage est régulier si et seulement s'il est exprimable par une expression régulière. Il faut aussi noter que l'on peut montrer que l'ensemble des langages réguliers sur un alphabet donné est clos par intersection et complément.

Définition 2.2. Problème régulier

Un problème est dit *régulier* s'il est représentable avec uniquement des langages réguliers.

2.1.3/ Produit d'automates

Le *produit* de deux automates, $\mathcal{A}_1 = (Q_1, \Sigma_1, E_1, I_1, F_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma_2, E_2, I_2, F_2)$, noté $\mathcal{A}_1 \times \mathcal{A}_2$, est l'automate (Q, Σ, E, I, F) où :

— $Q = Q_1 \times Q_2$,
— $\Sigma = \Sigma_1 \bigcap \Sigma_2$,
— $E = \{((p_1, p_2), a, (q_1, q_2)) \mid \exists a \in \Sigma_1 \bigcap \Sigma_2, (p_1, a, q_1) \in E_1, (p_2, a, q_2) \in E_2\}$,
— $I = I_1 \times I_2$,
— $F = F_1 \times F_2$.

Intuitivement, l'ensemble des chemins de $\mathcal{A}_1 \times \mathcal{A}_2$ est composé des chemins qui existent à la fois dans $\mathcal{A}_1$ et dans $\mathcal{A}_2$ (au nom des états près). Il reconnaît les mots qui font à la fois partie du langage de $\mathcal{A}_1$ et du langage de $\mathcal{A}_2$. $L(\mathcal{A}_1 \times \mathcal{A}_2) = L(\mathcal{A}_1) \bigcap L(\mathcal{A}_2)$.

Exemple 2.2. *La figure 2.3 représente deux automates $\mathcal{A}_1$ et $\mathcal{A}_2$ ainsi que leur produit, notée $\mathcal{A}_1 \times \mathcal{A}_2$.*

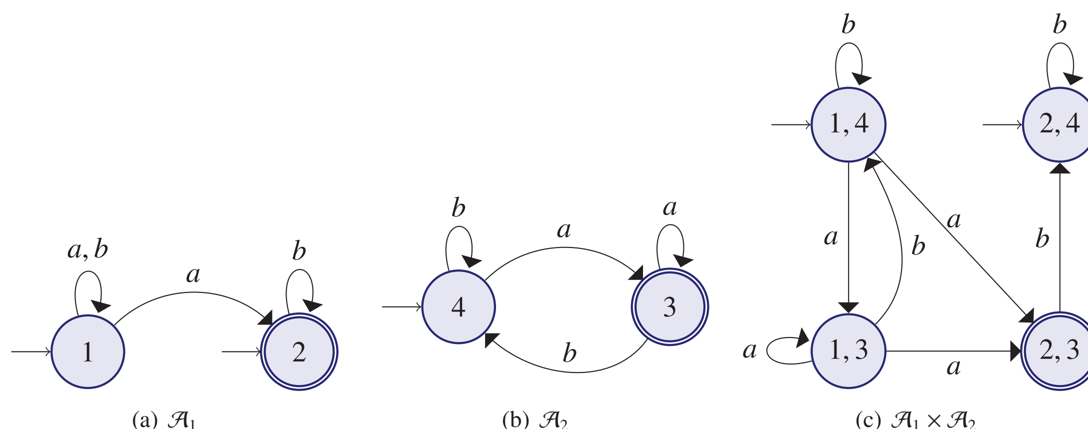


FIGURE 2.3 – $\mathcal{A}_1 \times \mathcal{A}_2$

Exemple 2.3. *L'automate $\mathcal{A}_{ex}$ reconnaît les mots de la forme a^*ba, ou a^*bb, ou a^*bba, ou a^*bbc, et $\mathcal{A}_{ex2}$ reconnaît les mots de la forme b^*c. Le produit $\mathcal{A}_{ex} \times \mathcal{A}_{ex2}$ de ces deux automates reconnaît donc le mot bbc. La figure 2.4 représente la version élaguée de l'automate $\mathcal{A}_{ex} \times \mathcal{A}_{ex2}$ (c'est-à-dire qu'on a enlevé les états qui ne sont pas atteignables à partir d'un état initial, et les états qui n'atteignent aucun état final).*

2.1.4/ Isomorphisme d'automates

Deux automates $\mathcal{A}_1 = (Q_1, \Sigma, E_1, I_1, F_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma, E_2, I_2, F_2)$ sont *isomorphes* s'il existe une bijection $f : Q_1 \to Q_2$ qui satisfait $(p, a, q) \in E_1$ si et seulement si $(f(p), a, f(q)) \in E_2$, et $f(I_1) = I_2$, $f(F_1) = F_2$.

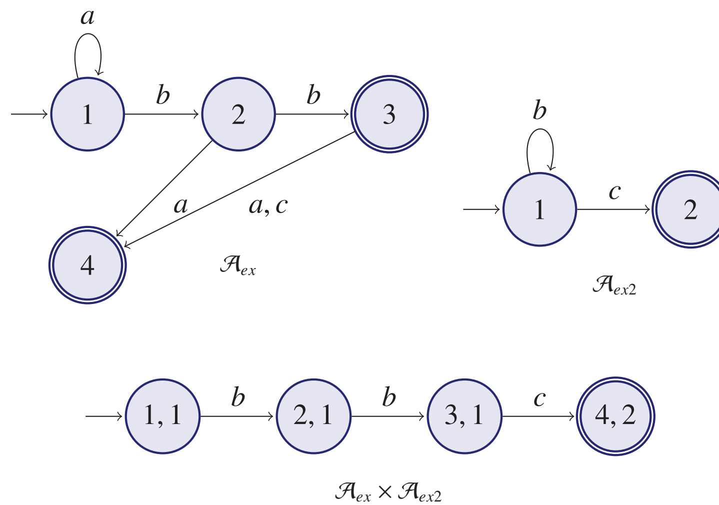


FIGURE 2.4 – $\mathcal{A}_{ex} \times \mathcal{A}_{ex2}$

Autrement dit, deux automates sont isomorphes s'ils sont identiques à un renommage des états près.

Exemple 2.4. *La figure 2.5 représente deux automates isomorphes : $\mathcal{A}_{ex}$ et $\mathcal{A}'_{ex}$. La bijection $f : Q_{ex} \to Q'_{ex}$ est la suivante : $f(1) = 0$, $f(2) = 2$, $f(3) = 1$, $f(4) = 3$.*

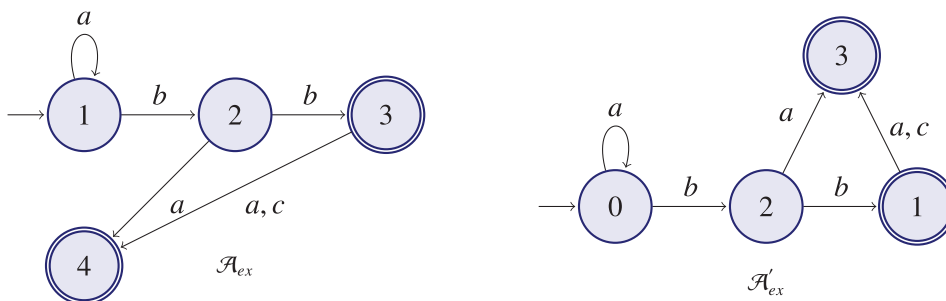


FIGURE 2.5 – Automates isomorphes

2.1.5/ RÉDUCTION D'UN AUTOMATE

Soient $\mathcal{A}$ un automate, $\hat{\mathcal{A}} = (\hat{Q}, \Sigma, \hat{E}, \hat{I}, \hat{F})$ l'automate élagué obtenu à partir de $\mathcal{A}$, et $\sim \subseteq Q \times Q$ une relation d'équivalence. On note $\mathcal{A}/_\sim$ l'automate $(\hat{Q}/_\sim, \Sigma, E', I', F')$ où $E' = \{(\tilde{p}, a, \tilde{q}) \mid \exists p \in \tilde{p}$ et $\exists q \in \tilde{q}$ tels que $(p, a, q) \in \hat{E}\}$, $I' = \{\tilde{p} \mid \exists p \in \tilde{p} \bigcap \hat{I}\}$, et $F' = \{\tilde{p} \mid \exists p \in \tilde{p} \bigcap \hat{F}\}$. Intuitivement, l'automate $\mathcal{A}_{/\sim}$ est obtenu en regroupant les états de chaque classe d'équivalence de la version élaguée de $\mathcal{A}$. Il est trivial que $L(\mathcal{A}) \subseteq L(\mathcal{A}_{/\sim})$.

La notion de réduction d'automate peut également être étendue à toutes les relations : si $\sim$ est une relation sur Q alors $\mathcal{A}_{/\sim}$ est l'automate $\mathcal{A}_{/\eqsim}$, où $\eqsim$ est la fermeture réflexive, symétrique et transitive de $\sim$.

Exemple 2.5. *Soit $\sim$ une relation sur Q définie comme ceci : soit p et q deux états de $\mathcal{A}$, $p \sim q$ si et seulement s'il existe dans $\mathcal{A}$ une transition qui va de p vers q et qui a pour étiquette la lettre*

c. Pour $\mathcal{A}_{ex}$ on a $3 \sim 4$. La relation $\sim$ n'étant pas une relation d'équivalence, $\mathcal{A}_{ex/\sim}$ est l'automate $\mathcal{A}_{ex/\eqsim}$ tel que $\eqsim$ est la fermeture réflexive, symétrique, et transitive de $\sim$. On obtient cet automate en fusionnant l'état 3 avec l'état 4, qui font partie d'une même classe d'équivalence (cf. figure 2.6).

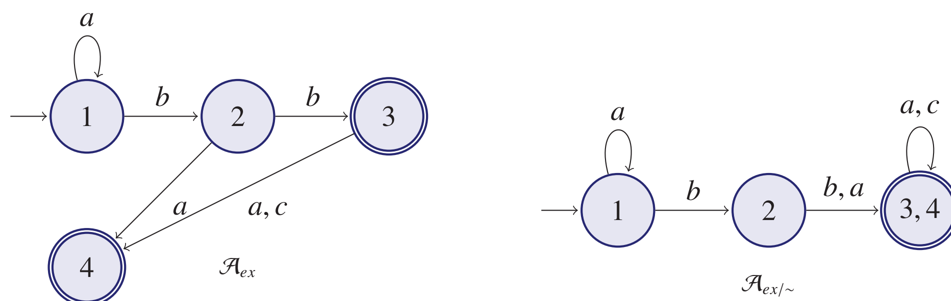
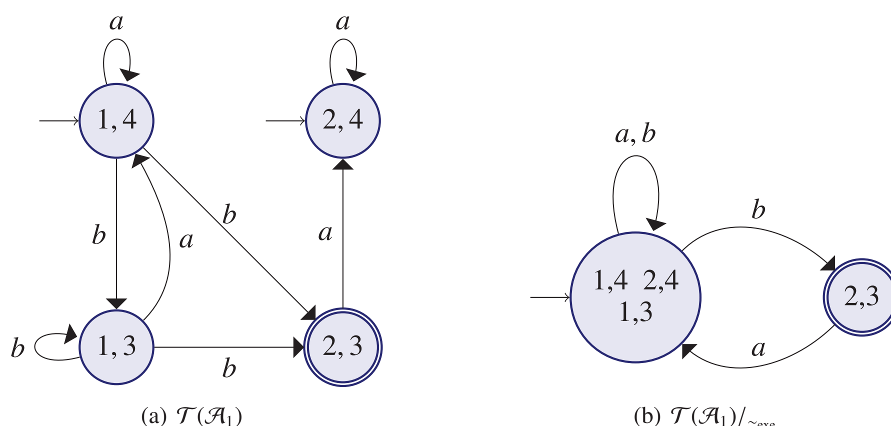


FIGURE 2.6 – $\mathcal{A}_{ex/\sim}$

Exemple 2.6. *D'après l'automate $\mathcal{T}(\mathcal{A}_1)$ (cf. figure 2.11) et la relation $\sim_{\text{exe}}$ dont les classes sont $\{(1,4),(2,4),(1,3)\}$ et $\{(2,3)\}$, l'automate $\mathcal{T}(\mathcal{A}_1)/_{\sim_{\text{exe}}}$ est représenté sur la Fig. 2.7.*



(a) $\mathcal{T}(\mathcal{A}_1)$ (b) $\mathcal{T}(\mathcal{A}_1)/_{\sim_{\text{exe}}}$

FIGURE 2.7 – $\mathcal{T}(\mathcal{A}_1)/_{\sim_{\text{exe}}}$

Définition 2.3. Fonction d'approximation

Une *fonction d'approximation* $\mathfrak{F}$ est une fonction qui associe à tout automate $\mathcal{A}$, une relation $\sim_{\mathfrak{F}}^{\mathcal{A}}$ sur les états de $\mathcal{A}$.

L'automate $\mathcal{A}_{/\sim_{\mathfrak{F}}^{\mathcal{A}}}$ sera noté $\mathfrak{F}(\mathcal{A})$. On défini inductivement $\mathfrak{F}^n(\mathcal{A})$ par $\mathfrak{F}^0(\mathcal{A}) = \mathcal{A}$, et $\mathfrak{F}^n(\mathcal{A}) = \mathfrak{F}(\mathfrak{F}^{n-1}(\mathcal{A}))$ pour $n > 0$.

La fonction d'approximation $\mathfrak{F}$ est *isomorphisme-compatible* si pour toute paire d'automates isomorphes $\mathcal{A}_1$, $\mathcal{A}_2$, et pour tout isomorphisme φ de $\mathcal{A}_1$ vers $\mathcal{A}_2$, $p \sim_{\mathfrak{F}}^{\mathcal{A}_1} q$ si et seulement si $\varphi(p) \sim_{\mathfrak{F}}^{\mathcal{A}_2} \varphi(q)$. Intuitivement, cela signifie que pour être isomorphisme-compatible, une fonction d'approximation ne doit pas dépendre du nom des états.

Exemple 2.7. *Soit $\mathfrak{In}$ la fonction d'approximation qui associe à tout automate $\mathcal{A}$, la relation $\sim_{\mathfrak{In}}^{\mathcal{A}}$ sur les états de $\mathcal{A}$ telle que, soient p et q deux états de $\mathcal{A}$, $p \sim_{\mathfrak{In}}^{\mathcal{A}} q$ si et seulement si l'ensemble*

des étiquettes des transitions qui arrivent sur l'état p est égal à l'ensemble des étiquettes des transitions qui arrivent sur l'état q.

Dans l'automate $\mathcal{A}_{ex}$ de la figure 2.8, 2 $\sim^{\mathcal{A}_{ex}}_{\mathfrak{In}}$ 3 car l'ensemble des étiquettes des transitions qui arrivent sur l'état 2 est égal à l'ensemble des étiquettes des transitions qui arrivent sur l'état 3 (qui est égal à $\{b\}$). Pour obtenir l'automate $\mathfrak{In}(\mathcal{A}_{ex})$, on fusionne donc l'état 2 et l'état 3 de l'automate $\mathcal{A}_{ex}$ (cf. figure 2.8).

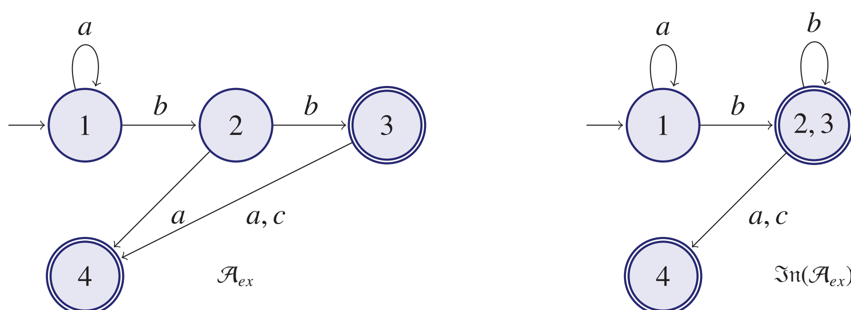


FIGURE 2.8 – $\mathfrak{In}(\mathcal{A}_{ex})$

2.1.6/ TRANSDUCTEURS LETTRE À LETTRE FINIS

Un *transducteur* lettre à lettre fini $\mathcal{T}$ est un automate fini dont l'alphabet Σ est de la forme $\Sigma_e \times \Sigma_s$, où Σ_e est appelé alphabet d'entrée et Σ_s est appelé alphabet de sortie. On notera les étiquettes (a_e, a_s) sous la forme $a_e|a_s$. L'étiquette d'un chemin $a_{e1}|a_{s1} \ldots a_{en}|a_{sn}$ s'écrira sous la forme d'un couple $(a_{e1} \ldots a_{en}, a_{s1} \ldots a_{sn})$.

Tout transducteur $\mathcal{T}$ sur $\Sigma_e \times \Sigma_s$ induit une relation $\mathcal{R}_{\mathcal{T}}$ sur $\Sigma_e^* \times \Sigma_s^*$ définie par : pour tous $a_{ei} \in \Sigma_e$ et $a_{sj} \in \Sigma_s$, $(a_{e1} \ldots a_{en}, a_{s1} \ldots a_{sm}) \in \mathcal{R}_{\mathcal{T}}$ si et seulement si $n = m$ et le mot $(a_{e1}, a_{s1}) \ldots (a_{en}, a_{sn})$ est reconnu par $\mathcal{T}$. Pour tout langage I, $\mathcal{R}_{\mathcal{T}}(\mathrm{I})$ est l'ensemble des mots v tels qu'il existe un mot u dans I, et $(u, v) \in \mathcal{R}_{\mathcal{T}}$. La clôture réflexive et transitive de $\mathcal{R}_{\mathcal{T}}$ est notée $\mathcal{R}_{\mathcal{T}}^*$.

La notion de transducteur est dans la littérature plus générale que la définition donnée ici. Mais comme nous n'utiliserons que des transducteurs lettre à lettre, l'utilisation dans la suite du document du terme *transducteur* désignera un transducteur lettre à lettre fini.

Exemple 2.8. *La figure 2.9 représente $\mathcal{T}_{ex}$, qui est un exemple de transducteur. Soit $\mathcal{T}_{ex} = (Q_{ex}, \Sigma_{eex} \times \Sigma_{sex}, E_{ex}, I_{ex}, F_{ex})$, on a $Q_{ex} = \{1, 2, 3\}$, $\Sigma_{eex} = \{a, b\}$, $\Sigma_{sex} = \{a, b\}$, $E_{ex} = \{(1, a|a, 1), (1, b|b, 1), (1, b|a, 2), (2, a|b, 3), (3, a|a, 3), (3, b|b, 3)\}$, $I_{ex} = \{1\}$ et $F_{ex} = \{1, 3\}$.*

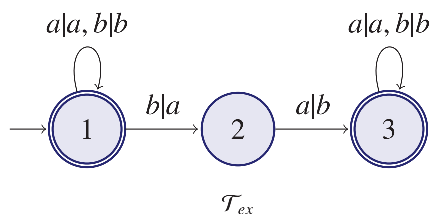


FIGURE 2.9 – $\mathcal{T}_{ex}$ (Exemple de transducteur)

2.1.7/ Appliquer un transducteur à un automate

Soit $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma_{\mathcal{A}}, E_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ un automate, et $\mathcal{T} = (Q_{\mathcal{T}}, \Sigma_{e_{\mathcal{T}}} \times \Sigma_{s_{\mathcal{T}}}, E_{\mathcal{T}}, I_{\mathcal{T}}, F_{\mathcal{T}})$ un transducteur, $\mathcal{T}(\mathcal{A})$ est l'automate (Q, Σ, E, I, F) défini par :

- $Q = Q_{\mathcal{A}} \times Q_{\mathcal{T}}$,
- $\Sigma = \Sigma_{s_{\mathcal{T}}}$,
- $E = \{(p_{\mathcal{A}}, p_{\mathcal{T}}), b, (q_{\mathcal{A}}, q_{\mathcal{T}}) \mid \exists a \in \Sigma_{\mathcal{A}} \bigcap \Sigma_{e_{\mathcal{T}}}$ et $\exists b \in \Sigma_{s_{\mathcal{T}}}$ tels que $(p_{\mathcal{A}}, a, q_{\mathcal{A}}) \in E_{\mathcal{A}}$ et $(p_{\mathcal{T}}, a|b, q_{\mathcal{T}}) \in E_{\mathcal{T}}\}$,
- $I = I_{\mathcal{A}} \times I_{\mathcal{T}}$,
- $F = F_{\mathcal{A}} \times F_{\mathcal{T}}$.

Intuitivement, l'application d'un transducteur $\mathcal{T}$ à un automate $\mathcal{A}$ ressemble beaucoup à un produit d'automates. Sauf qu'on utilise l'alphabet d'entrée du transducteur pour trouver les chemins en commun, mais qu'on remplace ensuite la lettre d'entrée par la lettre de sortie correspondante.

Par définition, $L(\mathcal{T}(\mathcal{A}))$ est l'ensemble des mots v qui satisfont $(u, v) \in \mathcal{R}_{\mathcal{T}}$ avec $u \in L(\mathcal{A})$. Si $\mathcal{T} = (Q_{\mathcal{T}}, \Sigma_{e_{\mathcal{T}}} \times \Sigma_{s_{\mathcal{T}}}, E_{\mathcal{T}}, I_{\mathcal{T}}, F_{\mathcal{T}})$ est un transducteur, on note $\mathcal{T}^{-1}$ le transducteur $(Q_{\mathcal{T}}, \Sigma_{s_{\mathcal{T}}} \times \Sigma_{e_{\mathcal{T}}}, E'_{\mathcal{T}}, I_{\mathcal{T}}, F_{\mathcal{T}})$ avec $E'_{\mathcal{T}} = \{(p, (b, a), q) \mid (p, (a, b), q) \in E_{\mathcal{T}}\}$. On peut démontrer que $(u, v) \in \mathcal{R}_{\mathcal{T}}$ si et seulement si $(v, u) \in \mathcal{R}_{\mathcal{T}^{-1}}$.

Exemple 2.9. $\mathcal{T}_{ex}(\mathcal{A}_{ex})$ *L'automate $\mathcal{A}_{ex}$ reconnaît les mots de la forme a^*ba, a^*bb, a^*bba et a^*bbc, et le transducteur $\mathcal{T}_{ex}$ reconnaît les mots de la forme $[(a|a) \cup (b|b)]^*(b|a)(a|b)[(a|a) \cup (b|b)]^*$. L'application $\mathcal{T}_{ex}(\mathcal{A}_{ex})$ du transducteur $\mathcal{T}_{ex}$ à l'automate $\mathcal{A}_{ex}$ reconnaît donc les mots de la forme a^*ab, a^*bab, a^*ba, a^*bb, et a^*bba. La figure 2.10 représente une version élaguée de l'automate $\mathcal{T}_{ex}(\mathcal{A}_{ex})$ (c'est-à-dire qu'on a enlevé les états qui ne sont pas atteignables à partir d'un état initial, et les états qui n'atteignent aucun état final).*

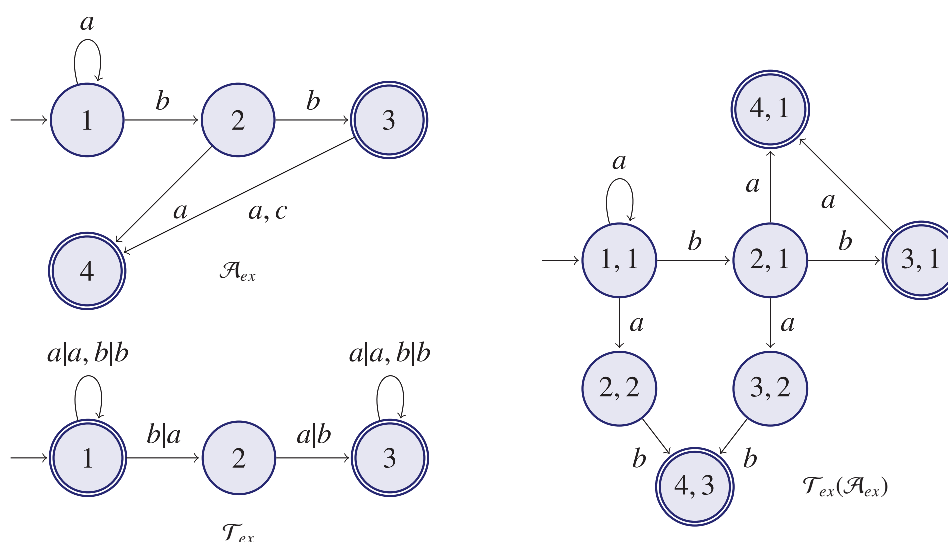
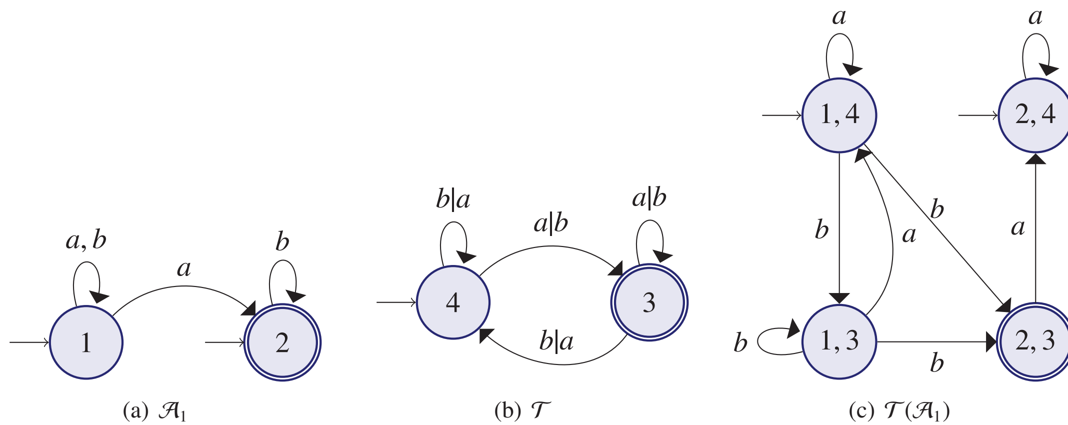


Figure 2.10 – $\mathcal{T}_{ex}(\mathcal{A}_{ex})$

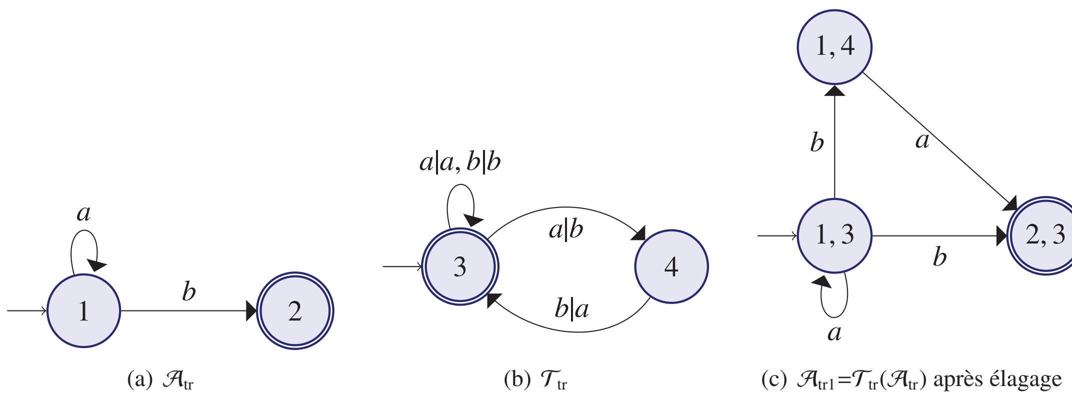
Exemple 2.10. $\mathcal{T}(\mathcal{A}_1)$ *D'après le transducteur $\mathcal{T}$ et l'automate $\mathcal{A}_1$ (cf. figure 2.11), $\mathcal{T}(\mathcal{A}_1)$ est représenté sur la figure 2.11.*

Exemple 2.11. Token ring



(a) $\mathcal{A}_1$ (b) $\mathcal{T}$ (c) $\mathcal{T}(\mathcal{A}_1)$

FIGURE 2.11 – $\mathcal{T}(\mathcal{A}_1)$



(a) $\mathcal{A}_{tr}$ (b) $\mathcal{T}_{tr}$ (c) $\mathcal{A}_{tr1}$=$\mathcal{T}_{tr}(\mathcal{A}_{tr})$ après élagage

FIGURE 2.12 – Token ring

2.1.8/ Appliquer un transducteur à un langage

Soit $\mathcal{T} = (Q, \Sigma_e \times \Sigma_s, E, I, F)$ un transducteur et K un langage, l'application de $\mathcal{T}$ à K donne un langage $\mathcal{T}(K) = \{v \in \Sigma_s^* | \exists u \in K, (u, v) \in L(\mathcal{T})\}$ (avec la convention $(a|b)(c|d) = (ac, bd)$).

Proposition 2.12 *Soit $\mathcal{A}$ un automate et $\mathcal{T}$ un transducteur,*

$$\mathcal{T}(L(\mathcal{A})) = L(\mathcal{T}(\mathcal{A}))$$

L'application d'un transducteur à un automate et l'application d'un transducteur à un langage sont deux opérations étroitement liées. Il est connu que l'application d'un transducteur $\mathcal{T}$ au langage reconnu par un automate $\mathcal{A}$ donnera comme résultat le langage reconnu par $\mathcal{T}(\mathcal{A})$.

2.2/ Grammaires algébriques et automates à pile

2.2.1/ Grammaires algébriques

Une *grammaire algébrique* – appelée aussi parfois *grammaire non contextuelle* ou encore *grammaire hors contexte* – est un 4-uplet $G = (\Sigma, \Gamma, S_0, R)$, où :

- Σ et Γ sont des alphabets finis disjoints,
- $S_0 \in \Gamma$ est le symbole initial,
- R est un sous-ensemble fini de $\Gamma \times (\Sigma \cup \Gamma)^*$.

Les éléments de Σ sont appelés *symboles terminaux*, et les éléments de Γ sont appelés *symboles non-terminaux*. Un élément (X, u) de R est appelé *une règle* de la grammaire et est fréquemment noté $X \to u$. Un mot $u_1 \in (\Sigma \cup \Gamma)^*$ est un *successeur* de $u_0 \in (\Sigma \cup \Gamma)^*$ dans la grammaire G s'il existe $v \in \Sigma^*$, $w, x \in (\Sigma \cup \Gamma)^*$, $S \in \Gamma$ tels que $u_0 = vSw$ et $u_1 = vxw$ et $S \to x \in R$.

Une *dérivation complète* [1] de la grammaire G est une séquence finie $u_0, \ldots, u_k$ de mots de $(\Sigma \cup \Gamma)^*$ telle que $u_0 = S_0$, $u_k \in \Sigma^*$ et pour tout i, u_{i+1} est un successeur de u_i. Un *arbre de dérivation* de G est un arbre fini dont les nœuds internes sont étiquetés par des lettres de Γ, dont les feuilles sont étiquetées par des éléments de $\Sigma \cup \{\varepsilon\}$, dont la racine est étiquetée par S_0 et qui satisfait : si un nœud est étiqueté par $X \in \Gamma$ et si ses fils sont étiquetés par $\alpha_1, \ldots, \alpha_k$ (dans cet ordre), alors $(X, \alpha_1 \ldots \alpha_k) \in R$ et soit $\alpha_1 = \varepsilon$ et $k = 1$, soit tous les α_i's sont dans $\Gamma \cup \Sigma$. La taille d'un arbre de dérivation est donnée par le nombre de nœuds de l'arbre.

Notons qu'il existe une bijection entre l'ensemble des dérivations complètes d'une grammaire et l'ensemble des arbres de dérivations de cette grammaire. Pour une grammaire algébrique G, $E_n(G)$ représente l'ensemble des arbres de dérivation de G composés de n nœuds. On dit qu'un arbre de dérivation *couvre* un élément X de Γ si au moins un de ses nœuds est étiqueté par X. Et on dit qu'un arbre de dérivation couvre une règle $X \to u$ de R si cette règle est utilisée dans la dérivation complète correspondante.

Exemple 2.13. Grammaire algébrique *Soit la grammaire* $G = (\{a, b\}, \{S, T\}, S, R)$*, avec* $R = \{S \to Tb, S \to aSb, T \to \varepsilon\}$*). La séquence* $S, aSb, aTbb, abb$ *est une dérivation complète de la grammaire. L'arbre de dérivation associé à cette séquence est représenté dans la figure 2.13. On peut y voir que les éléments* S *et* T *ainsi que la règle* $S \to aSb$ *sont couverts, puisqu'ils apparaissent dans l'arbre de dérivation.*

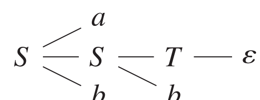


FIGURE 2.13 – Arbre de dérivation correspondant à la séquence $S, aSb, aTbb, abb$

2.2.2/ AUTOMATES À PILE

Un *automate à pile* est un 6-uplet $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ où :

- Q est l'ensemble fini des *états*,
- Σ (alphabet d'entrée) et Γ (alphabet de la pile) sont des alphabets finis disjoints tels que $\varepsilon \notin \Sigma$ et $\perp \in \Gamma$,
- $q_{\text{init}} \in Q$ est l'état initial,
- F est l'ensemble des états finaux,
- δ est une fonction partielle de $Q \times (\Sigma \cup \{\varepsilon\}) \times \Gamma$ vers $Q \times \Gamma^*$ telle que pour tout $q \in Q$, pour tout $X \in \Gamma \setminus \{\perp\}$, et pour tout $a \in \Sigma \cup \{\varepsilon\}$:
  - si $\delta(q, a, X) = (p, w)$ alors $w \in (\Gamma \setminus \{\perp\})^*$,
  - si $\delta(q, a, \perp) = (p, w)$ alors $w = \perp w'$ avec $w' \in (\Gamma \setminus \{\perp\})^*$.

La lettre $\perp$ est appelée le *symbole de pile vide*, ou encore le *fond de la pile*. Le terme "automate à pile" est abrégé en *PDA*, pour l'anglais PushDown Automaton. Une *configuration* d'un PDA est un élément de $Q \times \{\perp\}(\Gamma \setminus \{\perp\})^*$, c'est-à-dire une paire dont le premier élément est dans Q, et le second élément est un mot qui commence par $\perp$ et dont aucune des autres lettres n'est $\perp$. Intuitivement, la seconde partie encode la valeur courante de la pile. La *configuration initiale* est

1. Comme $v \in \Sigma^*$, cette dérivation est une dérivation gauche.

$(q_{\text{init}}, \bot)$. Deux configurations $(q, \bot u)$ et $(p, \bot v)$ sont *a-consécutives*, avec $a \in \Sigma \cup \{\varepsilon\}$, si $u = \varepsilon$ et $\delta(q, a, \bot) = (p, \bot v)$, ou si u est de la forme $u_0 X$ avec $X \in \Gamma \setminus \{\bot\}$ et qu'il existe $w \in \Gamma^*$ tel que $\delta(q, a, X) = (p, w)$ et $v = u_0 w$.

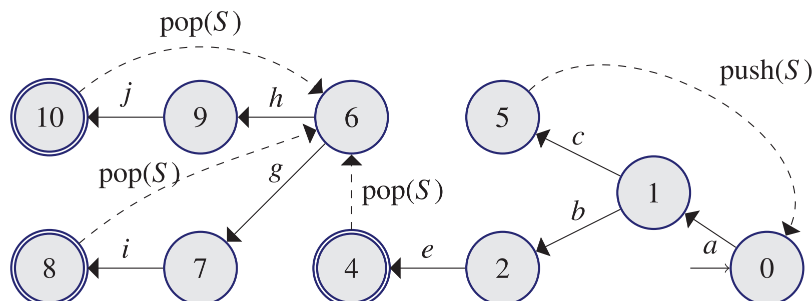


FIGURE 2.14 – $\mathcal{A}_{\text{power}}$, exemple de NPDA

Une *PDA-trace* de longueur n dans un PDA est une séquence $C_1 a_1 C_2 a_2 \ldots C_n a_n C_{n+1}$ où les C_i's sont des configurations, les a_i's sont dans $\Sigma \cup \{\varepsilon\}$ et tels que C_1 est la configuration initiale, pour tout i, C_i et C_{i+1} sont a_i-consécutives, et C_{n+1} est de la forme $(p, \bot)$ avec $p \in F$. Un *automate à pile normalisé* – abrégé en *NPDA*, pour l'anglais Normalised PushDown Automaton – est un PDA tel que pour tout état q, pour tout $a \in \Sigma \cup \{\varepsilon\}$, et pour tout $X \in \Gamma$, si $\delta(q, a, X) = (p, w)$ alors on se trouve dans l'un des cas suivants :

(i) $a = \varepsilon$ et $w = \varepsilon$, ou

(ii) $a = \varepsilon$ et w est de la forme $w = XY$ avec $Y \in \Gamma$, ou

(iii) $a \neq \varepsilon$ et $w = X$.

Il faut également que si $\delta(q, \varepsilon, X) = (p, XY)$, alors pour tout $Z \in \Gamma$, $\delta(q, \varepsilon, Z) = (p, ZY)$ – c'est-à-dire qu'une transition qui ajoute un élément au sommet de la pile peut être déclenchée indépendamment de l'état de la pile –, et que si $\delta(q, a, X) = (p, X)$, alors pour tout $Z \in \Gamma$, $\delta(q, a, Z) = (p, Z)$ – c'est-à-dire qu'une transition qui lit une lettre en entrée peut également être déclenchée indépendamment de l'état de la pile –.[2]

Intuitivement, le cas (*i*) correspond à l'action de dépilement (*pop*) – c'est-à-dire retirer l'élément au sommet de la pile –, le cas (*ii*) correspond à l'action d'empilement (*push*) – c'est-à-dire ajouter un élément au sommet de la pile –, et le cas (*iii*) correspond à l'action de lecture (*read*) – c'est-à-dire consommer l'élément suivant de l'entrée (sans modifier la pile) –. L'*automate sous-jacent* d'un NPDA $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ est l'automate $(Q, \Sigma \cup \{\text{pop(X)}, \text{push(X)} \mid X \in \Gamma\}, \mu, q_{\text{init}}, F)$, où l'ensemble des transitions μ est défini par :

(i) $\delta(q, \varepsilon, X) = (p, \varepsilon)$ si et seulement si $\mu(q, \text{pop(X)}) = p$,

(ii) $\delta(q, \varepsilon, X) = (p, XY)$ si et seulement si $\mu(q, \text{push(Y)}) = p$,

(iii) $\delta(q, a, X) = (p, X)$ si et seulement si $\mu(q, a) = p$.

Une *NPDA-trace* est une PDA-trace dans un NPDA. Les définitions ci-dessus peuvent être illustrées par le NPDA de la figure 2.14, qui correspond au graphe à droite de la figure 7.10. Pour cet automate $Q = \{0, \ldots, 10\}$, $\Sigma = \{a, b, c, e, g, h, i, j\}$, $\Gamma = \{\bot, S\}$, $q_{\text{init}} = 0$, $F = \{4, 8, 10\}$ et δ est

2. Ces conditions ne réduisent pas le pouvoir d'expression puisqu'il est possible d'encoder toute transition de la forme $\delta(q, \varepsilon, X) = (p, XY)$ qui requiert un X au sommet de la pile pour être déclenchée, par $\delta(q, \varepsilon, X) = (q_{\text{new1}}, \varepsilon)$ et pour tout $Z \in \Gamma$, $\delta(q_{\text{new1}}, \varepsilon, Z) = (q_{\text{new2}}, ZX)$ et $\delta(q_{\text{new2}}, \varepsilon, Z) = (p, ZY)$. Il en est de même pour toute transition de la forme $\delta(q, a, X) = (p, X)$ qui requiert un X au sommet de la pile pour être déclenchée, qui peut être encodée par $\delta(q, \varepsilon, X) = (q_{\text{new1}}, \varepsilon)$ et pour tout $Z \in \Gamma$, $\delta(q_{\text{new1}}, \varepsilon, Z) = (q_{\text{new2}}, ZX)$ et $\delta(q_{\text{new2}}, a, Z) = (p, Z)$.

défini par les flèches : pour une arête de la forme (q, a, p) avec $a \in \Sigma$, alors on a $\delta(q, a, \bot) = (p, \bot)$ et $\delta(q, a, S) = (p, S)$; pour une arête de la forme $(q, \mathrm{pop}(S), p)$ alors $\delta(q, \varepsilon, S) = (p, \varepsilon)$; pour une arête de la forme $(q, \mathrm{push}(S), p)$ alors $\delta(q, \varepsilon, \bot) = (p, \bot S)$ et $\delta(q, \varepsilon, S) = (p, SS)$. La séquence

$$(0, \bot)a(1, \bot)c(5, \bot)\varepsilon(0, \bot S)a(1, \bot S)c(5, \bot)\varepsilon(0, \bot SS)a(1, \bot SS)b(2, \bot SS)$$
$$e(4, \bot SS)\varepsilon(6, \bot S)h(9, \bot S)j(10, \bot S)\varepsilon(6, \bot)g(7, \bot)i(8, \bot)$$

est une NPDA-trace dans le NPDA. Cette trace correspond à l'exécution de `power(3,2)`, que l'on a détaillée dans l'exemple 7.3. Notons que chaque paire (q, w) signifie que le système est dans l'état q et que la pile d'appels est w. Par exemple, $(2, \bot SS)$ signifie que le système est dans l'état 2 et qu'il y a deux appels non terminés à la fonction `power`.

Chaque NPDA-trace d'un NPDA est associée à une DFA-trace dans l'automate sous-jacent selon la projection suivante : la NPDA-trace $C_1 a_1 \ldots a_n C_n$ est associée au chemin $(q_1, a_1, q_2) \ldots (q_{n-1}, a_n, q_n)$, où les C_i's sont de la forme (q_i, w_i). Par exemple, la DFA-trace associée à la NPDA-trace ci-dessus est $(0, a, 1)(1, c, 5)(5, \varepsilon, 0) \ldots (6, g, 7)(7, i, 8)$. Cette projection est notée proj et est injective. Son image forme un sous-ensemble de DFA-traces, appelées *DFA-traces cohérentes* de l'automate sous-jacent, qui est en bijection avec l'ensemble des NPDA-traces.

2.2.3/ Génération aléatoire uniforme d'un arbre de dérivation dans une grammaire

Le problème de la *génération aléatoire uniforme* sur les grammaires peut s'écrire ainsi :

Problème 2.14. Génération aléatoire d'un arbre de dérivation dans une grammaire

Entrée : Une grammaire algébrique G, un nombre entier positif n Question : Générer aléatoirement avec une distribution uniforme, des arbres de dérivation de G de taille n ?

Nous allons expliquer ici comment résoudre ce problème en utilisant des techniques combinatoires bien connues [FS08]. Notons que des techniques plus avancées permettent un calcul plus rapide, comme dans [DZ99].

Pour faciliter la compréhension, on utilise habituellement des lettres majuscules pour nommer les symboles non-terminaux Soit une grammaire algébrique $G = (\Sigma, \Gamma, S_0, R)$, un symbole non-terminal X de Γ, et un entier positif i, le nombre d'arbres de dérivation de taille i généré par (Σ, Γ, X, R) est noté $x(i)$ – c'est-à-dire qu'on utilise la lettre minuscule correspondante au symbole initial –.

Soit un nombre entier positif n, pour tout symbole $S \in \Gamma$, on peut calculer la séquence d'entiers positifs $s(1), \ldots, s(k), \ldots$. Le calcul récursif de ces $s(i)$ est le suivant : Pour tout entier strictement positif k et toute règle $r = (S, w_1 S_1 \ldots w_n S_n w_{n+1}) \in R$, avec $w_j \in \Sigma^*$ et $S_i \in \Gamma$, on pose

$$\begin{cases} \beta_r = 1 + \sum_{i=1}^{n+1} |w_i|, \\ \alpha_r(k) = \sum_{i_1+i_2+\ldots+i_n=k-\beta_r} \prod_{j=1}^{j=n} s_j(i_j) \quad \text{si } n \neq 0, \\ \alpha_r(k) = 0 \quad \text{si } n = 0 \text{ et } k \neq \beta_r, \\ \alpha_r(\beta_r) = 1 \quad \text{si } n = 0. \end{cases}$$

Il est connu que $s(k) = \sum_{r \in R \cap (S \times (\Sigma \cup \Gamma)^*)} \alpha_r(k)$ (*cf.* [FS08, Théorème I.1]).

Par hypothèse, il n'y a pas de règle de la forme (S, T) dans R avec $S, T \in \Gamma$, donc tous les i_j dans la définition de α_r sont strictement inférieurs à k. Les $s(i)$ peuvent donc être calculés récursivement.

Considérons par exemple la grammaire $(\{a,b\},\{X\},X,\{r_1,r_2,r_3\})$ avec $r_1 = (X,XX)$ $r_2 = (X,a)$ et $r_3 = (X,b)$. On a $\beta_{r_1} = 1+0 = 1, \beta_{r_2} = 1+1 = 2, \beta_{r_3} = 1+1 = 2$. Donc $x(k) = \sum_{i+j=k-1} x(i)x(j)$ si $k \neq 2$, et $x(2) = 1+1+\sum_{i+j=2-1} x(i)x(j) = 2$, sinon. Par conséquent, $x(1) = 0$, $x(2) = 2$, $x(3) = x(1)x(1) = 0$, $x(4) = x(1)x(2)+x(2)x(1) = 0$, $x(5) = x(2)x(2) = 4$, etc.... Les deux arbres de dérivation de taille 2 sont $\overset{X}{\underset{a}{|}}$ et $\overset{X}{\underset{b}{|}}$. Les quatre arbres de dérivation de taille 5 sont les arbres de la forme $\overset{X}{\underset{Z_1\,Z_2}{/\backslash}}$, où Z_1 et Z_2 sont des arbres de dérivation de taille 2.

Pour générer des arbres de dérivation de taille n, on doit calculer tous les $s(i)$, pour $S \in \Gamma$ et $i \leq n$, avec la méthode ci-dessus. Ce calcul peut être effectué en un temps polynomial. Ensuite, la génération aléatoire consiste à appliquer récursivement l'algorithme décrit dans la figure 2.15.

Génération aléatoire
Entrée : une grammaire algébrique $G = (\Sigma,\Gamma,X,R)$, un nombre entier strictement positif n.
Sortie : un arbre de dérivation t de G, de taille n.

Algorithme :

1. Soit $\{r_1,r_2,\ldots,r_\ell\}$ l'ensemble des éléments de R dont le premier élément est X.
2. **Si** $\sum_{j=1}^{j=\ell} \alpha_{r_j}(n) = 0$, **alors retourner** "Exception".
3. **Choisir** $i \in \{1,\ldots,\ell\}$ avec une probabilité $Prob(i = j) = \frac{\alpha_{r_i}(n)}{\sum_{j=1}^{j=\ell}\alpha_{r_j}(n)}$.
4. $r_i = (X, Z_1 \ldots Z_k)$, avec $Z_j \in \Sigma \cup \Gamma$.
5. Le symbole racine de t est X.
6. Les fils de t sont $Z_1,\ldots,Z_k$ dans cet ordre.
7. $\{i_1,\ldots,i_m\} = \{j \mid Z_j \in \Gamma\}$.
8. **Choisir** $(x_1,\ldots,x_m) \in \mathbb{N}^m$ tel que $x_1+\ldots+x_m = n-\beta_{r_i}$ avec une probabilité

$$Prob(x_1 = \ell_1,\ldots,x_m = \ell_m) = \frac{\prod_{j=1}^{j=m} z_{i_j}(\ell_j)}{\alpha_{r_i}(n)}.$$

9. Pour tout i_j, le i_j-ème sous-arbre de T est obtenu en appliquant l'algorithme **Génération aléatoire** sur $(\Sigma,\Gamma,Z_{i_j},R)$ et ℓ_j.
10. **Retourner** t.

FIGURE 2.15 – Algorithme Génération aléatoire

Il est connu [FS08] que cet algorithme produit une génération uniforme d'arbres de dérivations de taille n – c'est-à-dire que chaque arbre de dérivation a la même probabilité d'être généré –. Notons qu'une exception est renvoyée à l'étape 2 s'il n'y a aucun élément de la taille donnée. Pour l'exemple présenté précédemment, il n'y a pas d'élément de taille 3, il n'est donc pas possible de générer un arbre de dérivation de taille 3. En exécutant l'algorithme sur cet exemple avec $n = 2$, à l'étape 1 on considère l'ensemble $\{r_1,r_2,r_3\}$ puisque toutes ces règles ont X comme élément gauche. Comme $\alpha_{r_1}(2) = 0$, $\alpha_{r_2}(2) = 1$, $\alpha_{r_3}(2) = 1$, à l'étape 3 la probabilité que $i = 1$ est 0, la probabilité que $i = 2$ est 1/2 et la probabilité que $i = 3$ est 1/2. Si la valeur $i = 2$ est choisie, l'arbre généré a X comme symbole racine et a comme unique fils. L'exécution de l'algorithme avec $n = 3$ s'arrête à l'étape 2 puisqu'il n'y a pas d'arbre de taille 3. En exécutant l'algorithme sur cet exemple avec $n = 5$, à l'étape 1 on considère l'ensemble $\{r_1,r_2,r_3\}$. Comme $\alpha_{r_1}(5) = 4$, $\alpha_{r_2}(5) = 0$, $\alpha_{r_3}(5) = 0$, la valeur $i = 1$ est choisie avec une probabilité 1. Par conséquent, l'arbre a X comme symbole racine, et ses deux fils sont étiquetés par X. C'est pourquoi à l'étape 7, l'ensemble considéré est $\{1,2\}$. A l'étape 8, on a $n-\beta_{r_1} = 5-1 = 4$. La probabilité que $i_1 = 1$ et $i_2 = 3$ est 0 car $x(1) = 0$. De même, la probabilité que $i_1 = 3$ et $i_2 = 1$ est aussi 0. La probabilité que $i_1 = 2$ et

$$\begin{aligned}
\Sigma_G = \{&(0, a, 1), (1, c, 5), (5, \text{push}(S), 0), (1, b, 2), (2, e, 4),\\
&(4, \text{pop}(S), 6), (6, g, 7), (7, i, 8), (8, \text{pop}(S), 6), (6, h, 9), (9, j, 10),\\
&(10, \text{pop}(S), 6)\}, \text{et}\\
R_G = \{&(T, LRD(6, g, 7)(7, i, 8)),\\
&(T, LRD(6, h, 9)(9, j, 10)),\\
&(T, (0, a, 1)(1, b, 2)(2, e, 4)),\\
&(R, LRD),\\
&(R, (0, a, 1)(1, c, 5)(5, \text{push}(S), 0)(0, a, 1)(1, b, 2)(2, e, 4)(4, \text{pop}(S), 6)),\\
&(L, (0, a, 1)(1, c, 5)(5, \text{push}(S), 0)),\\
&(D, (6, g, 7)(7, i, 8)(8, \text{pop}(S), 6)),\\
&(D, (6, h, 9)(9, j, 10)(10, \text{pop}(S), 6))\}.
\end{aligned}$$

FIGURE 2.16 – Exemple d'une grammaire qui génère des DFA-traces cohérentes

$i_2 = 2$ est donc 1. L'algorithme est ensuite exécuté récursivement sur chacun des fils avec $n = 2$: chacun des quatre arbres est choisi avec une probabilité $1/4$.

Puisqu'on peut très simplement représenter un graphe sous la forme d'une grammaire, on peut utiliser cette méthode de génération aléatoire uniforme d'un arbre de dérivation dans une grammaire afin de générer aléatoire-uniformément des chemins dans un graphe.

Considérons par exemple l'automate $\mathcal{A}$, sous-jacent au NPDA décrit dans la figure 2.14. Des DFA-traces cohérentes de $\mathcal{A}$ sont générées à partir de la grammaire $(\Sigma_G, \Gamma_G, T, R_G)$ de la figure 2.16, avec $\Gamma_G = \{T, L, R, D\}$.

2.2.4/ D'UN NPDA VERS UNE GRAMMAIRE ALGÉBRIQUE

Plusieurs algorithmes classiques permettent la transformation d'un NPDA en une grammaire algébrique (par exemple, [Sip96]). Le théorème présenté ci-dessous est une combinaison directe de résultats bien connus sur les PDA.

Théorème 2.15 *Soit $\mathcal{A}$ un NPDA, on peut calculer en temps polynomial une grammaire G, qui satisfait les assertions suivantes :*

- *Le nombre de règles de G est au plus quadratique de la taille de $\mathcal{A}$, et G ne contient aucune règle de la forme (X, Y), avec X et Y qui sont des symboles de la pile.*
- *Il existe une bijection φ de l'ensemble des dérivations complètes de G vers l'ensemble des NPDA-traces de $\mathcal{A}$.*
- *L'image par φ d'une dérivation complète de G peut être calculée en temps polynomial.*

Notons que la complexité précise des algorithmes dépend des structures de données choisies, mais qu'elles peuvent toutes être implémentées de manière efficace.

II

Approximations régulières pour l'analyse d'accessibilité

Dans cette partie, notre contribution est la définition de deux nouvelles techniques d'approximation pour le model-checking régulier, dans le but de fournir des (semi-)algorithmes efficaces. On calcule des sur-approximations de l'ensemble des états accessibles, avec l'objectif d'assurer la terminaison de l'exploration de l'espace d'états. L'ensemble des états accessibles (ou les sur-approximations de cet ensemble) est représenté par un langage régulier, lui même codé par un automate fini. La première technique consiste à sur-approximer l'ensemble des états atteignables en fusionnant des états des automates, en fonction de critères syntaxiques simples, ou d'une combinaison de ces critères (*cf.* section 4). La seconde technique d'approximation consiste aussi à fusionner des états des automates, mais cette fois-ci à l'aide d'un transducteur (*cf.* section 5.2). De plus, pour cette seconde technique, nous développons une nouvelle approche pour raffiner les approximations (*cf.* section 5.3), qui s'inspire du paradigme CEGAR (*cf.* section 3.4). Les propositions sont évaluées sur des exemples de protocoles d'exclusion mutuelle (*cf.* section 4.4 et section 5.4).

3

Contexte et problématiques

3.1/ Introduction et énoncé du problème

Dans ce chapitre, nous allons présenter le problème d'accessibilité, puis sa traduction dans le cadre plus particulier du model-checking régulier qui en est un cas particulier. Enfin, nous présenterons l'approche par approximations régulières qui est une façon d'aborder le problème de l'analyse d'accessibilité par model-checking régulier.

Analyse d'accessibilité

L'analyse d'accessibilité est une question importante dans la vérification de logiciels, car elle permet d'en assurer la sécurité et la sûreté (c'est-à-dire que quelque chose de mauvais ne se produira pas). Elle consiste à vérifier (prouver) sur une description formelle du système, que certaines configurations peuvent être atteintes. On peut exprimer le problème de la façon suivante :

Problème 3.1. Accessibilité

Entrée : Un ensemble de configurations initiales I, une relation d'évolution du système $\mathcal{R}$, un ensemble d'états B.

Question : Le système décrit par I et $\mathcal{R}$, peut-il atteindre un état de B ?

Par exemple, savoir si un point de programme particulier est atteignable est un problème d'accessibilité, de même que de savoir si une machine de Turing reconnaît un langage non vide. Le problème d'accessibilité est indécidable dans la plupart des formalismes. Plusieurs approches ad hoc ont été développées afin de trouver des heuristiques ou de déterminer des formalismes pour lesquels le problème est décidable. Parmi ces approches, l'analyse d'accessibilité symbolique repose sur des représentations finies d'ensembles infinis d'états. Le *model checking régulier* – abrégé en *RMC*, pour l'anglais Regular Model Checking – est une approche symbolique utilisant des ensembles réguliers pour représenter des ensembles d'états : I et B sont des langages réguliers, donnés le plus souvent par des automates finis (parfois des expressions régulières) et R est les plus souvent une relation régulière donnée par un transducteur (ou un système de réécriture). Dans ce cadre, le problème d'accessibilité reste indécidable et est abordé des deux façons suivantes :

- soit en travaillant sur des classes d'ensembles réguliers et des relations pour lesquelles le problème de l'accessibilité est décidable (voir par exemple [GGP08]),
- soit en développant des approches semi-algorithmiques basées sur des accélérations ou des approximations (voir par exemple [DLS01, DLS02]).

L'analyse d'accessibilité permet de vérifier des propriétés de sûreté : si l'ensemble d'états B est un ensemble d'états d'erreurs, et que l'analyse d'accessibilité montre que le système ne peut pas atteindre un état de B, alors on a vérifié une propriété de sûreté (c'est une propriété qui dit que

quelque chose de mauvais ne doit pas pouvoir se produire, comme par exemple le fait que le train d'atterrissage d'un avion ne doit pas pouvoir être rentré si l'avion est au sol). D'autres types de propriétés, plus complexes, peuvent souvent se recoder dans le cadre de l'analyse d'accessibilité.

Model-Checking régulier

Le problème de l'analyse d'accessibilité par model-checking régulier sur les mots peut se formaliser ainsi :

Problème 3.2. RMC-1

Entrée : Un langage régulier I (ensemble de configurations initiales), une relation calculable $\mathcal{R}$ (relation d'évolution du système), un langage régulier B (propriété d'erreur).

Question : $\mathcal{R}^*(\mathrm{I}) \bigcap \mathrm{B} = \emptyset$?

En représentant l'ensemble de configurations initiales par un automate $\mathcal{A}_0$, la relation d'évolution du système par un transducteur $\mathcal{T}$, et la propriété d'erreur par un automate $\mathcal{B}$, on peut définir le problème de l'analyse d'accessibilité par RMC sur les mots de la façon suivante :

Problème 3.3. RMC-2

Entrée : Deux automates finis $\mathcal{A}_0$ et $\mathcal{B}$ sur un même alphabet Σ, et un transducteur $\mathcal{T}$ sur $\Sigma \times \Sigma$.

Question : $\mathcal{R}^*_{\mathcal{T}}(L(\mathcal{A}_0)) \cap L(\mathcal{B}) = \emptyset$?

Comme la sortie dépend de la fermeture réflexive et transitive, on peut supposer sans perte de généralité que pour tout $u \in \Sigma^*$, $(u, u) \in \mathcal{R}_{\mathcal{T}}$. On supposera donc que toutes les relations étudiées par la suite contiennent l'identité.

Depuis presque 20 ans, le model-checking régulier est un domaine de recherche actif en informatique (*cf.* [CGP00] et [BKe08] pour une vue d'ensemble). De nombreux travaux ont donc traité (et traitent encore) de variantes de ce problème. Par exemple, dans [KMM+97], les auteurs proposent de représenter les configurations possibles d'un tableau de processus sous la forme d'un ensemble régulier de mots – autrement dit, un langage régulier – défini par un automate $\mathcal{A}$, et de représenter l'effet d'une action du système sous la forme d'une règle de réécriture qui modifie l'automate $\mathcal{A}$ (un transducteur). Mais ce travail traite uniquement des transducteurs qui représentent l'effet d'une seule application de la transition, et il y a de nombreux protocoles pour lesquels l'analyse d'accessibilité ne se termine pas. Puis dans [WB98], les auteurs présentent l'idée de vérifier différents types de systèmes infinis en les représentant sous la forme de langages réguliers.

Les principales méthodes pour contourner le problème de non-terminaison de l'analyse d'accessibilité, et donc pour atteindre un point fixe, sont l'accélération [JN00, BJNT00, DLS01, DLS02, AJNd03, BLW03], l'élargissement (extrapolation) [BJNT00, Tou01, Leg12], et l'abstraction d'automates [BHV04].

Dans [BJNT00], les auteurs présentent le concept du model-checking régulier, et s'intéressent au calcul des états atteignables à partir d'un ensemble donné d'états initiaux, ainsi qu'au calcul de la fermeture transitive de la relation de transition. Ils proposent pour cela une technique s'appuyant sur la théorie des automates, et une technique d'accélération. Cet article est l'un des premiers à présenter la notion de model-checking régulier, et il est donc souvent utilisé comme référence pour les travaux dans ce domaine. Il s'intéresse au calcul des états atteignables à partir d'un ensemble donné d'états initiaux, et de la fermeture transitive de la relation de transition. Deux techniques y sont proposées. La première technique, basée sur la théorie des automates, consiste à considérer la relation d'évolution du système comme un transducteur $\mathcal{T}$, à construire une matrice qui regroupe

tous les chemins possibles de $\mathcal{T}$, à considérer cette matrice comme un unique transducteur $\mathcal{T}^*$ (dont chaque colonne est un état) que l'on va minimiser et déterminiser. Mais cette technique n'est applicable qu'à certains protocoles réguliers, car pour pouvoir construire la matrice qui regroupe tous les chemins possibles de $\mathcal{T}$, il faut que celui-ci ne contienne aucune boucle. La deuxième technique consiste à deviner automatiquement l'ensemble obtenu après application d'une relation d'évolution T, à partir d'un ensemble régulier donné, puis à vérifier si celui-ci est correct. Pour deviner cet ensemble, on compare les ensembles ϕ et $T(\phi)$ afin de détecter une éventuelle croissance (par exemple $T(\phi) = \phi.\Lambda$ pour un Λ régulier quelconque), puis à extrapoler en faisant l'hypothèse que chaque application de T aura le même effet. On fait ensuite un test de point fixe pour s'assurer que l'ensemble obtenu est bien une sur-approximation de $T^*(\phi)$. D'après l'article, cette technique a été appliquée sur les protocoles d'exclusion mutuelle de Szymanski, Burns, Dijkstra, sur le protocole de Bakery de Lamport et sur le Token passing protocol, mais seule l'application au Token passing protocol est présentée. Cette technique n'est pas non plus applicable à tous les protocoles réguliers : la relation T doit préserver la taille des mots et doit être noethérienne pour s'appliquer à un mot qu'un nombre fini de fois.

Dans [AJNd03], les auteurs proposent des améliorations de ces techniques qui réduisent la taille des automates intermédiaires, lors du calcul de la clôture transitive. C'est dans la même conférence que Sébastien Bardin, Alain Finkel, Jérôme Leroux et Laure Petrucci proposent leur outil de vérification $FAST$ (pour Fast Acceleration of Symbolic Transition systems) [BFLP03]. L'outil $FAST$ tente de calculer $\mathcal{R}^*_{\mathcal{T}}(\mathrm{I})$ – c'est-à-dire l'ensemble exact des configurations accessibles – en se basant sur des accélérations du transducteur $\mathcal{T}$.

Dans [BHV04], les auteurs proposent une technique de vérification des systèmes infinis paramétrés qui combine du model-checking régulier et de la vérification par abstraction. D'après cet article, l'explosion des états est un des plus gros problèmes du model-checking régulier, et une des sources de ce problème vient de la nature même des techniques de model-checking régulier de l'époque (2004). Ces techniques tentent de calculer l'ensemble d'accessibilité exact. Il propose donc de remplacer les techniques d'accélération précises par un calcul d'abstraction qui, lorsqu'un point fixe sera atteint, représentera une sur-approximation de l'ensemble des états atteignables. L'article propose une abstraction par fusion d'états, qui est assez proche de la technique qu'on a utilisée. Il présente également la possibilité de raffiner l'abstraction (ce qui, dans notre méthode, revient à durcir le critère de fusion avec un ET logique), au cas où l'approximation contiendrait un état d'erreur. Mais ils ne peuvent raffiner l'abstraction que si la relation $\sim$ sur les états est de la forme suivante : soit A un automate, p et q deux états de A, et $\mathcal{P}$ un ensemble fini d'automates, $p \sim^A q$ si et seulement si quelque soit P de $\mathcal{P}$, $L(P) \bigcap L(A^{Right,p}) \neq \emptyset$ si et seulement si $L(P) \bigcap L(A^{Right,q}) \neq \emptyset$ (les automates $A^{Right,p}$ et $A^{Right,q}$ sont définis dans ce mémoire, dans la section 4.2). Pour raffiner l'abstraction avec des relations de cette forme, l'article propose d'ajouter des automates dans $\mathcal{P}$. Cette technique est applicable sur une grande variété de systèmes, mais ne laisse pas de liberté concernant la forme des critères de fusion (on peut seulement décider du contenu de $\mathcal{P}$).

Dans [LGJJ06], les auteurs s'intéressent à l'analyse et la vérification de systèmes communiquants par des files de communication non bornées. Ils proposent une méthode d'analyse et de vérification de systèmes communiquants (ce sont des systèmes formés de processus séquentiels, communiquant par des files de communication non bornées), qui consiste à travailler sur des abstractions des systèmes en question. Dans le cadre de cet article, les processus sont d'états finis, et l'alphabet des messages échangés est également fini. Les protocoles étudiés sont le protocole de Connexion/Déconnexion, le Buffer infini, et l'Alternating Bit Protocol, et ils sont représentés par des systèmes de transitions étiquetés. Les auteurs proposent un treillis abstrait fondé sur les langages réguliers, afin d'étudier les systèmes avec une seule file de communication, puis ils généralisent leur proposition aux systèmes avec plusieurs files. Récemment, de

nouveaux résultats en model-checking régulier ont été obtenus pour des protocoles spécifiques comme les CLP [FPPS11], les systèmes communiquants par des files [LGJJ06], les langages d'arbres [AJMd02, BT11], ou la vérification de chaînes relationnelles utilisant des automates multi-pistes [YBI11]), en utilisant des techniques spécifiques aux catégories de systèmes données [Boi12].

Approximations régulières

De nombreuses méthodes de vérification proposées jusqu'ici sont des méthodes exactes, or il est souvent difficile, voire impossible, de trouver un point fixe de façon exacte, et donc de conclure. Puisque la valeur exacte de l'ensemble des états atteignables par le système – noté $\mathcal{R}^*(\mathrm{I})$ (*cf.* figure 3.1) – n'est pas forcément calculable, nous utilisons l'approche par approximations régulières. Cette approche est très utilisée pour attaquer le problème du model-checking régulier, et consiste à calculer un langage régulier K, qui contient $\mathcal{R}^*(\mathrm{I})$. On dit alors que K est une *sur-approximation* de $\mathcal{R}^*(\mathrm{I})$.

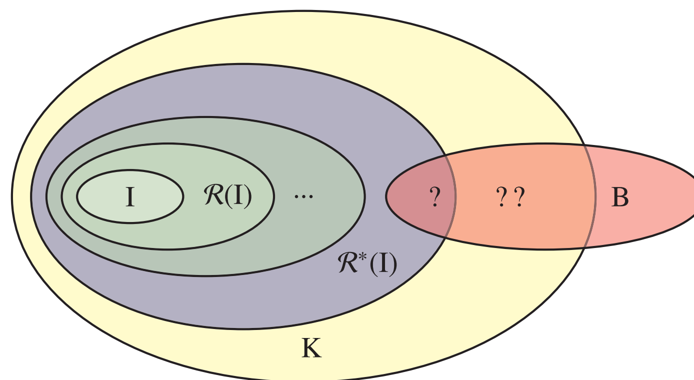


FIGURE 3.1 – Approche par sur-approximation

L'enjeu est de démontrer que $\mathcal{R}^*(\mathrm{I}) \bigcap \mathrm{B} = \emptyset$. Si on a $\mathrm{K} \bigcap \mathrm{B} = \emptyset$, on peut l'affirmer. Mais si après calcul, $\mathrm{K} \bigcap \mathrm{B} \neq \emptyset$, alors on ne peut rien déduire sur $\mathcal{R}^*(\mathrm{I}) \bigcap \mathrm{B}$. On cherchera donc une sur-approximation moins grossière de $\mathcal{R}^*(\mathrm{I})$.

La difficulté – et l'objectif même de cette partie de la thèse – est de trouver une façon automatique de calculer un ensemble K, qui permette de conclure le plus souvent possible (chaque fois que possible serait l'idéal). Or, les résultats d'une méthode de calcul par rapport à une autre diffèrent beaucoup selon le protocole qu'on souhaite vérifier.

3.2/ Technique proposée

Nos contributions visent à améliorer la méthode générique de [BHV04] en donnant des moyens de construire des sur-approximations en fusionnant des états abstraits du système (et non du transducteur, qui n'est jamais modifié).

Notre méthode utilise une approche par sur-approximations successives, qui consiste à calculer $\mathcal{R}_{\mathcal{T}}(\mathrm{I})$ puis à en calculer une sur-approximation I_1, à calculer ensuite $\mathcal{R}_{\mathcal{T}}(\mathrm{I}_1)$ puis à en calculer une sur-approximation I_2, ... et ainsi de suite jusqu'à atteindre un point fixe, qui sera notre sur-approximation K de $\mathcal{R}^*_{\mathcal{T}}(\mathrm{I})$ (*cf.* figure 3.2).

Malgré le fait que l'approche par sur-approximations successives augmente les chances de trouver un point fixe, on peut avoir besoin d'appliquer le transducteur de nombreuses fois avant d'en trouver un, ce qui multiplie à chaque fois le nombre d'états de l'automate obtenu. En fusionnant

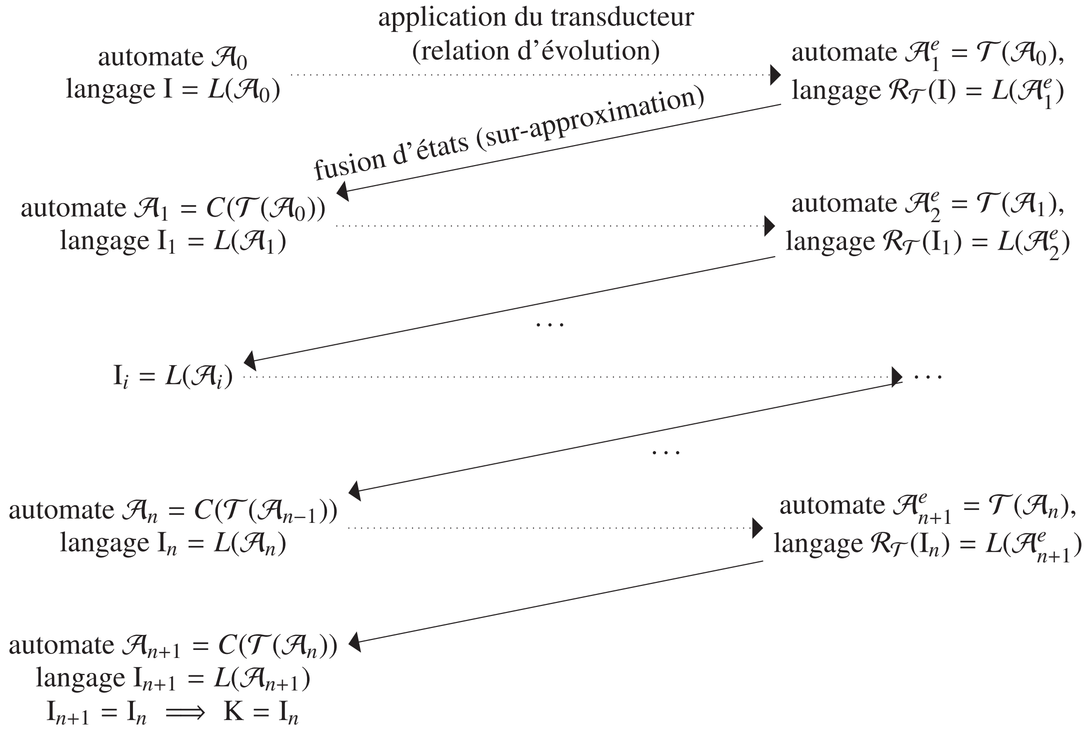


FIGURE 3.2 – Sur-approximations successives

des états entre eux, la sur-approximation effectuée à chaque étape (flèches en diagonale sur la figure 3.2) doit donc également empêcher l'explosion du nombre d'états.

Afin de déterminer les états à fusionner, on définit un ou plusieurs critères de fusion, qui sont les points communs que doivent avoir deux états pour être fusionnés. Ces critères sont définis sous la forme de relations entre les états (deux états seront fusionnés s'ils sont en relation), qui sont elles-même obtenues à partir de fonctions d'approximations (*cf.* définition 2.3). Le choix de bons critères de fusion, et donc de bonnes fonctions d'approximation, est extrêmement important. Si le critère est trop large l'approximation reconnaîtra des mots de B (la propriété d'erreur) avant d'atteindre un point fixe, et s'il est trop restrictif, on ne fusionnera pas d'états et la taille de l'automate risque d'exploser. Il faut également que les critères soient rapidement calculables, et qu'ils puissent s'appliquer à tous types de problèmes réguliers (*cf.* définition 2.2).

3.3/ Calcul de la sur-approximation

Dans cette section, on explique comment calculer une sur-approximation K de $\mathcal{R}_\mathcal{T}^*(\mathrm{I})$ (*cf.* figure 3.2), à partir d'une fonction d'approximation $\mathfrak{F}$.

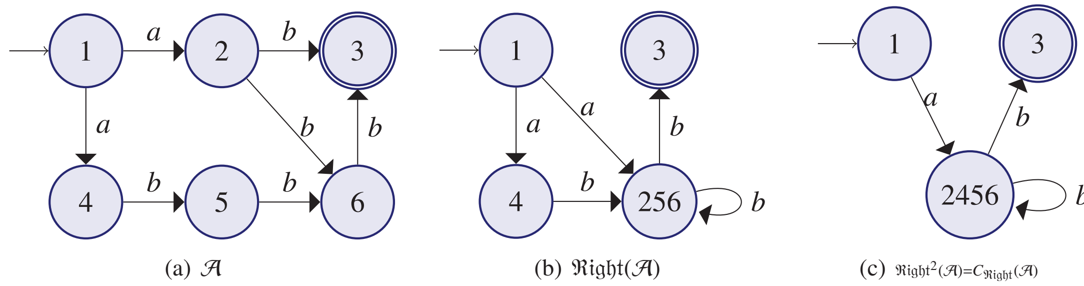
Proposition 3.4 *Pour tout automate $\mathcal{A}$, si $\mathfrak{F}$ est une fonction d'approximation isomorphisme-compatible, alors la séquence $(\mathfrak{F}^n(\mathcal{A}))_{n\in\mathbb{N}}$ est ultimement constante, à isomorphisme près. Notons $C_\mathfrak{F}(\mathcal{A})$ la limite de $(\mathfrak{F}^n(\mathcal{A}))_{n\in\mathbb{N}}$. Si pour tout automate $\mathcal{A}$ et toute paire d'états p, q de $\mathcal{A}$, on peut vérifier en temps polynomial si $p \sim_\mathfrak{F}^\mathcal{A} q$, alors $C_\mathfrak{F}(\mathcal{A})$ peut également être calculé en temps polynomial.*

PREUVE. Soit $\mathcal{A}$ un automate et $\sim$ une relation d'équivalence sur les états de $\mathcal{A}$. Si $\sim$ n'est pas l'identité, alors $\mathcal{A}/_\sim$ a strictement moins d'états que $\mathcal{A}$. Sinon, $\mathcal{A}/_\sim$ est isomorphe à $\mathcal{A}$. C'est pourquoi, dans la séquence $(\mathfrak{F}^n(\mathcal{A}))_{n\in\mathbb{N}}$, le nombre d'états des automates diminue et si $\mathfrak{F}^k(\mathcal{A})$ et

$\mathfrak{F}^{k+1}(\mathcal{A})$ ont le même nombre d'états, alors $(\mathfrak{F}^n(\mathcal{A}))_{n \geq k}$, est finalement constant, à l'isomorphisme près.

Si l'équivalence des états est calculée en temps polynomial, alors l'automate quotient peut également être calculé en temps polynomial : il suffit de calculer les composantes fortement connexes, du graphe de la relation $\sim$. De plus, puisque la convergence est obtenue en au plus n états, où n est le nombre d'états de $\mathcal{A}$, $C_{\mathfrak{F}}(\mathcal{A})$ peut être calculé en temps polynomial. □

Exemple 3.5. *La figure 3.3 représente un automate $\mathcal{A}$, ainsi que le calcul de $C_{\mathfrak{Right}}(\mathcal{A})$ (la relation $\mathfrak{Right}$ est définie plus tard, dans la section 4.2).*



(a) $\mathcal{A}$ (b) $\mathfrak{Right}(\mathcal{A})$ (c) $\mathfrak{Right}^2(\mathcal{A})=C_{\mathfrak{Right}}(\mathcal{A})$

FIGURE 3.3 – Calcul de $C_{\mathfrak{Right}}(\mathcal{A})$

Semi-Algorithme `PointFixe`
Entrée : $\mathcal{A}, \mathcal{T}, \mathcal{B}, \mathfrak{F}$
  Si $L(C_{\mathfrak{F}}(\mathcal{T}(\mathcal{A}))) \cap L(\mathcal{B}) \neq \emptyset$ **alors**
    retourner *Non-conclusif*
  FinSi
  Si $L(C_{\mathfrak{F}}(\mathcal{T}(\mathcal{A}))) = L(\mathcal{A})$ **alors**
    retourner *Sûr*
  FinSi
  Retourner `PointFixe`$(C_{\mathfrak{F}}(\mathcal{T}(\mathcal{A})), \mathcal{T}, \mathcal{B}, \mathfrak{F})$

FIGURE 3.4 – Semi-algorithme `PointFixe`

Dans le semi-algorithme `PointFixe` (*cf.* figure 3.4), soit un automate $\mathcal{A}$ (état du système), un transducteur $\mathcal{T}$ (relation de transition), un automate $\mathcal{B}$ (propriété d'erreur), et une fonction d'approximation isomorphisme-compatible $\mathfrak{F}$ (critère de fusion), la première vérification (le vide) peut être effectuée en temps polynomial. Malheureusement, l'égalité des langages ne peut pas être vérifiée en temps polynomial (les automates impliquées ne sont pas déterministes). Cependant, des algorithmes récemment développés [DR10, ACH+10, BP12] permettent de résoudre le problème du test de l'inclusion de langages (et donc de l'égalité) très efficacement. Notons également que le test d'égalité peut être remplacé par un autre test – comme l'isomorphisme ou encore la (bi)simulation – qui implique l'égalité ou l'inclusion des langages, puisque par construction $L(\mathcal{A}) \subseteq L(C_{\mathfrak{F}}(\mathcal{T}(\mathcal{A})))$. Dans ce cas le test serait plus rapide, mais la convergence moins rapide.

Proposition 3.6 *Le semi-algorithme* `PointFixe` *est correct : s'il retourne* Sûr, *alors* $R^*_{\mathcal{T}}(L(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset$.

PREUVE. Si le semi-algorithme `PointFixe` retourne *Sûr*, alors on a calculé un automate $\mathcal{A}'$ tel que $L(\mathcal{A}') = L(C_{\mathfrak{F}}(\mathcal{T}(\mathcal{A}')))$. On prouve tout d'abord que $R_{\mathcal{T}}(L(\mathcal{A}')) = L(\mathcal{A}')$: puisque l'identité est inclue dans $R_{\mathcal{T}}$, il suffit de prouver que $R_{\mathcal{T}}(L(\mathcal{A}')) \subseteq L(\mathcal{A}')$. Si $u \in R_{\mathcal{T}}(L(\mathcal{A}'))$, alors

$u \in L(\mathcal{T}(\mathcal{A}'))$. C'est pourquoi $u \in L(C_{\mathfrak{F}}(\mathcal{T}(\mathcal{A}'))) = L(\mathcal{A}')$. Par une induction directe, on a $R^n_{\mathcal{T}}(L(\mathcal{A}')) = L(\mathcal{A}')$ pour tout $n \geq 0$. C'est pourquoi $R^*_{\mathcal{T}}(L(\mathcal{A}')) = L(\mathcal{A}')$.

De plus, puisque $L(\mathcal{A}) \subseteq L(C_{\mathfrak{F}}(\mathcal{A}))$, on a $L(\mathcal{T}(\mathcal{A})) \subseteq L(\mathcal{T}(C_{\mathfrak{F}}(\mathcal{A})))$. C'est pourquoi, par induction directe, $L(\mathcal{A}) \subseteq L(\mathcal{A}')$. Il s'en suit que $R^*_{\mathcal{T}}(L(\mathcal{A})) \subseteq L(\mathcal{A}') = R^*_{\mathcal{T}}(L(\mathcal{A}'))$. Par conséquent, $R^*_{\mathcal{T}}(L(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset$. □

Exemple 3.7. *La figure 3.5 illustre cette approche, appliquée au Token ring et à l'aide de la fonction d'approximation* $\mathfrak{Left}$ *(définie plus tard, dans la section 4.2). On peut vérifier que* $C_{\mathfrak{Left}}(\mathcal{T}_{tr}(C_{\mathfrak{Left}}(\mathcal{T}_{tr}(\mathcal{A}_{tr}))))$ *et* $C_{\mathfrak{Left}}(\mathcal{T}_{tr}(\mathcal{A}_{tr}))$ *sont isomorphes. C'est pourquoi le semi-algorithme* `PointFixe` *s'arrête après un appel récursif et retourne Sûr.*

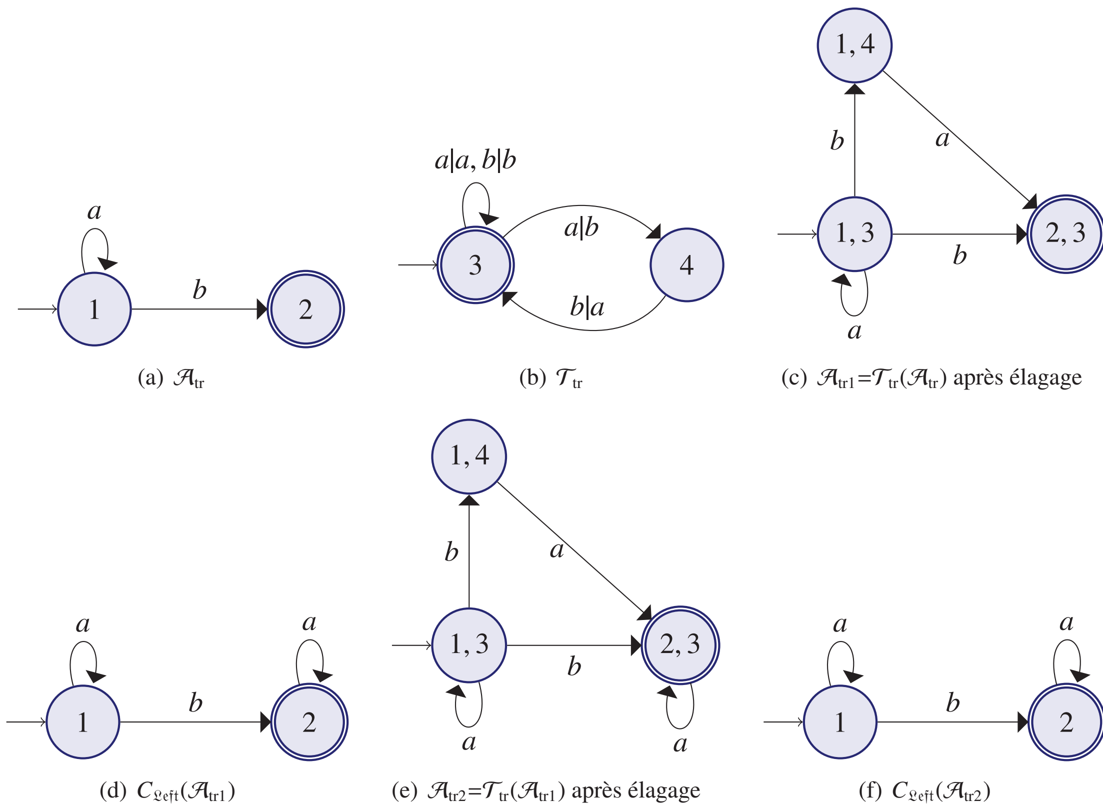


FIGURE 3.5 – Token ring : semi-algorithme `PointFixe` avec $\mathfrak{F} = \mathfrak{Left}$

3.4/ RAFFINEMENT D'APPROXIMATIONS

Rappelons que l'on s'intéresse au problème d'analyse d'accessibilité par model-checking régulier, et plus particulièrement à sa représentation RMC-2 (*cf.* problème 3.3). Le calcul de la sur-approximation de l'ensemble des états atteignables par le système, tel que présenté ci-avant est certes automatique mais non adaptatif : que faire si l'approximation est trop grossière ? Comment en calculer une plus fine ? Pour répondre à ces questions, nous nous sommes appuyés sur le concept de *raffinement d'abstraction à partir de contre-exemples* – abrégé en *CEGAR*, pour l'anglais Counter Example Guided Abstraction Refinement, ce paradigme est présenté dans [CGJ+00] et a été largement étudié durant ces dix dernières années (voir par exemple [BHV04, BCHK08]) – dans le but d'obtenir des approximations/abstractions plus fines en exploitant des contre-exemples.

Lors de l'application de l'approche par approximations successives – qui consiste à appliquer la relation d'évolution et fusionner des états, puis à recommencer jusqu'à atteindre un point fixe – (*cf.* figure 3.2), il est possible qu'on obtienne un langage I_k dont l'intersection avec la propriété d'erreur $\mathcal{B}$ n'est pas vide. Dans cette situation, il est inutile de continuer jusqu'à atteindre un point fixe puisque la sur-approximation K de $\mathcal{R}^*_{\mathcal{T}}(I)$ que l'on va calculer aura forcément une intersection non vide avec la propriété d'erreur $\mathcal{B}$, ce qui ne nous permettra pas de conclure (*cf.* figure 3.1). Dans ce cas, on effectue donc une analyse en arrière.

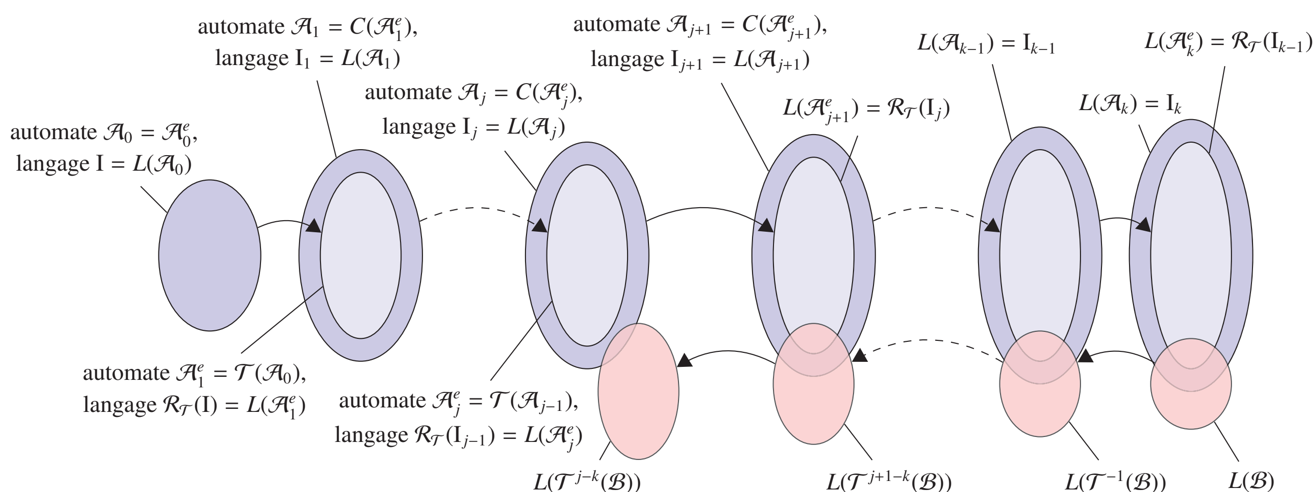


FIGURE 3.6 – Situation nécessitant un raffinement de l'approximation

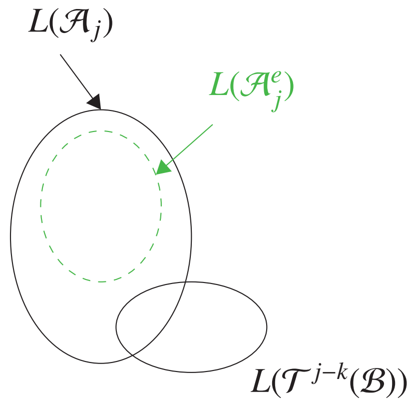
Algorithme `Analyse en arrière`
Entrée : La liste des automates $\mathcal{A}^e_i$ et $\mathcal{A}_i$ calculés avec $0 \leq i \leq k$, l'automate $\mathcal{B}$ qui représente la propriété d'erreur, et le transducteur $\mathcal{T}$ qui représente la relation d'évolution du système
  $n := 1$
  Tant que $L(\mathcal{T}^{-n}(\mathcal{B})) \cap L(\mathcal{A}^e_{k-n}) \neq \emptyset$ et $n \leq k$ **faire**
    $n := n + 1$
  FinTantQue
  Si $n > k$ alors
    Retourner *Non sûr*
  Sinon
    $j := k - n$
    Retourner La liste des automates $\mathcal{A}^e_i$ et $\mathcal{A}_i$ calculés avec $0 \leq i \leq j$, l'automate $\mathcal{T}^{-n}(\mathcal{B})$ qui représente la nouvelle propriété d'erreur, et le transducteur $\mathcal{T}$ qui représente la relation d'évolution du système
  FinSi

FIGURE 3.7 – Algorithme `Analyse en arrière`

L'algorithme d'analyse en arrière, qui est illustré par la figure 3.6 et détaillé dans la figure 3.7, sert à déterminer lors de quelle étape de sur-approximation – passage d'un automate $\mathcal{A}^e_i$ à un automate $\mathcal{A}_i$ dans les figures 3.2 et 3.6 – des fusions d'états ont permis d'atteindre la propriété d'erreur. Son principe est de calculer une nouvelle propriété d'erreur $L(\mathcal{T}^{-i}(\mathcal{B}))$, en incrémentant i tant que $L(\mathcal{T}^{-i}(\mathcal{B})) \bigcap L(\mathcal{A}^e_{k-i})$ n'est pas vide. Si l'algorithme d'analyse en arrière trouve une valeur de n telle que l'intersection entre $L(\mathcal{T}^{-n}(\mathcal{B}))$ et $L(\mathcal{A}_0)$ n'est pas vide, cela signifie que le système peut atteindre un état de $\mathcal{B}$, donc que $\mathcal{R}^*(I) \bigcap B \neq \emptyset$.

L'algorithme retourne donc *Non sûr*. Sinon, l'algorithme retourne la situation à partir de laquelle il va falloir raffiner l'approximation, afin d'empêcher les fusions d'états qui permettent d'atteindre la propriété d'erreur (*cf.* figure 3.8).

On raffine dans la situation suivante :



Afin d'obtenir la situation suivante :

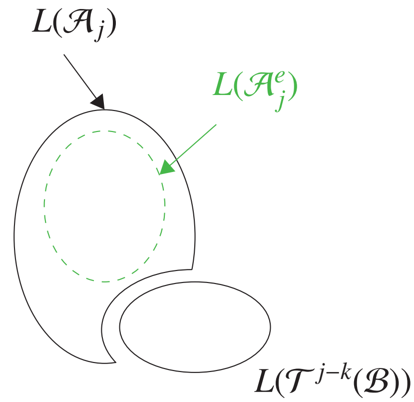


FIGURE 3.8 – Principe du raffinement

Dans le chapitre 4, nous définissons des fonctions d'approximations qui produisent des critères de fusions syntaxiques, puis nous les expérimentons sur quelques exemples courants. Ces fonctions d'approximations sont à la fois isomorphisme-compatibles, rapides à calculer, et calculables quelle que soit la forme de l'automate. Dans le chapitre 5, pour chaque état de chaque automate $\mathcal{A}_i$ ou $\mathcal{A}_i^e$ (*cf.* figures 3.2 et 3.6), nous mémorisons les états du transducteur utilisés lors de chaque étape d'application de la relation d'évolution – passage d'un automate $\mathcal{A}_i$ à un automate $\mathcal{A}_{i+1}^e$ dans les figures 3.2 et 3.6 – . Nous définissons des fonctions d'approximations qui exploitent ces listes d'états du transducteur, puis nous les expérimentons sur les mêmes exemples que ceux utilisés dans le chapitre 4.

4

Contribution : Critères de fusion syntaxiques

Dans ce chapitre, nous définissons des fonctions d'approximations qui exploitent des critères syntaxiques. Ces fonctions d'approximations produisent des critères qui nous servent à déterminer les états à fusionner lors des étapes de sur-approximation – flèches pleines en diagonale dans la figures 3.2 – afin de tenter de résoudre le problème du RMC, tel que défini dans le chapitre 3 (problème 3.3). Les critères de fusion syntaxiques obtenus se veulent généraux et rapides à calculer. Afin d'améliorer les chances de calculer une sur-approximation K de $\mathcal{R}^*_{\mathcal{T}}(\mathrm{I})$ dont l'intersection avec la propriété d'erreur $\mathcal{B}$ est vide, nous agençons ces critères de fusion entre eux avec des ET logiques et/ou des OU logiques :

— Si lors de l'approche par sur-approximations successives (*cf.* figure 3.2), on obtient un langage I_k dont l'intersection avec la propriété d'erreur $\mathcal{B}$ n'est pas vide, on peut obtenir un critère plus restrictif que celui utilisé en l'agençant avec un autre à l'aide d'un ET logique, et ainsi fusionner moins d'états.

— Si au contraire l'intersection avec la propriété d'erreur reste vide, mais que l'on ne parvient pas à trouver un point fixe, on peut élargir le critère utilisé en l'agençant avec un autre à l'aide d'un OU logique, et ainsi fusionner davantage d'états.

4.1/ $\mathfrak{In}$ et $\mathfrak{Out}$

Les deux premières fonctions d'approximations que nous avons imaginées sont la fonction $\mathfrak{In}$, qui permet de comparer les labels des transitions qui arrivent sur chacun des états, et la fonction $\mathfrak{Out}$, qui permet de comparer les labels des transitions qui sortent de chacun des états.

Soit $\mathcal{A} = (Q, \Sigma, E, I, F)$ un automate, et p et q deux états quelconques de $\mathcal{A}$, nous commençons par définir les sous-ensembles $In(q)$ et $Out(q)$:

— Soit a une lettre de Σ, a appartient à $In(q)$ si et seulement s'il existe un état q_1 de $\mathcal{A}$ tel que (q_1, a, q) est une transition de $\mathcal{A}$.

— Soit a une lettre de Σ, a appartient à $Out(q)$ si et seulement s'il existe un état q_1 de $\mathcal{A}$ tel que $(q, a, q1)$ est une transition de $\mathcal{A}$.

Autrement dit, $In(q)$ représente l'ensemble des étiquettes des transitions qui arrivent sur l'état q, et $Out(q)$ représente l'ensemble des étiquettes des transitions qui partent de l'état q.

Ce qui nous permet de définir les fonctions d'approximation $\mathfrak{In}$ et $\mathfrak{Out}$, de la façon suivante :

— La fonction d'approximation $\mathfrak{In}$ associe à tout automate $\mathcal{A}$ la relation d'équivalence $\sim^{\mathcal{A}}_{\mathfrak{In}}$, définie par $p \sim^{\mathcal{A}}_{\mathfrak{In}} q$ si et seulement si $In(p) = In(q)$.

Exemple 4.1. $\mathfrak{In}$ *Pour l'automate $\mathcal{A}_{ex}$, on a $In(1) = \{a\}$, $In(2) = \{b\}$, $In(3) = \{b\}$ et $In(4) =$*

$\{a,c\}$. *Comme* $In(2) = In(3)$, *on fusionne l'état 2 et l'état 3 pour obtenir l'automate* $\mathfrak{In}(\mathcal{A}_{ex})$ *(cf. figure 4.1).*

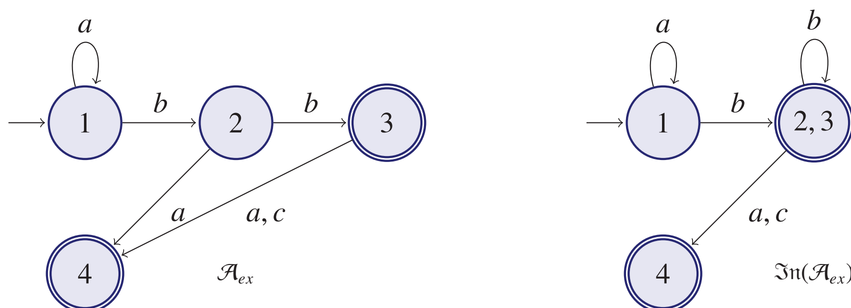


FIGURE 4.1 – $\mathfrak{In}(\mathcal{A}_{ex})$

— La fonction d'approximation $\mathfrak{Out}$ associe à tout automate $\mathcal{A}$ la relation d'équivalence $\sim^{\mathcal{A}}_{\mathfrak{Out}}$, définie par $p \sim^{\mathcal{A}}_{\mathfrak{Out}} q$ si et seulement si $Out(p) = Out(q)$.

Exemple 4.2. $\mathfrak{Out}$ *Pour l'automate* $\mathcal{A}_{ex}$, *on a* $Out(1) = \{a,b\}$, $Out(2) = \{a,b\}$, $Out(3) = \{a,c\}$ *et* $Out(4) = \{\}$. *Comme* $Out(1) = Out(2)$, *on fusionne l'état 1 et l'état 2 pour obtenir l'automate* $\mathfrak{Out}(\mathcal{A}_{ex})$ *(cf. figure 4.2).*

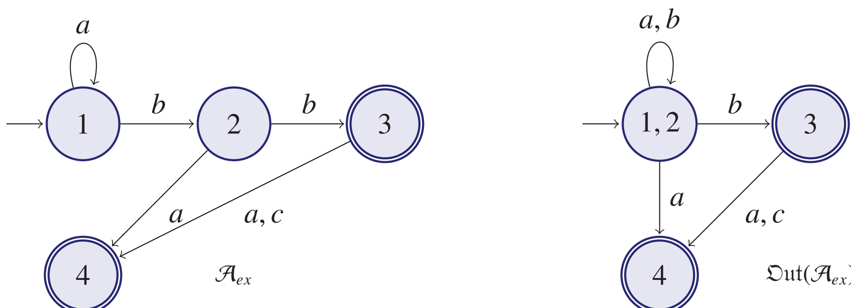


FIGURE 4.2 – $\mathfrak{Out}(\mathcal{A}_{ex})$

4.2/ $\mathfrak{Left}$ ET $\mathfrak{Right}$

Nous allons définir ici les fonctions d'approximation $\mathfrak{Left}$ et $\mathfrak{Right}$, qui représentent des critères simples et naturellement utilisés, comme par exemple dans [BHV04] pour calculer des relations d'équivalence, ou dans [BF12] pour surveiller des propriétés LTL.

Soit $\mathcal{A} = (Q, \Sigma, E, I, F)$ un automate, et p et q deux états quelconques de $\mathcal{A}$.

Le langage de $\mathcal{A}$ à gauche de q est le langage composé de l'ensemble des étiquettes de tous les chemins de $\mathcal{A}$, qui commencent par un état initial et se terminent par l'état q. L'automate $(Q, \Sigma, E, I, \{q\})$ qui reconnaît ce langage est noté $\mathcal{A}^{Left,q}$.

Exemple 4.3. $\mathcal{A}^{Left,2}_{ex}$ *L'automate* $\mathcal{A}_{ex}$ *reconnaît les mots de la forme* a^*ba, a^*bb, a^*bba *et* a^*bbc, *l'automate* $\mathcal{A}^{Left,2}_{ex}$ *reconnaît donc les mots de la forme* a^*b *(cf. figure 4.3).*

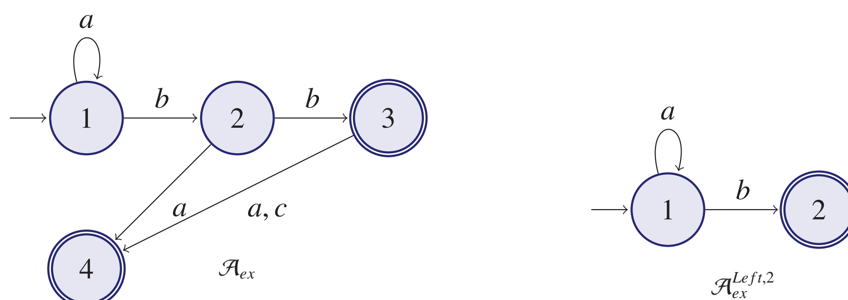


FIGURE 4.3 – $\mathcal{A}_{ex}^{Left,2}$

De façon similaire, le langage de $\mathcal{A}$ à droite de q est le langage composé de l'ensemble des étiquettes de tous les chemins de $\mathcal{A}$, qui commencent par l'état q et se terminent par un état final. L'automate $(Q, \Sigma, E, \{q\}, F)$ qui reconnaît ce langage est noté $\mathcal{A}^{Right,q}$.

Exemple 4.4. $\boldsymbol{\mathcal{A}_{ex}^{Right,3}}$ *L'automate $\mathcal{A}_{ex}$ reconnaît les mots de la forme a^*ba, a^*bb, a^*bba et a^*bbc, l'automate $\mathcal{A}_{ex}^{Right,3}$ reconnaît donc les mots a et c, ainsi que le mot vide (cf. figure 4.4).*

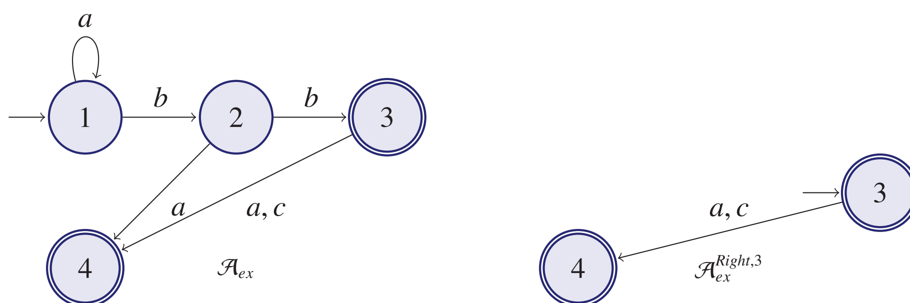
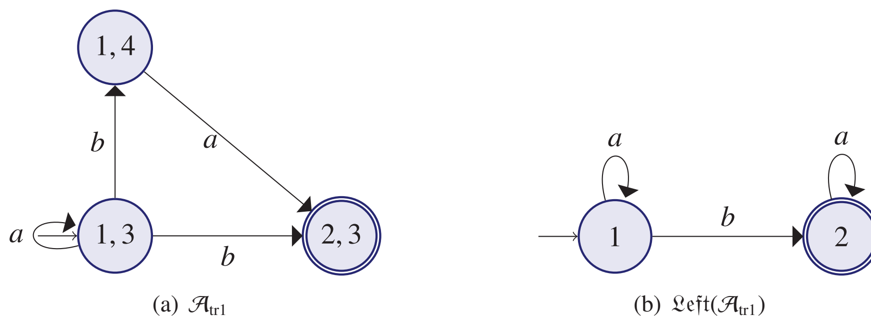


FIGURE 4.4 – $\mathcal{A}_{ex}^{Right,3}$

Cela nous permet de définir les fonctions d'approximation suivantes :

— La fonction d'approximation $\mathfrak{Left}$ associe à tout automate $\mathcal{A}$ la clôture réflexive et transitive de la relation $\sim_{\mathfrak{Left}}^{\mathcal{A}}$, définie par $p \sim_{\mathfrak{Left}}^{\mathcal{A}} q$ si et seulement si $L(\mathcal{A}^{Left,p}) \cap L(\mathcal{A}^{Left,q}) \neq \emptyset$.

Exemple 4.5. $\boldsymbol{\mathfrak{Left}(\mathcal{A}_{tr1})}$ *Pour l'exemple du protocole Token Ring – décrit dans la figure 2.12 – soit $\mathcal{A}_{tr1}$ l'automate obtenu en élaguant $\mathcal{T}_{tr}(\mathcal{A}_{tr})$. Pour la relation $\sim_{\mathfrak{Left}}^{\mathcal{A}_{tr1}}$, les états $(1,4)$ et $(2,3)$ sont équivalents puisqu'ils peuvent être atteints en lisant le mot b à partir d'un état initial. L'automate $\mathfrak{Left}(\mathcal{A}_{tr1})$ est décrit dans la figure 4.5(b).*



(a) $\mathcal{A}_{\text{tr1}}$ (b) $\mathfrak{Left}(\mathcal{A}_{\text{tr1}})$

FIGURE 4.5 – $\mathfrak{Left}(\mathcal{A}_{\text{tr1}})$

— La fonction d'approximation $\mathfrak{Right}$ associe à tout automate $\mathcal{A}$ la clôture réflexive et transitive de la relation $\sim^{\mathcal{A}}_{\mathfrak{Right}}$, définie par $p \sim^{\mathcal{A}}_{\mathfrak{Right}} q$ si et seulement si $L(\mathcal{A}^{Right,p}) \cap L(\mathcal{A}^{Right,q}) \neq \emptyset$.

Exemple 4.6. $\mathfrak{Right}(\mathcal{A}_{ex})$ *Pour l'automate $\mathcal{A}_{ex}$, $\mathcal{A}_{ex}^{Right,1} \times \mathcal{A}_{ex}^{Right,2}$ reconnaît le mot ba, $\mathcal{A}_{ex}^{Right,2} \times \mathcal{A}_{ex}^{Right,3}$ reconnaît le mot a, $\mathcal{A}_{ex}^{Right,3} \times \mathcal{A}_{ex}^{Right,4}$ reconnaît le mot vide (ε), et les autres produits d'automates ne reconnaissent aucun mot. On va donc fusionner les états 1, 2, 3 et 4 de $\mathcal{A}_{ex}$ pour obtenir l'automate $\mathfrak{Right}(\mathcal{A}_{ex})$ (cf. figure 4.6).*

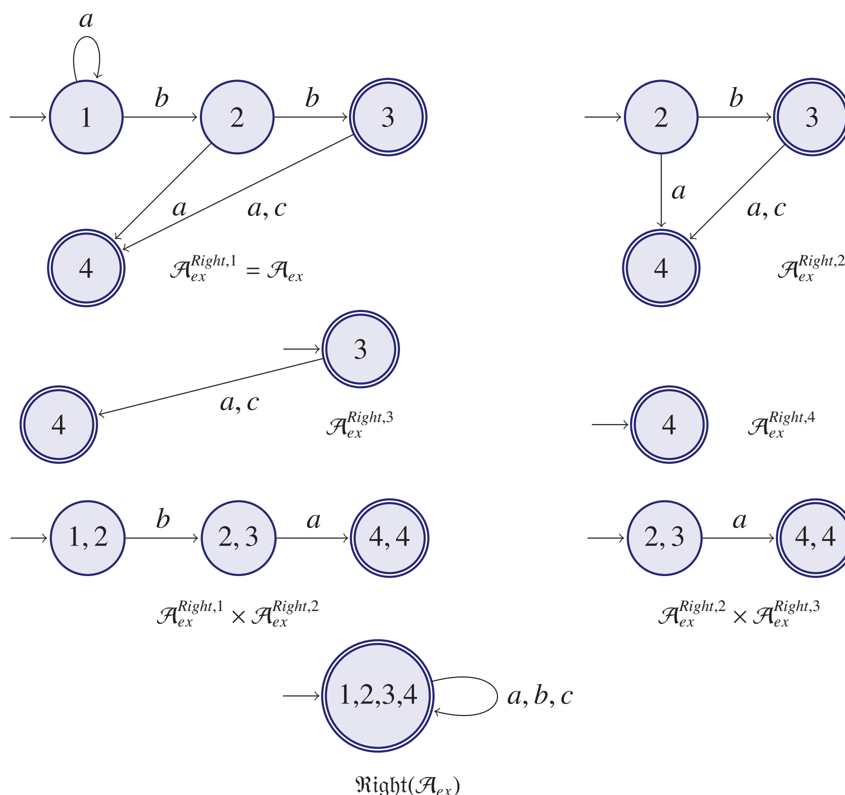


FIGURE 4.6 – $\mathfrak{Right}(\mathcal{A}_{ex})$

L'algorithmique des automates nous permet de vérifier facilement si p et q sont en relation étudiée, en procédant de la façon suivante : pour construire l'automate $\mathcal{A}^{Left,p}$ (resp. $\mathcal{A}^{Right,p}$) à partir

de $\mathcal{A}$, il suffit de déclarer l'état p comme unique état final (resp. état initial) (même chose avec l'état q pour construire $\mathcal{A}^{Left,q}$ ou $\mathcal{A}^{Right,q}$). Il suffit ensuite de calculer le produit des automates ($\mathcal{A}^{Left,p} \times \mathcal{A}^{Left,q}$ pour le critère $\mathfrak{Left}$, et $\mathcal{A}^{Right,p} \times \mathcal{A}^{Right,q}$ pour le critère $\mathfrak{Right}$). Comme le produit des deux automates reconnaît l'intersection de leurs langages, p et q sont en relation si ce produit reconnaît au moins un mot, et la vacuité est également très simple à vérifier.

4.3/ Combinaisons de fonctions d'approximation

Soit deux fonctions d'approximation $\mathfrak{F}$ et $\mathfrak{G}$, on note $\mathfrak{F}.\mathfrak{G}$ la fonction d'approximation qui associe à tout automate $\mathcal{A}$ la relation $\sim^{\mathcal{A}}_{\mathfrak{F}.\mathfrak{G}} = \sim^{\mathcal{A}}_{\mathfrak{F}} \cap \sim^{\mathcal{A}}_{\mathfrak{G}}$. On définit également la fonction d'approximation $\mathfrak{F}+\mathfrak{G}$ qui associe à tout automate $\mathcal{A}$, la relation $\sim^{\mathcal{A}}_{\mathfrak{F}+\mathfrak{G}}$ qui est la plus petite relation d'équivalence contenant à la fois $\sim^{\mathcal{A}}_{\mathfrak{F}}$ et $\sim^{\mathcal{A}}_{\mathfrak{G}}$. Ces deux opérateurs nous permettent de combiner des fonctions d'approximations, pour obtenir de nouvelles – plus strictes ou plus larges – approximations.

Exemple 4.7. ($\mathfrak{Left}+\mathfrak{Right}$).($\mathfrak{In}+\mathfrak{Out}$) *À tout automate $\mathcal{A}$, la fonction d'approximation* ($\mathfrak{Left}+\mathfrak{Right}$).($\mathfrak{In}+\mathfrak{Out}$) *associe la relation $\sim^{\mathcal{A}}_{(\mathfrak{Left}+\mathfrak{Right}).(\mathfrak{In}+\mathfrak{Out})}$ telle que, soient p et q deux états de $\mathcal{A}$, $p \sim^{\mathcal{A}}_{(\mathfrak{Left}+\mathfrak{Right}).(\mathfrak{In}+\mathfrak{Out})} q$ si et seulement si $(p \sim^{\mathcal{A}}_{\mathfrak{Left}} q \vee p \sim^{\mathcal{A}}_{\mathfrak{Right}} q) \wedge (p \sim^{\mathcal{A}}_{\mathfrak{In}} q \vee p \sim^{\mathcal{A}}_{\mathfrak{Out}} q)$.*

4.4/ Expérimentations

Cette section présente les résultats expérimentaux pour les fonctions d'approximation $\mathfrak{In}$, $\mathfrak{Out}$, $\mathfrak{Left}$, $\mathfrak{Right}$, ainsi que pour les combinaisons $\mathfrak{In}+\mathfrak{Out}$, $\mathfrak{In}.\mathfrak{Out}$, $\mathfrak{Left}+\mathfrak{Right}$, $\mathfrak{Left}.\mathfrak{Right}$, et ($\mathfrak{Left}+\mathfrak{Right}$).($\mathfrak{In}+\mathfrak{Out}$).

Nous avons expérimenté ces fonctions d'approximation sur quelques exemples courants qui sont l'algorithme de la boulangerie par Lamport, l'algorithme de l'anneau à jeton (Token Ring), ainsi que les protocoles de Dijkstra et de Burns, en utilisant le codage en automates de [Tou01]. Pour cela nous avons utilisé JAMF (pour Java Automata Manipulation Framework), qui est un framework de manipulation d'automates que nous avons développé en JAVA.

Les résultats sont détaillés dans le tableau de la figure 4.7. Dans la figure 4.7, la première colonne indique le protocole étudié : son nome, la taille (c'est-à-dire $|Q| + |E|$) de l'automate initial, ainsi que la taille du transducteur. Les colonnes suivantes donnent les résultats obtenus pour chaque critère :

- la première ligne donne l'étape à laquelle a été atteinte l'égalité des langages, ou No si l'intersection avec la propriété d'erreur n'est plus vide avant d'avoir atteint l'égalité des langages, ou T.o (pour Time out) si l'algorithme a été arrêté sans avoir atteint l'égalité des langages ni trouvé d'intersection avec la propriété d'erreur,
- la seconde ligne indique l'étape à partir de laquelle l'intersection avec la propriété d'erreur n'est plus vide, ou $\emptyset$ si cette intersection reste vide,
- la troisième ligne indique la taille du dernier automate obtenu.

Si on parvient à obtenir un automate qui reconnaît le même langage que l'automate calculé à l'étape précédente, et que l'intersection entre ce langage et la propriété d'erreur est vide, alors le protocole est sûr (*cf.* valeurs en gras) : le protocole ne pourra jamais atteindre une configuration qui satisfait la propriété d'erreur étudiée.

	Critère de fusion
Algorithme Taille de l'automate initial Taille du transducteur	Équivalence des langages Intersection avec la propriété d'erreur Taille de l'automate obtenu

Résultats - Légende

	$\mathfrak{In}$	$\mathfrak{Out}$	$\mathfrak{In}$+$\mathfrak{Out}$	$\mathfrak{In}$.$\mathfrak{Out}$	$\mathfrak{Left}$	$\mathfrak{Right}$	$\mathfrak{Left}$+$\mathfrak{Right}$	$\mathfrak{Left}$.$\mathfrak{Right}$	($\mathfrak{Left}$+$\mathfrak{Right}$).($\mathfrak{In}$+$\mathfrak{Out}$)
Token ring taille I : 4 taille T : 6	**Step 3** ∅ 8	**Step 3** ∅ 8	**Step 3** ∅ 5	**Step 4** ∅ 12	**Step 3** ∅ 8	**Step 2** ∅ 5	**Step 2** ∅ 5	**Step 3** ∅ 8	**Step 3** ∅ 5
Token ring taille I : 4 taille T : 9	**Step 3** ∅ 8	**Step 3** ∅ 8	**Step 3** ∅ 5	**Step 4** ∅ 12	T.o(Step 10) ∅ 109	**Step 2** ∅ 5	**Step 2** ∅ 5	T.o(Step 10) ∅ 109	**Step 3** ∅ 5
Dijkstra taille I : 5 taille T : 62	**Step 6** ∅ 49	**Step 6** ∅ 118	**Step 5** ∅ 20	**Step 7** ∅ 246	T.o(Step 10) ∅ 745	**Step 5** ∅ 11	**Step 5** ∅ 10	T.o(115 heures) T.o(115 heures) T.o(115 heures)	**Step 5** ∅ 15
Bakery taille I : 2 taille T : 24	**Step 7** ∅ 43	No Step 7 75	No Step 6 89	**Step 10** ∅ 97	T.o(Step 10) ∅ 368	No Step 3 31	No Step 3 31	T.o(Step 10) ∅ 1253	No Step 6 101
Burns taille I : 2 taille T : 22	No Step 5 100	**Step 6** ∅ 46	No Step 3 53	No Step 7 365	No Step 4 22	No Step 3 18	No Step 3 18	No Step 4 50	No Step 3 53

FIGURE 4.7 – Résultats avec des critères syntaxiques

4.5/ CONCLUSION

Les critères de fusion présentés dans ce chapitre sont généraux et rapides à calculer, et permettent de vérifier les algorithmes sur lesquels nous les avons expérimentés (*cf.* cases en gras dans le tableau de la figure 4.7). Afin d'anticiper le cas ou aucun des critères syntaxiques définis ne permettrait de calculer une sur-approximation K de $\mathcal{R}^*_{\mathcal{T}}(I)$ dont l'intersection avec la propriété d'erreur $\mathcal{B}$ est vide, nous avons également présenté une méthode heuristique qui permet de proposer de nouveaux critères de fusion, en agençant les critères testés précédemment avec des ET logiques et/ou des OU logiques.

5

Contribution : Fonction d'approximation qui utilise les états du transducteur

Ce chapitre présente un autre mécanisme d'approximation, qui consiste à raisonner sur l'application de k copies d'un transducteur $\mathcal{T}$ – qui représente la relation de transition d'un système étudié – à un automate $\mathcal{A}$ – qui représente les configurations initiales de ce système. Les états parcourus dans le transducteur sont encodés par des mots finis, et un automate supplémentaire C est utilisé pour spécifier quelles sont les combinaisons d'états de transducteurs qui doivent être fusionnées. Cette technique est inspirée par la construction de [BJNT00] où des états du transducteur $\mathcal{T}$ – qui représente la relation d'évolution du système – sont fusionnés. Par rapport à [BJNT00], nous définissons un nouveau critère de fusion –l'automate C – et à chaque étape i de sur-approximation (*cf.* flèches en diagonales dans la figure 3.2), nous fusionnons des états de l'automate $\mathcal{A}_i$ – qui représente les configurations accessibles par le système. Dans notre approche, le transducteur $\mathcal{T}$ n'est jamais modifié.

5.1/ Définitions supplémentaires

Z-automates

Nous commençons par quelques définitions et notations nécessaires pour le reste de ce chapitre.

Soient Z un ensemble, $\mathcal{A}$ un automate et $Q_{\mathcal{A}}$ l'ensemble fini des états de $\mathcal{A}$. L'automate $\mathcal{A}$ est un Z-automate s'il existe un ensemble Q, avec $Q_{\mathcal{A}} \subseteq Q \times Z$.

Soit $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma_{\mathcal{A}}, E_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ un automate fini, $s \cdot \mathcal{A}$ est l'automate $(Q_{\mathcal{A}} \times \{s\}, \Sigma_{\mathcal{A}}, E, I, F)$ isomorphe à $\mathcal{A}$ et défini par :

- $E = \{((p_{\mathcal{A}}, s), a, (q_{\mathcal{A}}, s)) \mid \exists (p_{\mathcal{A}}, a, q_{\mathcal{A}}) \in E_{\mathcal{A}}\}$
- $I = \{(p_{\mathcal{A}}, s) \mid p_{\mathcal{A}} \in I_{\mathcal{A}}\}$.
- $F = \{(p_{\mathcal{A}}, s) \mid p_{\mathcal{A}} \in F_{\mathcal{A}}\}$.

Exemple 5.1. $x \cdot \mathcal{A}_{ex}$ *L'automate $x \cdot \mathcal{A}_{ex}$ est un $\{x\}$-automate, isomorphe à l'automate $\mathcal{A}_{ex}$ (cf. figure 5.1).*

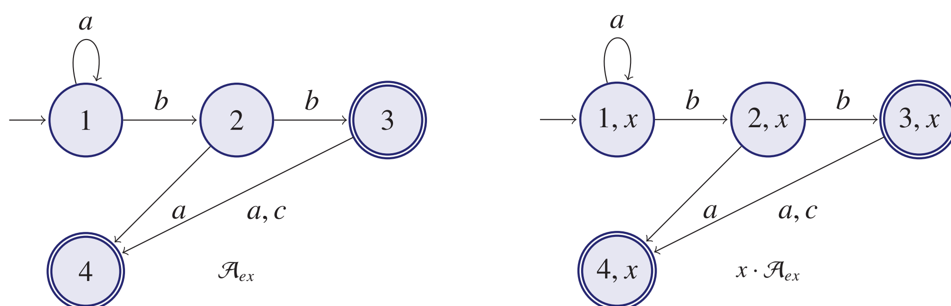


FIGURE 5.1 – $x \cdot \mathcal{A}_{ex}$

Exemple 5.2. $\mathcal{A}_{ex3}$ *L'automate* $\mathcal{A}_{ex3}$ *(cf. figure 5.2) est un* $\{w, x, y, z\}$*-automate, car* $Q_{ex3} \subseteq \{0, 1, 2, 3\} \times \{w, x, y, z\}$.

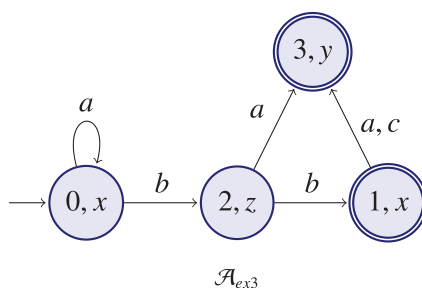


FIGURE 5.2 – $\mathcal{A}_{ex3}$

Exemple 5.3. $\mathcal{A}_{ex4}$ *L'automate* $\mathcal{A}_{ex4}$ *(cf. figure 5.3) est un* $\{w, x, y, z\}^*$*-automate, car* $Q_{ex4} \subseteq \{0, 1, 2, 3\} \times \{w, x, y, z\}^*$.

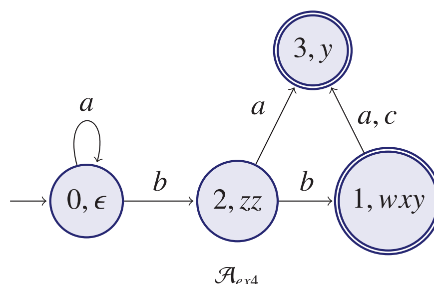


FIGURE 5.3 – $\mathcal{A}_{ex4}$

Proposition 5.4 *Pour tout ensemble Z et pour tout automate* $\mathcal{A}$*, si* $z \in Z$ *alors* $z \cdot \mathcal{A}$ *est un Z-automate.*

Proposition 5.5 *Pour tout ensemble Z et pour tout automate* $\mathcal{A}$*,* $\epsilon \cdot \mathcal{A}$ *est un* Z^**-automate.*

$Q_C \times Q_{\mathcal{T}}^*$-AUTOMATES

Notre méthode utilise des $Q_C \times Q_{\mathcal{T}}^*$-automates, qui sont des Z-automates tels que $Z = Q_C \times Q_{\mathcal{T}}^*$. Autrement dit, un automate $\mathcal{A} = (Q_{\mathcal{A}}, \Sigma_{\mathcal{A}}, E_{\mathcal{A}}, I_{\mathcal{A}}, F_{\mathcal{A}})$ est un $Q_C \times Q_{\mathcal{T}}^*$-automate si et seulement s'il existe un ensemble Q, avec $Q_{\mathcal{A}} \subseteq Q \times Q_C \times Q_{\mathcal{T}}^*$.

Nous utilisons ce type d'automates car il nous permet de mémoriser pour chaque état $(q_{\mathcal{A}_0}, q_C, \omega_{\mathcal{T}})$ de $\mathcal{A}$:

— l'état $q_{\mathcal{A}_0}$ – de l'automate de départ $\mathcal{A}_0$ – dont il est issu,
— la classe d'équivalence $q_C \in Q_C$ à laquelle il appartenait lors de la dernière fusion d'états,
— les états $q_{i_\mathcal{T}} \in Q_\mathcal{T}$ – du transducteur $\mathcal{T}$ – parcourus depuis.

Appliquer un transducteur $\mathcal{T}$ à un $Q_C \times Q_\mathcal{T}^*$-automate

Soit $\mathcal{T} = (Q_\mathcal{T}, \Sigma_\mathcal{A} \times \Sigma_\mathcal{A}, E_\mathcal{T}, I_\mathcal{T}, F_\mathcal{T})$ un transducteur, $C = (Q_C, \Sigma_C, E_C, I_C, F_C)$ un automate, et $\mathcal{A} = (Q_\mathcal{A}, \Sigma_\mathcal{A}, E_\mathcal{A}, I_\mathcal{A}, F_\mathcal{A})$ un $Q_C \times Q_\mathcal{T}^*$-automate, $\mathcal{T}(\mathcal{A})$ est l'automate (Q, Σ, E, I, F) défini par :

— $Q = \{(q_\mathcal{A}, (q_C, \omega_{q_\mathcal{A}} \cdot q_\mathcal{T}))$, tel que $(q_\mathcal{A}, (q_C, \omega_{q_\mathcal{A}})) \in Q_\mathcal{A}$ et $q_\mathcal{T} \in Q_\mathcal{T}\}$
— $\Sigma = \Sigma_\mathcal{A}$
— $E = \{(p_\mathcal{A}, (p_C, \omega_{p_\mathcal{A}} \cdot p_\mathcal{T})), b, (q_\mathcal{A}, (q_C, \omega_{q_\mathcal{A}} \cdot q_\mathcal{T}))$, tel qu'il existe $a, b \in \Sigma_\mathcal{A}$, $((p_\mathcal{A}, (p_C, \omega_{p_\mathcal{A}})), a, (q_\mathcal{A}, (q_C, \omega_{q_\mathcal{A}}))) \in E_\mathcal{A}$, et $(p_\mathcal{T}, a|b, q_\mathcal{T}) \in E_\mathcal{T}\}$
— $I = \{(p_\mathcal{A}, (p_C, \omega_{p_\mathcal{A}} \cdot p_\mathcal{T}))$, tel que $(p_\mathcal{A}, (p_C, \omega_{p_\mathcal{A}})) \in I_\mathcal{A}$, et $p_\mathcal{T} \in I_\mathcal{T}\}$
— $F = \{(p_\mathcal{A}, (p_C, \omega_{p_\mathcal{A}} \cdot p_\mathcal{T}))$, tel que $(p_\mathcal{A}, (p_C, \omega_{p_\mathcal{A}})) \in F_\mathcal{A}$, et $p_\mathcal{T} \in F_\mathcal{T}\}$

L'automate obtenu par cette application du transducteur est isomorphe à l'automate obtenu avec l'application courante d'un transducteur (*cf.* section 2.1.7). Cependant, avec cette nouvelle méthode d'application du transducteur, on utilise le nom des états pour mémoriser – puis retrouver facilement – les informations que l'on utilise pour comparer les états entre eux.

Exemple 5.6. Token Ring : Appliquer un transducteur $\mathcal{T}$ à un $Q_C \times Q_\mathcal{T}^*$-automate *(cf. figures 5.4 et 5.5).*

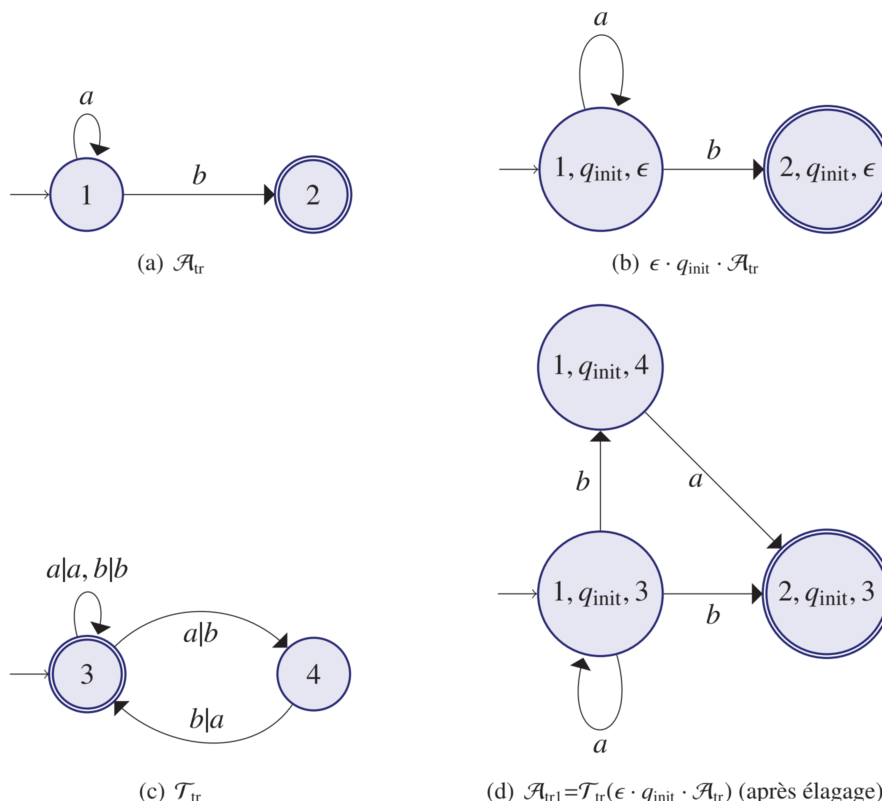
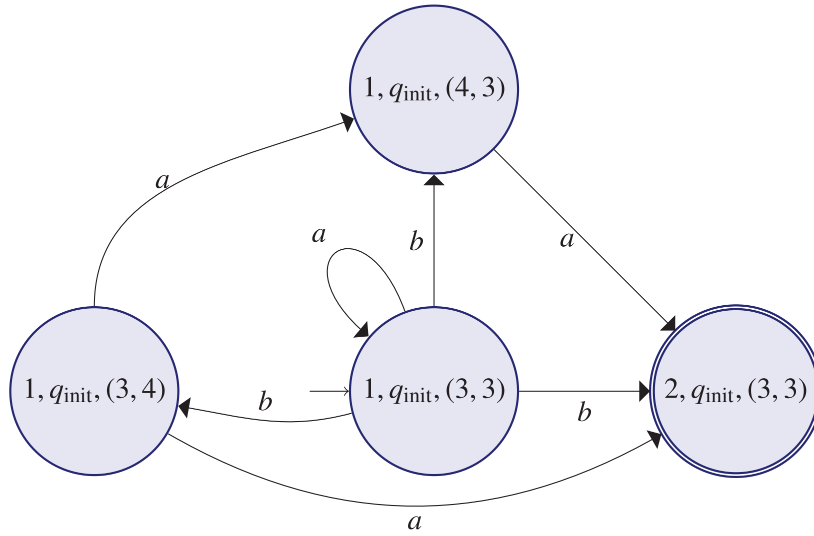


FIGURE 5.4 – Token ring : Appliquer un transducteur $\mathcal{T}$ à un $Q_C \times Q_\mathcal{T}^*$-automate (1)



(a) $\mathcal{A}_{\text{tr2}}=\mathcal{T}_{\text{tr}}(\mathcal{A}_{\text{tr1}})$ (après élagage)

FIGURE 5.5 – Token ring : Appliquer un transducteur $\mathcal{T}$ à un $Q_C \times Q_{\mathcal{T}}^*$-automate (2)

5.2/ CRITÈRE DE FUSION QUI UTILISE LES ÉTATS DU TRANSDUCTEUR

Soit $\mathcal{A} = (Q, \Sigma, E, I, F)$ un automate, $\mathcal{T} = (Q_{\mathcal{T}}, \Sigma \times \Sigma, E_{\mathcal{T}}, I_{\mathcal{T}}, F_{\mathcal{T}})$ un transducteur, et $C = (Q_C, Q_{\mathcal{T}}, E_C, \{q_{\text{init}}\}, \emptyset)$ un automate déterministe complet sur $Q_{\mathcal{T}}$, c'est-à-dire que les transitions de C sont étiquetées avec des états de $\mathcal{T}$. Soit φ_k une bijection des états de l'automate $\mathcal{T}^k(\mathcal{A})$ – qui sont de la forme $(((Q \times Q_{\mathcal{T}}) \times Q_{\mathcal{T}}) \ldots \times Q_{\mathcal{T}})$ – vers $Q \times Q_{\mathcal{T}}^k$, où $Q_{\mathcal{T}}^k$ est l'ensemble des mots de longueur k sur $Q_{\mathcal{T}}$. On définit une relation $\sim_C$ sur les états de $\mathcal{T}^k(\mathcal{A})$ de la façon suivante : si p et q sont des états de $\mathcal{T}^k(\mathcal{A})$ tels que $\varphi_k(p) = (p_0, w_p)$ et $\varphi_k(q) = (q_0, w_q)$, alors $p \sim_C q$ si et seulement si $p_0 = q_0$ et $q_{\text{init}} \cdot w_p = q_{\text{init}} \cdot w_q$ [1]. Autrement dit, $p \sim_C q$ si et seulement s'ils sont issus du même état de l'automate initial $\mathcal{A}_0$, et si la lecture – dans l'automate C – des états du transducteur dont sont issus p et q, conduit au même état de l'automate C. L'automate $\mathcal{T}^k(\mathcal{A})/_{\sim_C}$ est noté $\mathcal{T}_C^k(\mathcal{A})$.

Exemple 5.7. Token Ring : fusion en fonction des états du transducteur *Examinons à nouveau $\mathcal{A}_{\text{tr}}$ et $\mathcal{T}_{\text{tr}}$ de la figure 5.4, ainsi que l'automate C représenté dans la figure 5.6(a). L'automate $\mathcal{T}_{\text{tr}}^2(\mathcal{A}_{\text{tr}})$ est représenté dans la figure 5.6(b), après élagage. Les automates $\mathcal{T}_{\text{tr}}(\mathcal{A}_{\text{tr}})/_{\sim_C}$ et $\mathcal{T}_{\text{tr}}^2(\mathcal{A}_{\text{tr}})/_{\sim_C}$ sont représentés dans la figure 5.7. Par exemple, dans $\mathcal{T}_{\text{tr}}^2(\mathcal{A}_{\text{tr}})$ les états $(1, 3, 4)$ et $(1, 4, 3)$ sont $\sim_C$-équivalents puisqu'ils ont tous deux 1 comme premier élément, et $q_{\text{init}} \cdot 34 = q_{\text{init}} \cdot 43 = \triangledown$.*

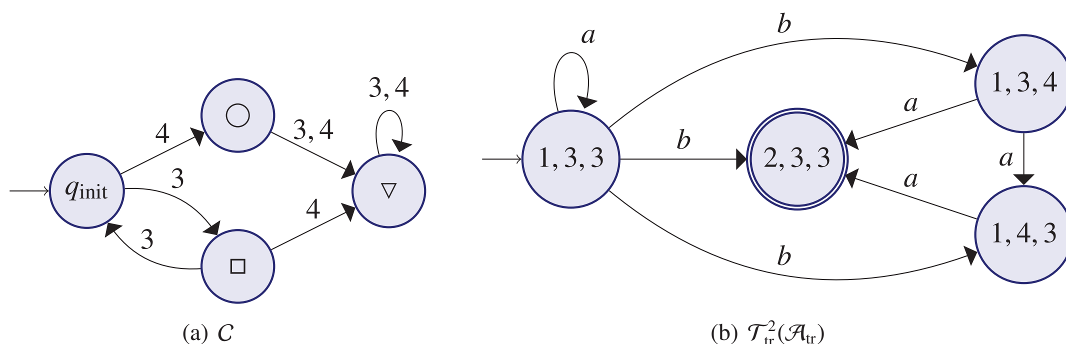
Proposition 5.8 *La relation $\sim_C$ est une relation d'équivalence.*

PREUVE. Soient p, q, et r des états de $\mathcal{T}^k(\mathcal{A})$ tels que $\varphi_k(p) = (p_0, w_p)$, $\varphi_k(q) = (q_0, w_q)$, et $\varphi_k(r) = (r_0, w_r)$.

φ_k étant une application, si $p = q$ alors $p_0 = q_0$ et $w_p = w_q$. L'automate C étant déterministe, si $w_p = w_q$ alors $q_{\text{init}} \cdot w_p = q_{\text{init}} \cdot w_q$. Donc si $p = q$, alors $p \sim_C q$. Autrement dit, la relation $\sim_C$ est réflexive.

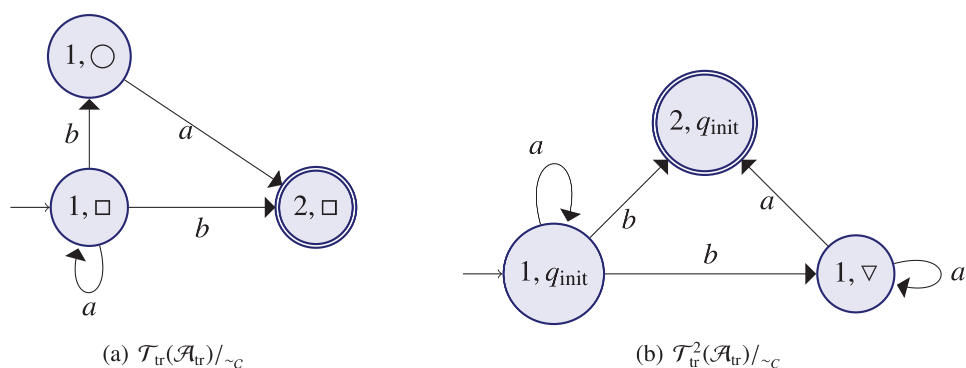
Par définition de la relation $\sim_C$, si $p \sim_C q$ alors $p_0 = q_0$ et $q_{\text{init}} \cdot w_p = q_{\text{init}} \cdot w_q$. Or $p_0 = q_0$ et $q_{\text{init}} \cdot w_p = q_{\text{init}} \cdot w_q$ si et seulement si $q_0 = p_0$ et $q_{\text{init}} \cdot w_q = q_{\text{init}} \cdot w_p$, qui est la définition de $q \sim_C p$. Donc si $p \sim_C q$, alors $q \sim_C p$. Autrement dit, la relation $\sim_C$ est symétrique.

1. $q_{\text{init}} \cdot w_p$ et $q_{\text{init}} \cdot w_q$ existent et sont uniques, car C est par définition déterministe et complet sur $Q_{\mathcal{T}}$.



(a) C (b) $\mathcal{T}^2_{tr}(\mathcal{A}_{tr})$

FIGURE 5.6 – Token Ring : fusion en fonction des états du transducteur (1)



(a) $\mathcal{T}_{tr}(\mathcal{A}_{tr})/_{\sim_C}$ (b) $\mathcal{T}^2_{tr}(\mathcal{A}_{tr})/_{\sim_C}$

FIGURE 5.7 – Token ring : fusion en fonction des états du transducteur (2)

Par définition de la relation $\sim_C$, si $p \sim_C q$ et $q \sim_C r$, alors $p_0 = q_0$ et $q_{init} \cdot w_p = q_{init} \cdot w_q$ et $q_0 = r_0$ et $q_{init} \cdot w_q = q_{init} \cdot w_r$. Or si $p_0 = q_0$ et $q_0 = r_0$ alors $p_0 = r_0$, et si $q_{init} \cdot w_p = q_{init} \cdot w_q$ et $q_{init} \cdot w_q = q_{init} \cdot w_r$ alors $q_{init} \cdot w_p = q_{init} \cdot w_r$. Donc si $p \sim_C q$ et $q \sim_C r$, alors $p \sim_C r$. Autrement dit, la relation $\sim_C$ est transitive.

La relation $\sim_C$ étant à la fois réflexive, symétrique et transitive, c'est donc une relation d'équivalence. □

Un moyen direct et naïf de calculer $\mathcal{T}^k(\mathcal{A})/_{\sim_C}$ serait de commencer par calculer $\mathcal{T}^k(\mathcal{A})$, puis de calculer son quotient par $\sim_C$. Cependant, $\mathcal{T}^k(\mathcal{A})$ ayant un nombre d'états exponentiel en k, nous proposons une construction alternative que nous décrivons ci-dessous.

Proposition 5.9 *Un automate isomorphe à $\mathcal{T}^k(\mathcal{A})/_{\sim_C}$ peut être calculé en temps polynomial par rapport à k et aux tailles de $\mathcal{A}$, $\mathcal{T}$ et C.*

PREUVE. La preuve est organisée de la façon suivante. Nous commençons par construire un isomorphisme ψ de chaque $\mathcal{T}^k_C(\mathcal{A})$, et un automate $\psi(\mathcal{T}^k_C(\mathcal{A}))$. Puis nous présentons une fonction Θ et nous prouvons par induction que $\psi(\mathcal{T}^k_C(\mathcal{A})) = \Theta^k(\mathcal{A}_0)$, où $\mathcal{A}_0$ est une copie particulière de $\mathcal{A}$. Et enfin, nous prouvons que ce calcul peut être effectué en temps polynomial.

Soient p et q des états $\mathcal{T}^k(\mathcal{A})$. Supposons que $\varphi_k(p) = (p_0, w_p)$ et $\varphi_k(q) = (q_0, w_q)$. On peut noter que si $p \sim_C q$, alors, par définition de $\sim_C$, $q_{init} \cdot w_p = q_{init} \cdot w_q$. On peut donc définir la fonction ψ de l'ensemble des classes d'équivalence de $\sim_C$ sur les états $\mathcal{T}^k(\mathcal{A})$ vers $Q \times Q_C$ par $\psi(p) = (p_0, q_{init} \cdot w_p)$. On affirme que ψ est un isomorphisme de $\mathcal{T}^k_C(\mathcal{A})$ à $\psi(\mathcal{T}^k_C(\mathcal{A}))$. En effet, si

$\psi(p) = \psi(q)$, avec $\varphi_k(p) = (p_0, w_p)$ et $\varphi_k(q) = (q_0, w_q)$, alors $p_0 = q_0$ et $q_{\text{init}} \cdot w_p = q_{\text{init}} \cdot w_q$. Donc $p \sim_C q$, ce qui prouve que ψ est injective.

Nous allons maintenant démontrer que $\psi(\mathcal{T}_C^k(\mathcal{A}))$ peut être calculé en temps polynomial. Soit $\mathcal{A}_0 = (Q \times \{q_{\text{init}}\}, \Sigma, \Delta_0, I \times \{q_{\text{init}}\}, F \times \{q_{\text{init}}\})$, où

$$\Delta_0 = \{((p, q_{\text{init}}), a, (q, q_{\text{init}})) \mid (p, a, q) \in \Delta\}.$$

Soit Θ la fonction de $(Q \times Q_C) \times Q_{\mathcal{T}}$ vers $Q \times Q_C$ définie par $\Theta((q_1, q_2), \alpha) = (q_1, q_2 \cdot \alpha)$. Notons que $q_2 \cdot \alpha$ est bien défini puisque l'alphabet de C est $Q_{\mathcal{T}}$. On définit maintenant $\mathcal{A}_{k+1}$ pour $k \geq 0$, par $\mathcal{A}_{k+1} = \Theta(\mathcal{T}(\mathcal{A}_k))$. Nous allons démontrer par induction sur k que $\mathcal{A}_{k+1}$ est égal à $\psi(\mathcal{T}_C^{k+1}(\mathcal{A}))$. On note E_k l'ensemble des transitions de $\mathcal{A}_k$.

- Par définition, $\mathcal{A}_1 = \Theta(\mathcal{T}(\mathcal{A}_0))$. L'ensemble des états de $\mathcal{T}(\mathcal{A}_0)$ est $Q \times \{q_{\text{init}}\} \times Q_{\mathcal{T}}$. Donc l'ensemble des états de $\mathcal{A}_1$ est $Q \times (q_{\text{init}} \cdot Q_{\mathcal{T}})$. Soit p un état de $\mathcal{T}(\mathcal{A})$ et soit $\varphi_1(p) = (p_0, \alpha)$ avec $\alpha \in Q_{\mathcal{T}}$, alors $\psi(p) = (p_0, q_{\text{init}} \cdot \alpha)$. Donc l'ensemble des états de $\psi(\mathcal{T}(\mathcal{A}))$ est également $Q \times (q_{\text{init}} \cdot Q_{\mathcal{T}})$. $\psi(\mathcal{T}(\mathcal{A}))$ et $\mathcal{A}_1$ ont tous deux $\{(p, q_{\text{init}} \cdot \alpha) \mid p \in I, \alpha \in I_T\}$ pour ensemble d'états initiaux, et $\{(p, q_{\text{init}} \cdot \alpha) \mid p \in F, \alpha \in F_T\}$ pour ensemble d'états finaux. L'ensemble E_1 des transitions de $\mathcal{A}_1$ est

$$\{((p, q_{\text{init}} \cdot \alpha), a, (p', q_{\text{init}} \cdot \alpha') \mid \exists (p, b, p') \in E \text{ et } (\alpha, (b, a), \alpha') \in E_T\},$$

  qui est exactement le même que l'ensemble des transitions de $\psi(\mathcal{T}(\mathcal{A}))$. On peut en déduire que $\psi(\mathcal{T}(\mathcal{A})) = \mathcal{A}_1$.
- Supposons maintenant que $\mathcal{A}_k = \psi(\mathcal{T}_C^k(\mathcal{A}))$, avec $k \geq 1$. Par définition, $\mathcal{A}_{k+1} = \Theta(\mathcal{T}(\mathcal{A}_k))$. Donc l'ensemble des états de $\mathcal{A}_{k+1}$ est

$$Q \times \{q \cdot \alpha \mid \alpha \in Q_{\mathcal{T}} \text{ et } q \text{ est un état de } \mathcal{A}_k\}.$$

  C'est pourquoi, par induction directe, l'ensemble des états de $\mathcal{A}_{k+1}$ est

$$Q \times \{q_{\text{init}} \cdot w \mid w \in Q_{\mathcal{T}}^{k+1}\},$$

  qui est exactement l'ensemble des états de $\psi(\mathcal{T}_C^{k+1}(\mathcal{A}))$. Les états initiaux sont dans l'ensemble $\{(p, q_{\text{init}} \cdot w) \mid p \in I \text{ et } w \in I_T^{k+1}\}$ pour $\psi(\mathcal{T}_C^{k+1}(\mathcal{A}))$ et $\mathcal{A}_{k+1}$. De façon similaire, leur ensemble d'états finaux est $\{(p, q_{\text{init}} \cdot w) \mid p \in F \text{ et } w \in I_F^{k+1}\}$. On a également

$$E_k = \{(p, q \cdot \alpha), a, (p', q' \cdot \alpha') \mid \exists ((p, q), b, (p', q')) \in E_k \text{ et } (\alpha, (b, a), \alpha') \in E_T\}.$$

  L'ensemble des transitions de $T^{k+1}(\mathcal{A})$ est :

$$\{((p, p_1, \ldots, p_{k+1}), a, (p', p'_1, \ldots, p'_{k+1})) \mid (p_{k+1}, (b, a), p'_{k+1}) \in E_T \text{ et } \exists ((p, p_1, \ldots, p_k), b, (p', p'_1, \ldots, p'_k)) \text{ transition de } T^k(\mathcal{A})\}.$$

  L'ensemble des transitions de $\psi(\mathcal{T}_C^{k+1}(\mathcal{A}))$ est donc

$$\{((p, q_{\text{init}} \cdot (w p_{k+1})), a, (p', q_{\text{init}} \cdot (w' p'_{k+1}))) \mid (p_{k+1}, (b, a), p'_{k+1}) \in E_T \text{ et } \exists ((p, q_{\text{init}} \cdot w), b, (p', q_{\text{init}} \cdot w') \text{ transition de } \psi(T_C^k(\mathcal{A}))\}.$$

  Comme, par l'hypothèse d'induction, $\psi(\mathcal{T}_C^k(\mathcal{A})) = \mathcal{A}_k$, $\mathcal{A}_{k+1}$ et $\psi(\mathcal{T}_C^{k+1}(\mathcal{A}))$ ont le même ensemble de transitions, cela conclut la preuve par induction.

Semi-Algorithme `PointFixeT`
Entrée : $\mathcal{A}, \mathcal{T}, \mathcal{B}, C$
Variable : k
  k := 0
  Tant que $(L(\mathcal{T}_C^{k+1}(\mathcal{A})) \neq L(\mathcal{T}_C^{k}(\mathcal{A})))$ **faire**
    $k := k + 1$
  FinTantQue
  Si $(L(\mathcal{T}_C^{k}(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset)$ **alors**
    Retourner *Sûr*
  Sinon
    Retourner *Non-conclusif*
  FinSiAlorsSinon

FIGURE 5.8 – Semi-algorithme `PointFixeT`

Il reste à montrer que $\mathcal{A}_k$ peut être calculé en temps polynomial. Comme à chaque étape d'induction k l'automate isomorphe visé a au plus $|Q \times Q_C|$ états, on peut calculer $\mathcal{A}_k$ en temps polynomial par rapport à k et aux tailles de $\mathcal{A}$, $\mathcal{T}$ et C. □

Soit un automate $\mathcal{B}$ donné, on peut désormais utiliser les automates calculés en appliquant le semi-algorithme `PointFixeT` représenté dans la figure 5.8. On peut donc espérer obtenir une sur-approximation des états accessibles : si le semi-algorithme `PointFixeT` s'arrête sur une approximation qui n'est pas trop grossière on pourra en déduire que $\mathcal{R}^*_{\mathcal{T}}(L(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset$.

Proposition 5.10 *Le semi-algorithme* `PointFixeT` *est correct : s'il retourne Sûr alors* $\mathcal{R}^*_{\mathcal{T}}(L(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset$.

La preuve de la Proposition 5.10 est similaire à la preuve de la Proposition 3.6.

5.3/ RAFFINER LES APPROXIMATIONS QUI UTILISENT LES ÉTATS DU TRANSDUCTEUR

Dans cette section, on propose de raffiner les approximations qui utilisent les états du transducteur, lorsque l'approche par sur-approximations successives (*cf.* figure 3.2) se termine sans permettre de conclure. Intuitivement, cela se produit lorsque l'approximation est trop grossière.

Notre algorithme peut être vu comme faisant partie des algorithmes CEGAR (*cf.* section 3.4).

Proposition 5.11 *Si* $L(\mathcal{T}_C^{k}(\mathcal{A})) \cap L(\mathcal{B}) \neq \emptyset$*, alors soit* $L(\mathcal{A}) \cap L(\mathcal{T}^{-k}(\mathcal{B})) \neq \emptyset$*, soit il existe* j*,* $0 \leq j \leq k$ *tel que* $L(\mathcal{T}_C^{j}(\mathcal{A})) \cap L(\mathcal{T}^{j-k}(\mathcal{B})) \neq \emptyset$ *et* $L(\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))) \cap L(\mathcal{T}^{j-k}(\mathcal{B})) = \emptyset$.

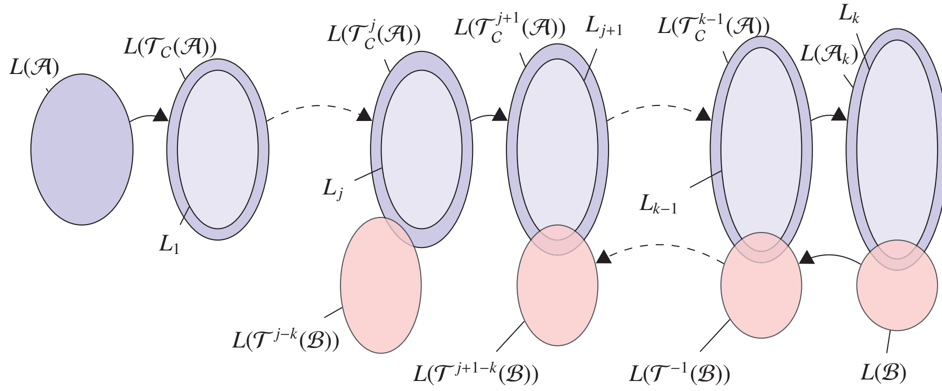
PREUVE. Supposons que $L(\mathcal{T}_C^{k}(\mathcal{A})) \cap L(\mathcal{B}) \neq \emptyset$. La construction est présentée dans la figure 5.9. On définit $L_i = L(\mathcal{T}(\mathcal{T}_C^{i-1}(\mathcal{A}))$ pour tout $1 \leq i \leq k$.

Supposons tout d'abord que $L_k \cap L(\mathcal{B}) = \emptyset$, alors $j = k$ et la proposition supposée tient pour $j = k$. Supposons ensuite que $L_k \cap L(\mathcal{B}) \neq \emptyset$. Soit

$$K = \{\ell \mid 1 \leq \ell \leq k \wedge L_\ell \cap L(\mathcal{T}^{\ell-k}(\mathcal{B})) \neq \emptyset\}.$$

Par hypothèse $k \in K$. On peut donc définir $j = -1 + \min K$. Deux cas se présentent :

- $j \neq 0$: Puisque $j \notin K$, on a $L_j \cap L(\mathcal{T}^{j-k}(\mathcal{B})) = \emptyset$. Puisque $j+1 \in K$, $L_{j+1} \cap L(\mathcal{T}^{j-k+1}(\mathcal{B})) \neq \emptyset$. Donc $L(\mathcal{T}^{j-k}(\mathcal{B})) \cap L(\mathcal{T}_C^{j}(\mathcal{A})) \neq \emptyset$.
- $j = 0$: On a donc $L_1 \cap L(\mathcal{T}^{1-k}(\mathcal{B})) \neq \emptyset$. Par conséquent, $L(\mathcal{A}) \cap L(\mathcal{T}^{-k}(\mathcal{B})) \neq \emptyset$, ce qui complète la preuve.



les L_i's représentent les langages $L(\mathcal{T}(\mathcal{T}_C^{i-1}(\mathcal{A}))$'s.

FIGURE 5.9 – Situation nécessitant un raffinement de l'approximation

□

Supposons que $L(\mathcal{T}_C^j(\mathcal{A})) \cap L(\mathcal{T}^{j-k}(\mathcal{B})) \neq \emptyset$ et $L(\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))) \cap L(\mathcal{T}^{j-k}(\mathcal{B})) = \emptyset$. Comme on le fait souvent en CEGAR, on peut calculer une relation d'équivalence $\equiv$ sur $\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))$ telle que $\equiv \subseteq \sim_C$ et $L(\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))) /_{\equiv} \cap L(\mathcal{T}^{j-k}(\mathcal{B})) = \emptyset$. L'existence de $\equiv$ est triviale puisque les résultats tiennent pour la relation identité. Néanmoins, en utilisant l'approche CEGAR, notre but est de calculer une relation $\equiv$ aussi large que possible, dans l'idée d'assurer la terminaison de l'exploration de l'espace d'états. Intuitivement, la relation $\equiv$ peut être calculée simplement à partir des automates $\mathcal{A}' = \mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))$ et $\mathcal{AB} = \mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A})) \times \mathcal{T}^{j-k}(\mathcal{B})$, de la façon suivante :

- On fusionne deux à deux les états p et q de $\mathcal{A}'$ tels que $p \sim_C q$,
- pour chaque fusion dans $\mathcal{A}'$, on fusionne les états correspondants à p et q dans l'automate $\mathcal{AB}$.
  - Si $L(\mathcal{AB}) = \emptyset$ alors $p \equiv q$,
  - sinon on annule la fusion de p et de q, et $p \not\equiv q$.

Au lieu de calculer la relation $\equiv$, puis construire l'automate $\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))/_{\equiv}$ correspondant, puis calculer le point fixe, on propose d'utiliser une approche dynamique. Plus précisément, on préfère modifier C en fonction de $\equiv$ pour éviter de fusionner des états similaires qui pourraient conduire à une sur-approximation trop grossière. Pour modifier C en fonction de $\equiv$, on propose d'utiliser les algorithmes des figures 5.10 et 5.11. L'algorithme `Séparer` modifie l'automate déterministe donné pour produire une abstraction plus large. L'idée de cet algorithme est assez naturelle : si deux états équivalents peuvent être distingués, l'automate C est raffiné pour prendre en compte cette contrainte. L'algorithme `Raffiner` (*cf.* figure 5.11) utilise l'algorithme `Séparer` – qui dissocie deux états – autant de fois que nécessaire pour obtenir l'approximation raffinée.

Exemple 5.12. *La figure 5.12(a) montre l'automate C' obtenu par* `Séparer`$(C, \bigcirc, \square, 3, 4)$*, où C est l'automate de la figure 5.6(a).*

Proposition 5.13 *L'algorithme* `Raffiner` *termine toujours.*

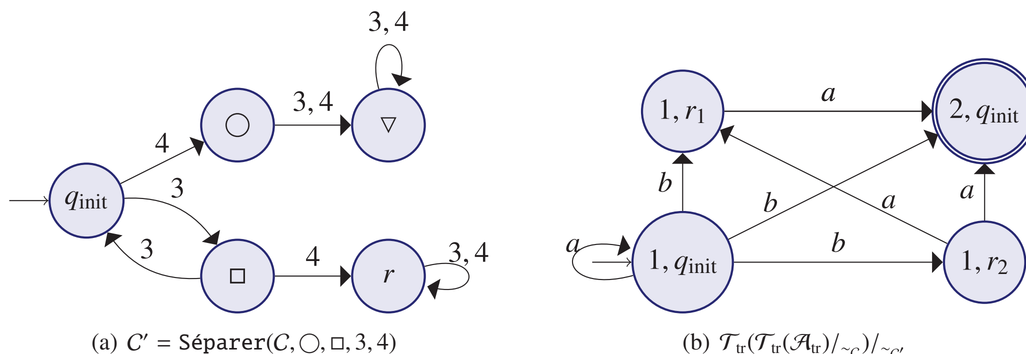
PREUVE. Soient $\mathcal{T}$ un transducteur, C un automate déterministe, $S = (Q_S \times Q_C, Q, E, \{q_0\}, F_S)$ un automate fini et $\equiv$ une relation telle que $\equiv \subseteq \sim_C$ et $L(\mathcal{T}_C(\mathcal{A}))/_{\equiv} \cap L(\mathcal{T}^{-1}(\mathcal{B})) = \emptyset$. Soit $C_2 =$

Algorithme `Séparer`
Entrée : Un automate déterministe $S = (Q_S, Q_{\mathcal{T}}, E_S, \{q_0\}, \emptyset)$, $p, q \in Q_S$ et $\alpha, \beta \in Q_{\mathcal{T}}$ tels que $p \cdot_S \alpha = q \cdot_S \beta$
$Q'_S := Q_S \cup \{r\}$ où $r \notin Q_S$
$I'_S := \{q_0\}$
$E'_S := E_S \setminus \{(q, \beta, q \cdot_S \beta)\}$
$E'_S := E'_S \cup \{(q, \beta, r)\} \cup \{(r, a, s) \mid (p \cdot \alpha, a, s) \in E_S \text{ et } s \in Q_S \setminus \{p \cdot_S \alpha\}\}$
$E'_S := E'_S \cup \{(r, a, r) \mid (p \cdot \alpha, a, p \cdot \alpha) \in E_S\}$
Retourner $(Q'_S, Q_{\mathcal{T}}, E'_S, I'_S, \emptyset)$

FIGURE 5.10 – Algorithme `Séparer`

Algorithme `Raffiner`
Entrée : Un transducteur $\mathcal{T}$, un automate déterministe C, un automate $S = (Q_S \times Q_C, Q, E, \{q_0\}, F_S)$, une relation $\equiv$ telle que $\equiv \subseteq \sim_C$ et $L(\mathcal{T}_C(\mathcal{A}))/_{\equiv} \cap L(\mathcal{T}^{-1}(\mathcal{B})) = \emptyset$
Tant que $(\sim_C \not\subseteq \equiv)$ **faire**
Choisir (p, q, α) **et** (p, q', α') deux états de $\mathcal{T}(S)$ **tels que** $(p, q, \alpha) \sim_C (p, q', \alpha')$ mais $(p, q, \alpha) \not\equiv (p, q', \alpha')$
C :=`Séparer`$(C, q, \alpha, q', \alpha')$
FinTantQue
Retourner C

FIGURE 5.11 – Algorithme `Raffiner`



(a) $C' = $ `Séparer`$(C, \bigcirc, \square, 3, 4)$ (b) $\mathcal{T}_{\text{tr}}(\mathcal{T}_{\text{tr}}(\mathcal{A}_{\text{tr}})/_{\sim_C})/_{\sim_{C'}}$

FIGURE 5.12 – Exemples pour les algorithmes `Séparer` et `Raffiner`

`Split`$(C, q, \alpha, q', \alpha')$, où q, α, q', α' satisfait $q \cdot_C \alpha = q' \cdot_C \alpha'$ et $(q, \alpha) \neq (q', \alpha')$.

Dans un premier temps, on affirme que $\sim_{C_2} \subseteq \sim_C$. Supposons que (p, a, s) et (p, a', s') sont des états de $\mathcal{T}(S)$ tels que $(p, a, s) \nsim_C (p, a', s')$. Quatre cas se présentent alors :

Cas 1 : $(a, s) \neq (q', \alpha')$ et $(a', s') \neq (q', \alpha')$. Donc, par construction dans `Séparer`, $a \cdot_{C_2} s = a \cdot_C s$ et $a' \cdot_{C_2} s' = a' \cdot_C s'$. Puisque $(p, a, s) \nsim_C (p, a', s')$, on a $a \cdot_C s \neq a' \cdot_C s'$. Par conséquent, $a \cdot_{C_2} s \neq a' \cdot_{C_2} s'$, ce qui permet de démontrer que $(p, a, s) \nsim_{C_2} (p, a', s')$

Cas 2 : $(a, s) = (q', \alpha')$ et $(a', s') \neq (q', \alpha')$. Donc $a' \cdot_{C_2} s' = a' \cdot_C s'$ et $a \cdot_{C_2} s = r$, où r est le nouvel état crée par l'algorithme `Séparer`. En particulier, $r \neq a' \cdot_C s'$. Par conséquent, $a \cdot_{C_2} s \neq a' \cdot_{C_2} s'$. On peut noter que dans ce cas, l'hypothèse $(p, a, s) \nsim_C (p, a', s')$ est inutile.

Cas 3 : $(a, s) \neq (q', \alpha')$ et $(a', s') = (q', \alpha')$. Ce cas est le dual au Cas 2.

Cas 4 : $(a, s) = (q', \alpha')$ et $(a', s') = (q', \alpha')$. Ceci est impossible puisque $(p, a, s) \nsim_C (p, a', s')$.

Dans un deuxième temps, les Cas 2 et 3 ci-dessus montrent que $q \cdot_{C_2} \alpha \neq q' \cdot_{C_2} \alpha'$. Donc l'inclusion $\sim_{C_2} \subsetneq \sim_C$ est stricte. Ceci implique, grâce à la finitude des cardinalités, que l'algorithme `Raffiner` termine toujours.

□

Exemple 5.14. *On considère la relation* $\equiv$ *dont les classes d'équivalence sont* $\{1, \square, 4\}$, $\{(2, \square, 3)\}$, $\{(1, \bigcirc, 3), (1, \bigcirc, 4)\}$ *et* $\{(1, \square, 3)\}$. *On applique l'algorithme* `Raffiner` *aux automates* $\mathcal{T}_{\mathrm{tr}}$ *(figure 5.4(c)),* C *(figure 5.6(a)), et* $\mathcal{T}_{\mathrm{tr}}(\mathcal{A}_{\mathrm{tr}})/_{\sim_C}$ *(figure 5.7(a)). Puisque* $(1, \bigcirc, 3) \sim_C (1, \square, 4)$*, on a* $\sim_C \nsubseteq \equiv$. *Donc l'algorithme va calculer* $C' =$ `Séparer`$(C, \bigcirc, \square, 3, 4)$ *(figure 5.12(a)). On peut donc vérifier que* $\sim_{C'} \subseteq \equiv$. *L'automate* $\mathcal{T}_{\mathrm{tr}}(\mathcal{T}_{\mathrm{tr}}(\mathcal{A}_{\mathrm{tr}})/_{\sim_C})/_{\sim_{C'}}$ *est représenté dans la figure 5.12(b).*

Semi-Algorithme `Vérif-avec-raffinement`
Entrée : deux automates $\mathcal{A}, \mathcal{B}$, un transducteur $\mathcal{T}$, un automate déterministe C, un nombre entier ℓ
Variables : deux nombres entiers j, k, et une relation d'équivalence $\equiv$

- $k := \ell$
- **Tant que** $(L(\mathcal{T}_C^k(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset$ **et** $L(\mathcal{T}_C^{k+1}(\mathcal{A})) \neq L(\mathcal{T}_C^k(\mathcal{A})))$ **faire**
  - $k := k + 1$
- **FinTantQue**
- **Si** $(L(\mathcal{T}_C^{k+1}(\mathcal{A})) = L(\mathcal{T}_C^k(\mathcal{A}))$ **et** $L(\mathcal{T}_C^k(\mathcal{A})) \cap L(\mathcal{B}) = \emptyset)$ **alors**
  - **Retourner** *Sûr*
- **FinSi**
- **Si** $L(\mathcal{A}) \cap L(\mathcal{T}^{-k}(\mathcal{B})) \neq \emptyset$ **alors**
  - **Retourner** *Non sûr*
- **FinSi**
- $j :=$ `J`$(\mathcal{A}, \mathcal{B}, C, \mathcal{T}, k)$
- **Soit** $\equiv$ tel que $\equiv \subseteq \sim_C$ et $L(\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A})))/_{\equiv} \cap L(\mathcal{T}^{j-k}(\mathcal{B})) = \emptyset$
- **Retourner** `Vérif-avec-raffinement`$(\mathcal{A}, \mathcal{T}^{-k}(\mathcal{B}), \mathcal{T},$ `Raffiner`$(\mathcal{T}, C, \mathcal{T}^j(\mathcal{A}), \equiv), j)$

FIGURE 5.13 – Semi-algorithme `Vérif-avec-raffinement`

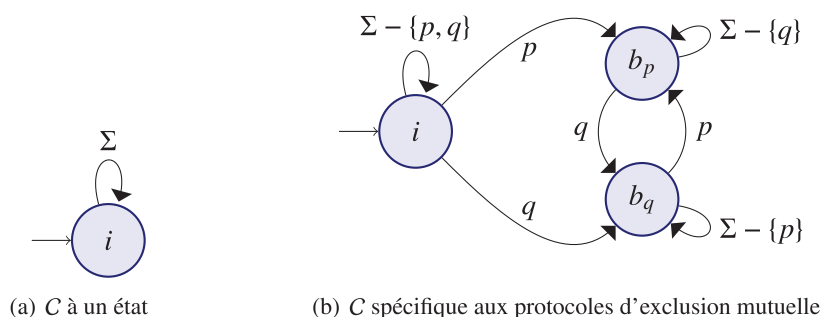
Si $L(\mathcal{T}_C^k(\mathcal{A})) \cap L(\mathcal{B}) \neq \emptyset$ et $L(\mathcal{A}) \cap L(\mathcal{T}^{-k}(\mathcal{B})) = \emptyset$, alors on note `J`$(\mathcal{A}, \mathcal{B}, C, \mathcal{T}, k)$ le plus grand nombre entier j tel que $0 \leq j \leq k$ et $L(\mathcal{T}_C^j(\mathcal{A})) \cap L(\mathcal{T}^{j-k}(\mathcal{B})) \neq \emptyset$ et $L(\mathcal{T}(\mathcal{T}_C^{j-1}(\mathcal{A}))) \cap L(\mathcal{T}^{j-k}(\mathcal{B})) = \emptyset$. Avec cette notation, le semi-algorithme `Vérif-avec-raffinement` (figure 5.13) correspond à l'approche suivante : chaque fois qu'une approximation trop grossière est détectée, elle est raffinée. Ce semi-algorithme retourne *Sûr* si on parvient à calculer une sur-approximation des configurations atteignables par le système dont l'intersection avec la propriété d'erreur est vide. Il retourne *Non sûr* s'il détecte qu'une partie de la propriété d'erreur et atteignable par le système. Malheureusement, il peut également ne pas parvenir à conclure si les approximations calculées doivent être raffinées encore et encore.

5.4/ EXPÉRIMENTATIONS

Cette section présente les expérimentations de notre technique de vérification qui utilise les états du transducteur comme critère de fusion d'états. Pour effectuer ces expérimentations, nous avons utilisé JAMF (pour Java Automata Manipulation Framework), qui est un framework de manipulation d'automates que nous avons développé en JAVA.

Nous avons expérimenté sur quelques exemples couramment étudiés dans la littérature, qui sont l'algorithme de la boulangerie par Lamport, l'algorithme de l'anneau à jeton (Token Ring), ainsi que les protocoles de Dijkstra et de Burns, encore avec le codage de [Tou01].

Pour les besoins d'expérimentations, nous avons défini différents types d'automates C : soit un automate C *à un état* (*cf.* figure 5.14(a)), soit un automate C *spécifique* (*cf.* figure 5.14(b)). Lorsqu'on commence avec un automate C à un état, tous les états de l'automate calculé sont donc C-équivalents. Alors qu'au contraire, un automate C spécifique représente une propriété d'intérêt. Par exemple, si deux a consécutifs sont interdits, et qu'il y a une transition $(p, (x, a), q)$ dans le transducteur du protocole considéré, alors l'automate C spécifique est représenté dans la figure 5.14(b). Intuitivement, le principe de cet automate C spécifique est d'éviter – à toute étape i – la fusion des états de $\mathcal{A}_i^e$ liés à l'état p du transducteur, avec les états de $\mathcal{A}_i^e$ liés à l'état q du transducteur. Ceci afin d'éviter d'obtenir une boucle sur un état de de $\mathcal{A}_i$ qui permettrait de lire une infinité de a.



(a) C à un état

(b) C spécifique aux protocoles d'exclusion mutuelle

FIGURE 5.14 – Types d'automates C

En suivant l'algorithme `Vérif-avec-raffinement`, si on parvient à obtenir un automate qui reconnaît le même langage que l'automate calculé à l'étape précédente, et que l'intersection entre ce langage et la propriété d'erreur est vide, alors le protocole est sûr (*cf.* valeurs en gras). Dans ce cas, on vérifie que le protocole ne pourra jamais atteindre une configuration qui satisfait la propriété d'erreur étudiée.

Les résultats sont détaillés dans le tableau de la figure 5.15. La première colonne indique le protocole étudié : son nome, la taille (c'est-à-dire $|Q| + |E|$) de l'automate initial, ainsi que la taille du transducteur. La seconde colonne indique si on est parti d'un automate C à un état, ou d'un automate spécifique. Les colonnes suivantes donnent les résultats obtenus, sans raffinement ou avec raffinement :

- la première ligne donne l'étape à laquelle a été atteinte l'égalité des langages, ou No si l'intersection avec la propriété d'erreur n'est plus vide avant d'avoir atteint l'égalité des langages, ou T.o (pour Time out) si l'algorithme a été arrêté sans avoir atteint l'égalité des langages ni trouvé d'intersection avec la propriété d'erreur,
- la seconde ligne indique l'étape à partir de laquelle l'intersection avec la propriété d'erreur n'est plus vide, ou No si cette intersection reste vide,
- la troisième ligne indique la taille du dernier automate obtenu.

Si on parvient à obtenir un automate qui reconnaît le même langage que l'automate calculé à l'étape précédente, et que l'intersection entre ce langage et la propriété d'erreur est vide, alors le protocole est sûr (*cf.* valeurs en gras) : le protocole ne pourra jamais atteindre une configuration qui satisfait la propriété d'erreur étudiée.

Notre méthode montre que les deux versions du protocole Token Ring sont sûres, en 4 étapes

		Critère de fusion
Algorithme Taille de l'automate initial Taille du transducteur	Automate C de départ	Équivalence des langages Intersection avec la propriété d'erreur Taille de l'automate obtenu

Résultats - Légende

		Sans raffinement	Avec raffinement
Token Ring (2 states in T) taille I : 4 taille T : 6	one state	No Step 2 7	**Step 4** No 8
Token Ring (3 states in T) taille I : 4 taille T : 9	one state	No Step 2 9	**Step 4** No 8
Dijkstra taille I : 5 taille T : 62	one state	No Step 5 97	T.o (Step 20) No 5955
	specific	**Step 15** No 65	**Step 15** No 65
Bakery taille I : 2 taille T : 24	one state	No Step 3 19	T.o (Step 20) No 851
	specific	No Step 3 20	**Step 6** No 16
Burns taille I : 2 taille T : 22	one state	No Step 4 18	T.o (Step 20) No 966
	specific	No Step 5 33	**Step 14** No 57

FIGURE 5.15 – Résultats avec la méthode reposant sur les états du transducteur comme critère de fusion

avec la méthode fondée sur les états du transducteur à partir d'un automate C à un état. A partir d'un automate C spécifique, on montre en 15 étapes que le protocole de Dijkstra est sûr, ceci sans utiliser de raffinement. On montre que l'algorithme de Bakery et le protocole de Burns sont sûrs en respectivement 6 et 14 étapes, à partir de automate C spécifique, en utilisant le raffinement. Pour tous ces protocoles, la taille des automates obtenus est dans le même ordre de grandeur que celle des automates en entrée, ce qui signifie que l'on parvient à éviter l'explosion du nombre d'états.

5.5/ CONCLUSION

Le mécanisme d'approximation présenté dans ce chapitre, ainsi que les critères de fusion qui y sont associés (automates C), sont généraux et rapides à calculer. Afin de gérer les cas dans lesquels aucun des critères définis ne permet de calculer une sur-approximation K de $\mathcal{R}^*_{\mathcal{T}}(\mathrm{I})$ dont l'intersection avec la propriété d'erreur $\mathcal{B}$ est vide, nous avons également présenté une méthode de raffinement grâce à laquelle nous parvenons à vérifier les exemples traités (*cf.* cases en gras dans le tableau de la figure 5.15).

6

Conclusion de la partie II

Développer des techniques de vérification efficaces basées sur des approximations est le défi à relever pour résoudre les problèmes d'accessibilité quand les méthodes exactes ne fonctionnent pas.

Dans cette partie, nous avons défini et présenté deux nouvelles techniques d'approximation pour le problème de l'accessibilité régulier, qui utilisent des algorithmes polynomiaux – à condition d'utiliser les algorithmes récents pour tester l'inclusion des langages de deux automates, comme dans [DR10, ACH+10, BP12] –. Ces techniques d'approximation sont génériques, c'est-à-dire qu'elles ne sont pas liées au spécificités d'un domaine en particulier, et sont donc applicables à tous types de problèmes réguliers (*cf.* définition 2.2). Nos propositions – contrairement à [BJNT00, BHV04] dont elles sont inspirées – n'exigent pas de minimiser/déterminiser l'automate obtenu à chaque étape du model-checking régulier. Nous avons publié ces travaux dans [DHK13a].

Comme direction future, nous pouvons envisager d'adapter nos travaux aux langages d'arbres, comme l'ont fait par exemple les auteurs de [BJNT00, BHV04] ; car après avoir défini une méthode de vérification qui fonctionne sur les mots, vient naturellement l'envie de la confronter à un modèle plus puissant comme les arbres. Mais avant cela, nous n'avons pas épuisé les idées d'améliorations de notre méthode sur les mots... Nous pensons que notre approche de raffinement doit pouvoir être améliorée, sur la précision des approximations comme sur le temps de calcul. Par exemple, pour la technique dont la fonction d'approximation est l'automate C, nous raffinons actuellement en dupliquant simplement l'état p de C qui correspond à la classe d'équivalence des états que l'on ne souhaite plus fusionner. Au lieu de ça, nous pourrions essayer de dupliquer l'ensemble de l'automate C, afin de séparer également les états de C qui suivent l'état p – nous avons nommé cette idée *raffinement par couleur* –. Une autre idée que nous n'avons pas eu le temps d'approfondir est d'éviter de fusionner entre eux des états qui étaient déjà présents dans l'automate précédent, afin que l'automate $\mathcal{A}_i$ ressemble davantage à l'automate $\mathcal{A}_{i-1}$, et d'augmenter ainsi les chances d'atteindre un point fixe. Et ensuite, une autre direction envisagée est de généraliser encore davantage nos mécanismes d'approximation, afin de les appliquer à d'autres problèmes de model-checking régulier, comme les systèmes à compteur (*cf.* [BFL04]) ou encore les systèmes à pile (*cf.* [EHRS00]).

III

Combiner génération aléatoire et critère de couverture, à partir d'une grammaire algébrique ou d'un automate à pile

Les modèles à pile sont fréquemment utilisés dans le cadre du test de logiciel. Par exemple, la génération de données d'entrée complexes passe souvent par une spécification du format de données valides sous forme de grammaire. Les automates à pile sont également fréquemment utilisés, par exemple pour représenter une abstraction du graphe de flot de contrôle – c'est-à-dire de l'ensemble des chemins qui peuvent être suivis par le logiciel durant son exécution –. Pour des raisons combinatoires, une génération exhaustive des données – d'une taille donnée – est impossible en pratique. La plupart des approches utilisent donc soit des techniques aléatoires, soit des techniques destinées à satisfaire un critère de couverture donné. Dans cette partie nous montrons comment combiner ces deux techniques, en biaisant la génération aléatoire afin d'optimiser la probabilité de satisfaire un critère de couverture donné.

7

Contexte et problématiques

7.1/ Le test logiciel

7.1.1/ Test automatique

La production de logiciels sûrs est une question centrale dans le domaine du génie logiciel. Le test est une étape incontournable pour assurer la qualité d'un logiciel. On peut distinguer le test *statique* qui consiste à analyser le modèle et/ou les spécifications sans exécuter le programme, du test *dynamique* qui consiste à exécuter les tests puis à évaluer le résultat. Pour s'affranchir des limitations du test manuel sans compromettre la qualité des logiciels, la recherche s'intéresse beaucoup à l'automatisation des différentes étapes du test, à savoir :

1. générer les tests,
2. définir l'*oracle* (c'est-à-dire la réponse attendue) pour chacun des test,
3. *jouer* les tests (c'est-à-dire les exécuter) sur le système,
4. comparer le résultat d'exécution de chaque test, avec son oracle.

Afin de présenter les différents types de tests, nous nous sommes appuyés sur [Gau14], ainsi que sur la classification proposée par [Tre04, Gra08] selon les trois axes suivants :

1. le niveau de test,
2. les propriétés testées du système,
3. le support de conception des tests.

Les différentes déclinaisons de ces trois axes sont représentées dans la figure 7.1, et présentées ci-dessous :

Les niveaux de test. Le test unitaire consiste à chercher les erreurs dans une partie précise d'un logiciel ou d'une portion d'un programme – appelée *unité* –. Le test de composant est semblable au test unitaire mis à part le fait que les unités testées – appelées *composants* ou encore *modules* – sont des ensembles d'unités plus petites. Le test d'intégration se concentre sur les interfaces entre les différentes unités. Et le test système consiste à s'assurer que le système global respecte les spécifications initiales.

Les propriétés testées. Le test fonctionnel (appelé également test de comportement) consiste à tester le respect des spécifications fonctionnelles. Le test de robustesse (stress test) consiste à évaluer le fonctionnement du système dans des conditions extrêmes, ou non prévues par la spécification fonctionnelle. Le test de performance sert à mesurer les performances du système (souvent des

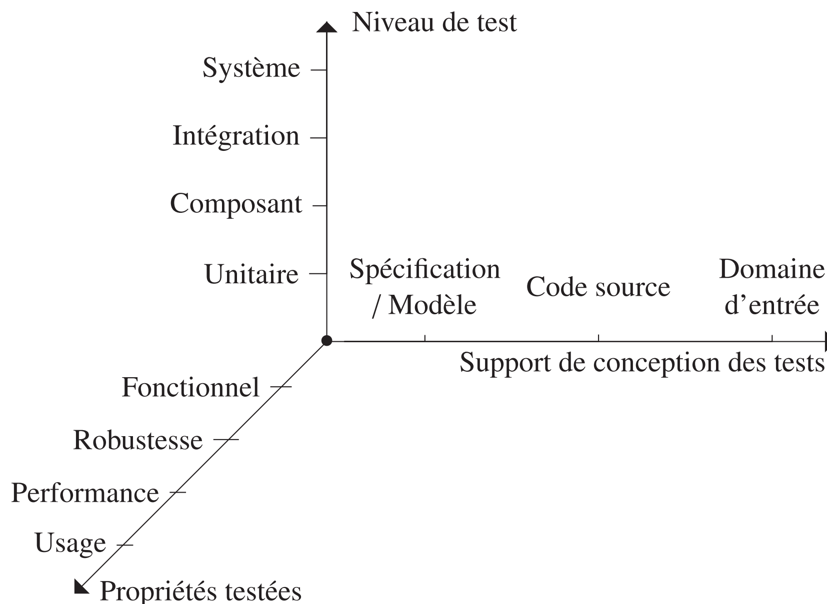


FIGURE 7.1 – Classification des différents types de tests

temps de réponse). Il est souvent utilisé afin de comparer les performances de deux versions d'un même logiciel (test de non-régression des performances), ou de deux logiciels différents (benchmark). Le test d'usage (ou d'utilisabilité) permet d'évaluer la facilité d'utilisation du système par un utilisateur (ergonomie).

Les supports utilisés. Lorsque le code source du logiciel est connu – on parle dans ce cas de test en *boîte blanche* –, on peut l'utiliser afin de concevoir les tests. Lorsqu'il n'est pas connu – on parle dans ce cas de test en *boîte noire* –, la conception des tests s'appuiera sur la spécification/le modèle, ou sur le domaine d'entrée. Le domaine d'entrée étant le plus souvent infini, il existe deux approches principales pour obtenir un jeu de tests fini : le tirage aléatoire et la décomposition en sous-domaines. Les approches centrées sur le domaine d'entrée sont souvent combinées avec d'autres, car on utilise le code ou la spécification pour guider le tirage / la décomposition. Le test à partir de la spécification/du modèle est décrit dans la sous-section suivante.

7.1.2/ TEST À PARTIR DE MODÈLES (MBT)

Le test de systèmes à partir de modèles à états finis [LY96, Bei95] – abrégé en *MBT*, pour l'anglais Model-Based Testing – consiste à décrire le système sous la forme d'un système de transitions étiquetées, sur lequel différents algorithmes peuvent être utilisés afin d'extraire des cas de tests. C'est par exemple le principe de SpecExplorer [CGN+05], ou encore de TGV [JJ04]. Le test à partir de *graphe de flot de contrôle* [GBR98] – qui est une modélisation sous forme de graphe de tous les chemins qui peuvent être suivis par le programme durant son exécution –, a fait l'objet de nombreux travaux de recherche. Pour plus d'informations, le lecteur intéressé peut se reporter au livre de référence [AO08], ou à la plus récente taxonomie du MBT [UPL12].

7.1.3/ CRITÈRES DE COUVERTURE

Comme l'exploration de toutes les configurations d'un logiciel est la plupart du temps impossible, l'un des principaux problèmes de l'ingénieur validation est de trouver une suite de tests pertinente,

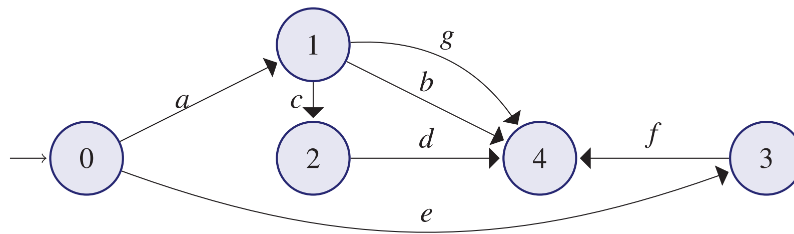


Figure 7.2 – Exemple de graphe fini

tout en contrôlant le nombre de tests. La question cruciale qui se pose alors est : Qu'entend-on par "pertinent" ? Une réponse fréquente dans la littérature et dans la pratique, est de considérer une suite de tests comme pertinente si elle répond à certains critères de couverture. Il existe différents types de critères de couverture, comme :

- la couverture du code source (tests structurels),
- la couverture du modèle/des spécifications (tests fonctionnels),
- la détection de *mutants* (tests par mutations).

La couverture du code source se situe dans le cadre du test par boîte blanche. Les tests fonctionnels sont couramment utilisés, notamment pour les tests de type boîte noire [Bei95]. Les *mutants* sont des erreurs qu'on a volontairement introduites dans le programme, afin d'évaluer la capacité de la suite de tests à les détecter.

Comme il y a de nombreuses façons de satisfaire les critères de couverture [OXL99], d'autres critères peuvent être pris en compte, par exemple s'appuyant sur le calcul de suites de test de longueur minimale / maximale, ou sur la sélection de valeurs aux bornes ou de valeurs aléatoires pour les données de test.

7.1.4/ Test aléatoire

Le test aléatoire est une approche naturelle, connue pour permettre la détection de nombreux types de bogues. Cependant, par définition, les comportements qui ont une faible probabilité de se produire ont également peu de chance d'être testés par ce moyen. En revanche, le test non aléatoire a tendance à se concentrer sur quelques cas précis qui présentent un intérêt particulier pour le testeur, au détriment des cas auxquels le testeur peut ne pas avoir pensé. En effet, le test non aléatoire peut couvrir différents comportements, mais le choix de ces comportements dépend des priorités du testeur et n'est généralement testé qu'une seule fois.

Des méthodes de génération de tests aléatoires – initialement proposées dans [DN81, Ham94] – sont souvent utilisées pour des raisons pratiques, pour générer des données de tests comme dans DART [GKS05], ou pour générer des séquences complètes de tests comme dans l'outil Jartege [Ori05]. Une autre approche combinant du test à partir de modèles et aléatoire est présentée dans un papier écrit par F. Dadeau et al. [DLH09]. Il existe également un travail relativement récent [DGG+12], qui propose une approche combinant test aléatoire et model-checking, sur lequel nous reviendrons plus en détail car nos contributions sont dans sa continuité.

7.1.5/ Génération aléatoire-uniforme d'un chemin dans un graphe fini

Dans [GJ08] les auteurs montrent comment exploiter des méthodes de génération uniforme de chemins dans des graphes finis, afin de générer des tests à partir du graphe de flot de contrôle de programmes C. L'idée est de définir des critères de couverture probabilistes pour les tests, et la méthode consiste à appliquer un algorithme combinatoire afin de résoudre le problème probabiliste

suivant :

Problème 7.1. Génération aléatoire d'un chemin dans un graphe fini

Entrée : Un graphe étiqueté fini $\mathcal{G}$, sommet v_0, un entier positif n

Question : Générer aléatoirement un chemin de taille n dans $\mathcal{G}$, qui part de v_0 ? La génération doit être uniforme vis-à-vis de l'ensemble des chemins de taille n dans $\mathcal{G}$.

Ce problème est un cas particulier de la génération d'un chemin dans une grammaire. La méthode exposée dans [DGG+12] (et issue de [FZC94, FS08]) pour traiter ce problème nécessite deux étapes :

1. Dans un premier temps, le nombre de chemins de taille $i \leq n$ entre chaque paire de sommets est calculé récursivement à l'aide de méthode combinatoires. Il peut être facilement calculé à l'aide du schéma récursif suivant :

   Si on note $s_i(p,q)$ le nombre de chemins de longueur i de p à q, et $\alpha(p,q)$ le nombre d'arêtes de p à q, on a :
   — $s_1(p,q) = \alpha(p,q)$,
   — $s_{i+1}(p,q) = \sum \alpha(p,r) \times s_i(r,q)$, où la somme est prise pour tous les sommets r.
2. Dans un second temps, le chemin aléatoire est généré de manière récursive en utilisant les probabilités calculées : Pour générer un chemin de longueur n de p à q, la probabilité que le second sommet visité par le chemin (après p) soit r est $\alpha(p,r) \times \frac{s_{n-1}(r,q)}{s_n(p,q)}$.

La première arête est choisie au hasard, uniformément parmi les arêtes qui vont de p vers r (s'il y en a plusieurs), puis on génère un chemin de r vers q de taille $n-1$ en utilisant la même méthode. Par exemple, pour le graphe étiqueté représenté sur la figure 7.2, pour générer un chemin de longueur 2 de 0 vers 4, la probabilité de choisir 3 comme second sommet sera $\frac{s_1(3,4)}{s_2(0,4)} = \frac{1}{3}$. La probabilité de choisir 1 comme second sommet sera $\frac{s_1(1,4)}{s_2(0,4)} = \frac{2}{3}$. La probabilité de choisir 2 ou 4 ou 0 est nulle. Ensuite, si 3 est choisi, le chemin généré sera forcément $(0,e,3)(3,f,4)$. Ce chemin a donc une probabilité de $1/3$ d'être généré. Si c'est 1 qui est choisi comme second sommet, la dernière arête est choisie au hasard, uniformément parmi $(1,b,4)$ et $(1,g,4)$. Les chemins $(0,a,1)(1,b,4)$ et $(0,a,1)(1,g,4)$ on donc chacun une probabilité de $1/3$ d'être générés. Tous les chemins de longueur 2 ont donc bien la même probabilité d'être générés.

7.1.6/ TEST À PARTIR DE MODÈLES À PILE

Test à partir de grammaires. La génération de tests à partir de grammaires est fréquemment utilisée pour générer des entrées structurées [McK97, HC83, Mau92, HN11], comme dans [Pur72] pour tester des parseurs, ou dans [DDGM07] pour tester des *re-factoring engines* (logiciels de transformation de programmes). Les approches combinatoires systématiques (*cf.* [CL05]) conduisant à un très, voire trop, grand nombre de séquences, les approches symboliques leur sont souvent préférées (*cf.* [LS06, MX07]). A noter que ces algorithmes ont été également utilisés dans un ouvrage de P. Godefroid et al. [GKL08], pour du test à données aléatoires par boîte blanche (white-box fuzzing). Notre contribution par rapport à ces approches est de combiner la génération de tests aléatoires à partir de grammaires avec l'exploitation de critères de couverture. Dans [XZC10], un outil générique pour générer des données de tests à partir de grammaires est proposé. Cet outil ne fournit pas de fonction aléatoire mais est basé sur des algorithmes et des techniques de couverture de règles, telle que définie dans [Pur72, Läm01, ZW09, AV08].

Génération aléatoire-uniforme d'arbres de dérivation. Des techniques combinatoires pour générer aléatoire-uniformément des arbres d'exécution à partir de grammaires algébriques ont été développées par Flajolet et al. [FZC94, FS08]. Plusieurs outils existants comme GenRgenS [PTD06]

ou le paquetage CS de Mupad [DDZ98] peuvent être utilisés à cet effet. Pour les expérimentations présentées dans la section 9.3, nous avons utilisé GenRgenS, qui est un logiciel de génération aléatoire de séquences, qui supporte plusieurs types de modèles comme les chaînes de Markov, les grammaires algébriques, etc.

7.2/ Où se situe-t-on ?

7.2.1/ Génération aléatoire de chemins contre parcours aléatoire

Cette sous-section explique la différence entre une marche aléatoire isotrope, et une génération aléatoire-uniforme des chemins de longueur donnée.

La première façon qui vient habituellement à l'esprit, pour générer aléatoirement des chemins, est d'effectuer un parcours classique – appelé *marche aléatoire isotrope*, ou simplement *marche aléatoire*, ou encore *parcours aléatoire de Markov* – : À chaque étape la prochaine transition est choisie aléatoire-uniformément parmi les transitions sortantes du sommet actuel, indépendamment des tirages précédents. Cette technique conduit à une distribution inconnue (difficile à caractériser) de chemins de longueur n et est très sensible à la topologie du graphe.

La deuxième méthode, qui permet de générer des chemins de manière uniforme, consiste à choisir la transition suivante aléatoirement mais pas uniformément : cette fois, la probabilité de chaque transition est proportionnelle au nombre de chemins différents qui pourront être générés si on choisit cette transition.

Considérons par exemple la génération d'un chemin de longueur 3 à partir de l'état 0, dans le graphe étiqueté représenté sur la figure 7.3 :

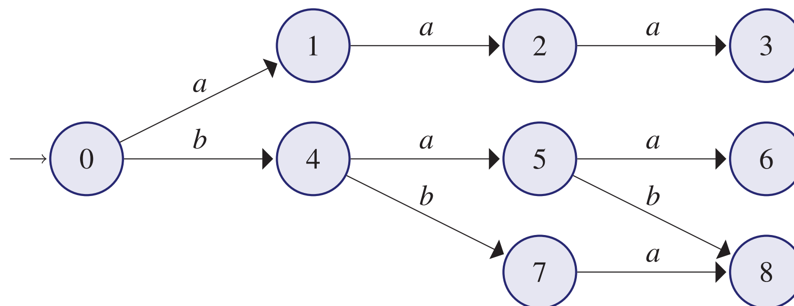


Figure 7.3 – Exemple 1

- En utilisant une approche de type Markov : le chemin *aaa* a une probabilité de $1/2$ d'être généré, les chemins *baa* et *bab* ont chacun une probabilité de $1/8$ d'être générés, et le chemin *bba* a une probabilité de $1/4$ d'être généré. On peut donc constater que la génération de chemins n'est pas uniforme. La couverture globale des états est également mauvaise, puisque les états 1, 2 et 3 ont chacun une chance sur deux d'être couverts (par le chemin *aaa*), contre une chance sur huit pour l'état 6 (qui est couvert uniquement lorsque le chemin *baa* est généré).
- Pour la génération uniforme de chemin : puisqu'à partir de l'état 0, le choix de la lettre *a* permet de générer un chemin de longueur 3 (*aaa*) et que la lettre *b* permet de générer trois chemins (*baa*, *bab* et *bba*), on donnera à la lettre *a* une probabilité de $1/4$ d'être choisie, et une probabilité de $3/4$ à la lettre *b*. Et puisqu'à partir de l'état 4, le choix de la lettre *a* permet de générer deux chemins de longueur 2 (*aa* et *ab*) et que la lettre *b* permet de générer un chemin (*ba*), on donnera à la lettre *a* une probabilité de $2/3$ d'être choisie, et une probabilité de $1/3$ à la lettre *b*. Ainsi, chacun des chemins de longueur 3 (*aaa*, *baa*,

bab et *bba*) a la même probabilité d'être généré (= 1/4). Sans être uniforme, la couverture globale des états est également meilleure.

Considérons maintenant la génération d'un chemin de longueur 3 à partir de l'état 0, dans le graphe étiqueté représenté sur la figure 7.4 :

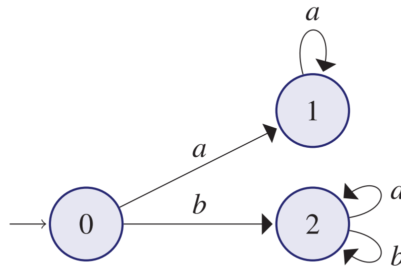


FIGURE 7.4 – Exemple 2

- En utilisant une approche de Markov : puisqu'à partir de l'état 0, la lettre a et la lettre b ont la même probabilité d'être choisies, l'état 1 et l'état 2 ont chacun la même probabilité (= 1/2) d'être parcourus par le chemin généré.
- Pour la génération uniforme de chemin : puisqu'à partir de l'état 0, le choix de la lettre a permet de générer un chemin (*aaa*) alors que la lettre b permet de générer quatre chemins (*baa*,*bab*,*bba* et *bbb*), on donnera à la lettre a (pour la première transition) une probabilité de 1/5 d'être choisie, et une probabilité de 4/5 à la lettre b (toujours pour la première transition). Dans ce cas, la génération uniforme de chemin donnera une mauvaise couverture globale des états, car l'état 2 sera sur-représenté.

Nous avons choisi la génération aléatoire uniforme de chemins, car elle a le précieux avantage d'offrir la possibilité de biaiser la distribution, afin d'optimiser la couverture du graphe (comme on l'explique dans la sous-section suivante).

7.2.2/ COMBINER GÉNÉRATION ALÉATOIRE ET CRITÈRES DE COUVERTURE

Dans le contexte du test logiciel, la génération aléatoire a l'avantage de fournir rapidement de nombreuses données de test différentes, pour chaque comportement du système. De plus, ces données de test sont indépendantes des choix de l'ingénieur de test, et peuvent par conséquent permettre de détecter des problèmes auxquels on ne s'attendait pas. Par exemple, le *fuzzing* (test à données aléatoires) est particulièrement pertinent pour tester des exigences de sécurité [GKL08]. Cependant, des tests aléatoires risquent de manquer un comportement important s'il se produit avec une probabilité très faible. Pour corriger cet inconvénient tout en gardant les avantages cités ci-dessus, une solution consiste à combiner la génération aléatoire avec des critères de couverture.

Problème 7.2. Combiner la génération aléatoire avec un critère de couverture

Entrée : Un modèle $\mathcal{M}$, un critère de couverture C, un entier positif n Question : Générer aléatoirement des chemins de taille n dans $\mathcal{M}$, en optimisant la probabilité de couvrir C ?

Qualité d'une méthode de génération aléatoire. D'après [TF89] et [DGG+12], on peut définir la qualité d'une technique de test aléatoire en fonction d'un critère de couverture C, comme la probabilité $q_{C,N}$ minimale de couvrir au moins une fois un des éléments de C, en générant N tests (de taille n). Lorsque les tests sont indépendants les uns des autres, on a $q_{C,N} = 1 - (1 - q_{C,1})^N$;

qui correspond à la probabilité de *rater* N fois l'élément qu'on a le moins de chance de couvrir. Par conséquent, le calcul ou l'estimation de $q_{C,1}$ est une question centrale pour déterminer a priori le potentiel de couverture d'une approche aléatoire. Avec une approche consistant à générer des tests jusqu'à ce que tous les éléments de C aient été couverts, le nombre moyen de tests requis est borné par $\frac{|C|}{q_{C,1}}$. Puisque les probabilités de couvrir différents éléments en générant une trace ne sont pas indépendantes, on ne peut en général pas calculer de prévision exacte du nombre de tests nécessaires pour couvrir au moins une fois chaque élément de C.

Pour tout élément $e \in C$, on note $p_{e,n}$ la probabilité qu'un test généré de taille n couvre e. On peut facilement vérifier qu'en générant N données de test indépendamment de C (algorithme 1) (figure 7.5), on a $q_{C,N} = 1 - (1 - p_{\min})^N$, où $p_{\min} = \min_{e \in C}\{p_{e,n}\}$.

Algorithme 1 - Génération aléatoire-uniforme

1. On génère aléatoirement une exécution dans $\mathcal{M}$.
2. S'il existe un élément de $\mathcal{M}$ qui n'a pas été couvert par les exécutions précédemment générées, on retourne à l'étape 1.

Figure 7.5 – Génération aléatoire - Algorithme 1

Afin de combiner test aléatoire et critère de couverture, une approche naturelle est décrite dans l'algorithme 2 (figure 7.6). Avec cette approche, le nombre d'essais est limité par le nombre d'états.

Algorithme 2 - Approche naturelle pour combiner test aléatoire et critère de couverture

1. Pour tout élément e du critère de couverture C, on calcule la probabilité $p_{e,n}$ qu'un test de taille n généré aléatoirement visite e.
2. Si certaine probabilités $p_{e,n}$ sont égales à 0, alors il n'est pas possible de couvrir tous les états en générant des exécutions de longueur n.
3. Sinon, on choisit aléatoirement un élément e qui n'a pas été couvert par les exécutions précédemment générées (s'il en existe un).
4. On génère aléatoirement une exécution dans $\mathcal{M}$ qui visite e.
5. S'il existe un élément de $\mathcal{M}$ qui n'a pas été couvert par les exécutions précédemment générées, on retourne à l'étape 3.

Figure 7.6 – Génération aléatoire - Algorithme 2

Mais l'espace d'états peut être très grand et donc causer des problèmes lorsque les tests sont difficiles à exécuter, par exemple lorsque jouer les tests nécessite des manipulations physiques. Dans ce cas, la procédure ci-dessus peut être arrêtée après un nombre fixé N de tests générés. Cette procédure nécessite de savoir générer uniformément un test de taille n qui couvre un élément donné.

Optimisation de la qualité. D'après les travaux de A.Denise et al. [DGG+12, Section 5.2], il est possible d'optimiser la qualité attendue des suites de tests en choisissant l'élément e (à l'étape 3) avec une distribution non uniforme. Le schéma général de cette combinaison est le suivant : Si on considère un algorithme de génération aléatoire de données de test de taille n, et un critère de couverture un ensemble d'éléments C, le but est d'utiliser l'algorithme de génération N fois afin d'optimiser la probabilité de couvrir tous les éléments de C. Un moyen plus efficace que de générer N données indépendamment de C, est de répéter N fois la procédure décrite dans l'algorithme 3 (figure 7.7).

Algorithme 3 - Approche aléatoire-uniforme, biaisée afin d'optimiser la probabilité de couvrir C

1. Choisir aléatoirement un élément $e \in C$ avec une probabilité π_e, et
2. Générer uniformément un test de taille n qui couvre e.

FIGURE 7.7 – Génération aléatoire - Algorithme 3

En plus de savoir générer uniformément un test de taille n qui couvre un élément donné, cette procédure nécessite également de choisir les probabilités π_e pour optimiser la probabilité de couvrir tous les éléments de C. Comme le montre [DGG+12], les probabilités π_e peuvent être calculées en résolvant le système à contraintes de la figure 7.8.

maximiser p qui satisfait

$$\begin{cases} p \leq \sum_{e \in C} \pi_e \frac{p_{e,f,n}}{p_{e,n}} \text{ for all } f \in C \\ \sum_{e \in C} \pi_e = 1 \end{cases}$$

où $p_{e,f,n}$ est la probabilité qu'une donnée de test de taille n et générée aléatoirement, couvre à la fois e et f.

FIGURE 7.8 – Calcul des probabilités π_e, afin d'optimiser la probabilité de couvrir C

Ce problème de programmation linéaire peut aisément être résolu en utilisant des approches basées sur l'algorithme du simplexe (et disponible dans de nombreux outils très efficaces comme `lp_solve` ou `CPLEX`).

En résumé, pour combiner génération aléatoire et critères de couverture (problème 7.2), on doit résoudre un système de contraintes et on doit savoir :

1. générer aléatoirement un test d'une taille donnée qui couvre un élément donné,
2. calculer les probabilités $p_{e,n}$ qu'un test de taille n couvre un élément e, quel que soit e de C, et
3. calculer les probabilités $p_{e,f,n}$ qu'un test de taille n couvre à la fois e et f , quels que soient e et f de C.

7.2.3/ MODÈLES RÉGULIERS (SANS PILE) CONTRE MODÈLES ALGÉBRIQUES (À PILE)

Cette sous-section explique pourquoi les modèles à pile sont mieux que les graphes sans pile pour l'étape de concrétisation.

Dans [DGG+12], il est expliqué comment biaiser une approche de test aléatoire uniforme en utilisant des contraintes données par un critère de couverture, afin d'optimiser la probabilité de satisfaire ce critère. La méthode est développée pour la production de chemins dans un graphe fini, or un graphe fini représente souvent une abstraction forte du système sous test. Il y a donc un risque que de nombreux tests abstraits ne soient pas concrétisables (c'est-à-dire jouables sur l'implémentation).

Considérons par exemple le programme récursif en C qui est décrit dans la figure 7.9, et qui calcule x^n : le graphe de flot de contrôle de ce programme est décrit dans la figure 7.10. Le graphe de flot de contrôle de gauche ne tient pas compte de l'appel récursif à la fonction `power`. Dans celui de droite, les appels à la fonction *power* sont représentés par des flèches pointillées, étiquetées soit par *call(power)* pour un appel à la fonction, soit par *return(power)* pour un retour de la fonction.

Dans le graphe de gauche, comme les appels à `power` sont ignorés, il est impossible de calculer arbitrairement de longs chemins. Alors que dans le graphe de droite, c'est tout à fait possible.

```
int power(float x, int n){
   int res;
   if (n==0) {
      return 1;
   } else {
      res = power(x,n/2);
      if (n%2==0) {
         return res*res;
      } else {
         return res*res*x;
      }
   }
}
```

FIGURE 7.9 – Programme en C, qui calcule x^n

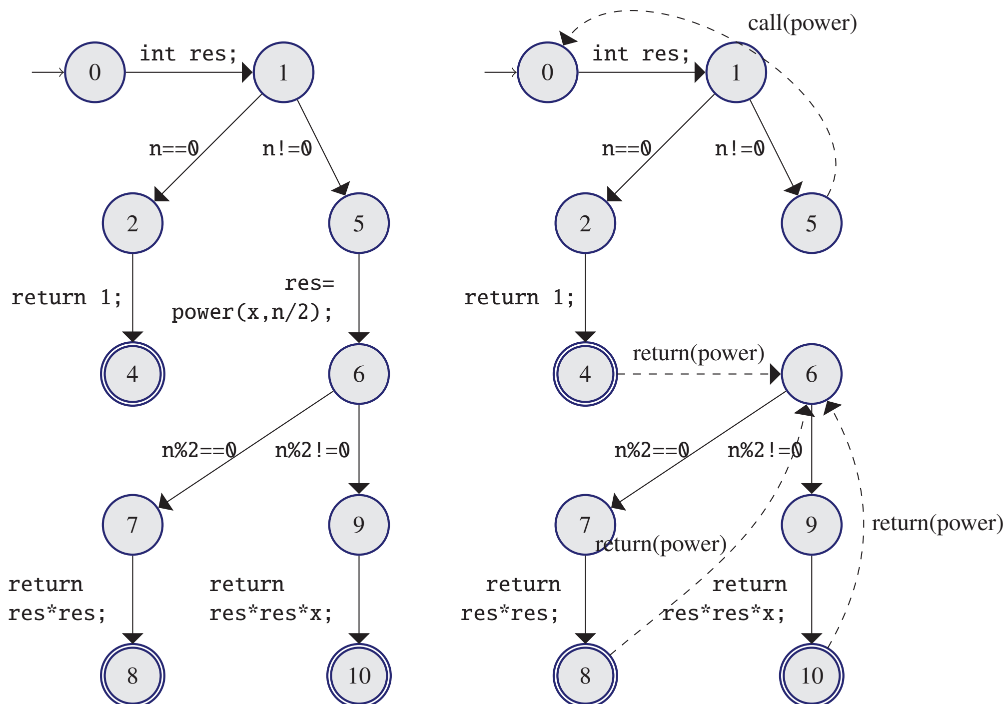


FIGURE 7.10 – Graphe de flot de contrôle de x^n

Exemple 7.3. Exécution de `power(3,2)` *L'exécution de* `power(3,2)` *correspond au chemin*

$$0 \rightarrow 1 \rightarrow 5 \dashrightarrow 0 \rightarrow 1 \rightarrow 5 \dashrightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4 \dashrightarrow 6 \rightarrow 9 \rightarrow 10 \dashrightarrow 6 \rightarrow 7 \rightarrow 8$$

La première occurrence de $5 \dashrightarrow 0$ *correspond à l'appel à* `power(3,1)`*. La seconde occurrence de* $5 \dashrightarrow 0$ *correspond à l'appel à* `power(3,0)`*. La transition* $4 \dashrightarrow 6$ *correspond au retour de* `power(3,0)`*, et la transition* $10 \dashrightarrow 6$ *correspond au retour de* `power(3,1)`*.*

Si on prend a pour l'instruction `int res;`, b pour `n==0`, c pour `n!=0`, d pour *call(power)*, e pour `return 1;`, f pour *return(power)*, g pour `n%2==0`, h pour `n%2!=0`, i pour `return res*res;`,

et j pour `return res*res*x;` : les mots acceptés par l'automate de droite de la figure 7.10 (*cf.* également la figure 2.14), sont ceux du langage $[(acd)^*abef(gif+hjf)^*(gi+hj)]+(acd)^*abe$ (si on considère les chemins qui vont de l'état 0 à un état final). De plus, les mots acceptés qui correspondent aux occurrences correctes [1] des *call(power)* et des *return(power)*, sont ceux décrits par $[(acd)^k abef(gif+hjf)^{k-1}(gi+hj)]+abe$, avec $k \geq 1$. Par exemple le chemin $0 \rightarrow 1 \rightarrow 5 \dashrightarrow 0 \rightarrow 1 \rightarrow 2 \rightarrow 4$, qui correspond à *acdabe*, est un chemin réussi – appelé aussi une DFA-trace – dans l'automate qui ne peut pourtant pas être concrétisé puisque l'appel à la fonction `power` (via la lettre d) n'est jamais retourné. C'est pourquoi la probabilité qu'un chemin de longueur n de l'automate corresponde à un chemin cohérent vis-à-vis des appels/retours de la fonction `power` est en $o(\frac{1}{2^{n/6}})$, et converge vers 0 lorsque n tend vers $+\infty$. Ceci implique que l'approche qui consiste à générer aléatoire-uniformément des chemins dans l'automate fini générera la plupart du temps des tests non concrétisables. Cela exclut donc la possibilité d'utiliser une approche de rejet – qui consiste à générer des DFA-traces jusqu'à ce qu'elles correspondent à des chemins cohérents vis-à-vis des appels/retours de fonctions – pour générer de longs cas de tests.

7.2.4/ Contribution : Combiner génération aléatoire et critères de couverture, dans des modèles à pile

Dans le papier [EDGB12], les auteurs combinent génération de tests aléatoires à partir de grammaires et critères de couverture, en utilisant un parcours aléatoire isotrope. Notre méthode traite également le problème de combinaison de la génération aléatoire avec des critères de couverture, dans des modèles à pile, mais notre algorithme de génération de chemins est aléatoire-uniforme.

Nos contributions consistent à enrichir l'approche de [DGG$^+$12] – qui combine de la génération aléatoire uniforme avec des critères de couverture, sur des graphes finis – afin de l'appliquer sur des modèles algébriques (des grammaires algébriques dans le chapitre 8, et des automates à pile dans le chapitre 9), ce qui permet de réduire le degré d'abstraction du modèle et donc de générer moins de tests non concrétisables.

1. C'est-à-dire que le mot est cohérent avec les appels à la fonction `power`, comme une expression correctement parenthésée.

8

Contribution : Application sur les grammaires

Les grammaires sont un modèle très utilisé en informatique, notamment dans les compilateurs et dans les parseurs, ou encore pour décrire le format de fichiers XML (DTD). On peut également citer le langage des expressions arithmétiques, qui est couramment défini sous la forme d'une grammaire. Les grammaires sont souvent utilisées afin de définir le format des données acceptées en entrée d'un programme, comme l'outil Seed [HN11] qui permet de générer aléatoire-uniformément des structures de données récursives, dont le format est défini par des règles de grammaire.

Ce chapitre est consacré au problème de la génération d'arbres d'exécution d'une grammaire, avec le critère de couverture *Tous les symboles non-terminaux*. Plus précisément, soit une grammaire $G = (\Sigma, \Gamma, S_0, R)$, le critère de couverture étant Γ, un test de taille n étant un arbre de dérivation de G de taille n, on dit que $X \in \Gamma$ est couvert par un test si ce test – donc cet arbre de dérivation – couvre X. Nous réutiliserons les notions et notations définies dans le chapitre 2, ainsi que la méthode d'optimisation du test aléatoire pour satisfaire un critère de couverture donné, qui est décrite dans la section 7.2.2. Notre contribution est présentée dans la section 8.1, qui explique comment optimiser la couverture de symboles non-terminaux dans la génération de tests à partir de grammaire. La section 8.3 illustre cette méthode par un exemple.

Combiner génération aléatoire et critères de couverture, dans une grammaire

Problème 8.1. Combiner la génération aléatoire avec un critère de couverture dans une grammaire

Entrée : Une grammaire $\mathcal{G}$, un critère de couverture C, un entier positif n

Question : Générer aléatoire-uniformément des arbres de dérivation de taille n dans $\mathcal{G}$, en optimisant la probabilité de couvrir C ?

Le problème 8.1 est une spécification du problème 7.2, appliqué aux grammaires. Pour le résoudre, on doit donc savoir :

1. générer aléatoirement un arbre de dérivation d'une taille donnée qui couvre un élément donné,
2. calculer les probabilités $p_{e,n}$ qu'un test de taille n couvre un élément e, quel que soit e de C, et
3. calculer les probabilités $p_{e,f,n}$ qu'un test de taille n couvre à la fois e et f, quels que soient e et f de C.

8.1/ CALCULER $p_{X,n}$ ET $p_{X,Y,n}$

Dans cette section on explique comment calculer la probabilité $p_{X,n}$ qu'un arbre de dérivation de taille n généré aléatoirement couvre X, et la probabilité $p_{X,Y,n}$ qu'un arbre de dérivation de taille n généré aléatoirement couvre à la fois X et Y.

Soit $G = (\Sigma, \Gamma, S_0, R)$ une grammaire algébrique. On note $E_n(G)$ l'ensemble des arbres de dérivation de taille n de G. On note $E_{X,n}(G)$ l'ensemble des arbres de dérivation de taille n de G qui couvrent X, et $E_{X,Y,n}(G)$ l'ensemble des arbres de dérivation de taille n de G qui couvrent à la fois X et Y. De façon similaire, on note $E_{X\to u,n}(G)$ l'ensemble des arbres de dérivation de taille n de G qui couvrent la règle $X \to u$, et $E_{X\to u,Y\to v,n}(G)$ l'ensemble des arbres de dérivation de taille n de G qui couvrent à la fois $X \to u$ et $Y \to v$. On peut facilement voir que :

- si $E_n(G)$ est vide alors $p_{X,n} = 0$ [resp. $p_{X,Y,n} = 0$].
- Sinon, $p_{X,n} = \frac{|E_{X,n}(G)|}{|E_n(G)|}$ [resp. $p_{X,Y,n} = \frac{|E_{X,Y,n}(G)|}{|E_n(G)|}$].

Par conséquent, le calcul la probabilité définie dans la section 7.2.2 – nécessaire pour résoudre le programme de contrainte linéaire – revient à calculer la cardinalité des ensembles $E_{X,n}(G)$ et $E_{X,Y,n}(G)$.

8.2/ CALCULER $|E_{X,n}(G)|$ ET $|E_{X,Y,n}(G)|$

Pour calculer $|E_{X,n}(G)|$, on construit une grammaire G_X telle que $E_n(G_X)$ et $E_{X,n}(G)$ sont en bijection (et ont donc le même nombre d'éléments).

Pour tout $u \in (\Gamma \cup \Sigma)^*$, $[u]_0$ est défini récursivement par : $[\varepsilon]_0 = \varepsilon$, $[Zu]_0 = (Z, 0)[u]_0$ (avec $Z \in \Gamma$) et $[au]_0 = a[u]_0$ (avec $a \in \Sigma$). Intuitivement, on obtient $[u]_0$ à partir de u en remplaçant chaque lettre Z de u qui est dans Γ par la paire $(Z, 0)$. Par exemple, pour la grammaire de l'exemple 2.13 (*cf.* figure 2.13), on a $[aSbbT]_0 = a(S, 0)bb(T, 0)$. Pour tout $u \in (\Gamma \cup \Sigma)^*$, $[u]_2$ se construit de la même façon, c'est-à-dire qu'on remplace chaque lettre Z de u qui est dans Γ par la paire $(Z, 2)$.

Pour tout $u \in (\Gamma \cup \Sigma)^*$, $\{u\}_{1,2}$ est l'ensemble des mots $u' \in (\Sigma \cup \Gamma \times \{1, 2\})^*$ obtenus à partir de u en remplaçant chaque lettre Z of Γ par $(Z, 1)$ ou par $(Z, 2)$, avec la restriction qu'au moins une lettre Z de u soit remplacée par $(Z, 1)$. Les lettres de u qui sont dans Σ sont inchangées.

Exemple 8.2. *Si $u = aSbT$, alors $\{u\}_{1,2} = \{a(S, 1)b(T, 1), a(S, 2)b(T, 1), a(S, 1)b(T, 2)\}$.*

Notons que si $u \in \Sigma^*$ alors $\{u\}_{1,2} = \emptyset$ puisque la contrainte qu'au moins une lettre Z de u soit remplacée par $(Z, 1)$ ne peut être satisfaite.

Soit $G_X = (\Sigma, \Gamma \times \{0, 1, 2\}, (S_0, 1), R_X)$ où $R_X = R_0 \cup R_1 \cup R'_1 \cup R_2$ avec :

- $R_0 = \{(Z, 0) \to [u]_0 \mid Z \to u \in R\}$,
- $R_1 = \{(Z, 1) \to u' \mid Z \neq X$ et $\exists Z \to u \in R$ telle que $u' \in \{u\}_{1,2}\}$,
- $R'_1 = \{(X, 1) \to [u]_0 \mid X \to u \in R\}$,
- $R_2 = \{(Z, 2) \to [u]_2 \mid Z \to u \in R$ and $Z \neq X\}$.

Intuitivement, ajouter la valeur 0 à un symbole de Γ signifie que si cette règle est utilisée, alors il existe une occurence de X à une position plus haute de l'arbre de dérivation. Ajouter la valeur 1 à un symbole de Γ signifie qu'il n'existe pas d'occurrence de X à une position plus haute de l'arbre de dérivation, mais qu'il existe une occurrence de X à cette position ou à une position plus basse de l'arbre de dérivation. Et enfin, la valeur 2 signifie qu'il n'existe pas d'occurrence de X à une position plus haute ou plus basse de l'arbre de dérivation.

Exemple 8.3. G_X *Soit* $G = (\{a,b\}, \{S,T,X\}, S, R)$ *une grammaire avec* $R = \{S \rightarrow SS, S \rightarrow aT, S \rightarrow Xb, T \rightarrow aa, X \rightarrow TX, X \rightarrow b\}$. *L'ensemble des règles de la grammaire* G_X *est :* $\{(S,0) \rightarrow (S,0)(S,0), (S,0) \rightarrow a(T,0), (S,0) \rightarrow (X,0)b, (T,0) \rightarrow aa, (X,0) \rightarrow b, (X,0) \rightarrow (T,0)(X,0)\} \cup \{(S,1) \rightarrow (S,1)(S,1), (S,1) \rightarrow (S,1)(S,2), (S,1) \rightarrow (S,2)(S,1), (S,1) \rightarrow a(T,1), (S,1) \rightarrow (X,1)b\} \cup \{(X,1) \rightarrow b, (X,1) \rightarrow (T,0)(X,0)\} \cup \{(S,2) \rightarrow (S,2)(S,2), (S,2) \rightarrow a(T,2), (S,2) \rightarrow (X,2)b, (T,2) \rightarrow aa\}$.

Proposition 8.4 (Bijection) *Il existe une bijection entre* $E_n(G_X)$ *et* $E_{X,n}(G)$.

PREUVE. L'exemple 8.5 illustre certains éléments de cette preuve. Soit φ la fonction de $(\Gamma \times \{0,1,2\} \cup \Sigma)^*$ vers $(\Gamma \cup \Sigma)$ définie inductivement par : $\varphi(\varepsilon) = \varepsilon$, $\varphi(au) = a\varphi(u)$ si $a \in \Sigma \cup \Gamma$, et $\varphi((Z,\alpha)u) = Z\varphi(u)$ si $Z \in \Gamma$ et $\alpha \in \{0,1,2\}$. Intuitivement, φ est une projection qui supprime le second élément (α) des paires $(Z,\alpha) \in \Gamma \times \{0,1,2\}$.

Par construction de G_X, si $(Z,\alpha) \rightarrow u$ est une règle de G_X alors $\varphi((Z,\alpha)) \rightarrow \varphi(u)$ est une règle de G. Donc si $x_0, \ldots, x_k$ est une dérivation complète de G_X, alors $\varphi(x_0), \ldots, \varphi(x_k)$ est une dérivation complète de G. De plus, le symbole initial de G_X est $(S,1)$ et toutes les règles de R_X dont la partie gauche est dans $(\Gamma \setminus \{X\}) \times \{1\}$ ont une partie droite où un élément de $\Gamma \times \{1\}$ apparaît. Comme $x_k \in \Sigma^*$, le seul moyen d'enlever l'élément 1 est donc d'utiliser une règle dont la partie gauche est $(X,1)$. C'est pourquoi l'arbre de dérivation associé à $\varphi(x_0), \ldots, \varphi(x_k)$ couvre forcément X.

Par conséquent, φ induit une fonction de $E_n(G_X)$ vers $E_{X,n}(G)$. Soit $x_0, \ldots, x_k$ et $x'_0, \ldots, x'_k$ des dérivations complètes de G_X, telles que $\varphi(x_0), \ldots, \varphi(x_k) = \varphi(x'_0), \ldots, \varphi(x'_k)$. Si on suppose que $x_0, \ldots, x_k \neq x'_0, \ldots, x'_k$, alors il existe un index minimum i_0 tel que $x_{i_0} \neq x'_{i_0}$. De plus, $x_0 = (S_0, 1) = x'_0$, donc $i_0 \geq 1$, donc $x_{i_0-1} = x'_{i_0-1}$ existe. Si on fixe $x_{i_0-1} = v_0(Z,\alpha)v_1$, avec $Z \in \Gamma$ et $\alpha \in \{0,1,2\}$, alors on se trouve dans l'un des cas suivants :

- Si $\alpha = 0$, alors il existe $Z \rightarrow u$ et $Z \rightarrow u'$ dans R telles que $x_{i_0} = v_0[u]_0v_1$ et $x'_{i_0} = v_0[u']_0v_1$. Comme $\varphi(x_{i_0}) = \varphi(x'_{i_0})$, on peut en déduire que $\varphi([u]_0) = \varphi([u\prime]_0)$. Mais $\varphi([u]_0) = u$ et $\varphi([u']_0) = u'$, ce qui prouve que $x_{i_0} = x'_{i_0}$, ce qui est une contradiction.
- Si $\alpha = 2$, on peut utiliser la même preuve, en remplaçant simplement les 0 par des 2.
- Si $\alpha = 1$ et $Z = X$, on peut à nouveau utiliser la même preuve.
- Si $\alpha = 1$ et $Z \neq X$, alors il existe $Z \rightarrow u$ et $Z \rightarrow u'$ dans R telles que $x_i = v_0u_1v_1$ et $x'_i = v_0u_2v_1$, avec $u_1 \in \{u\}_{1,2}$ et $u_2 \in \{u'\}_{1,2}$. Comme $\varphi(x_{i_0}) = \varphi(x'_{i_0})$, on a $u = u'$, donc $u_1, u_2 \in \{u\}_{1,2}$. Comme $u_1 \neq u_2$, on définit j comme étant la première lettre de u_1 qui est différente de la lettre correspondante (c'est-à-dire à la même position) dans u_2. Par construction de $\{u\}_{1,2}$, j de u_1 et j de u_2 doivent toutes deux appartenir à $\Gamma \times \{1,2\}$. Par exemple : (T,β_1) et (H,β_2). Comme $\varphi((T,\beta_1)) = \varphi((H,\beta_2))$, on a donc $T = H$. Sans perte de généralité, on peut donc supposer que $\beta_1 = 1$ et $\beta_2 = 2$. Par conséquent, x_{i_0} a un préfixe de la forme $v_0(T,1)$: dans l'arbre de dérivation correspondant à $x_0, \ldots, x_k$, le sous-arbre dont la racine est ce $(T,1)$ contient un X (par construction de R_1). Réciproquement, x'_{i_0} a un préfixe de la forme $v_0(T,2)$: dans l'arbre de dérivation correspondant à $x'_0, \ldots, x'_k$, le sous-arbre dont la racine est ce $(T,2)$ ne contient pas de X (par construction de R_2). On peut donc en déduire que les deux dérivations correspondantes ne peuvent pas avoir la même image par φ, ce qui est une contradiction.

Donc φ induit une fonction injective de $E_n(G_X)$ vers $E_{X,n}(G)$.

Soit $y_0, \ldots, y_k$ une dérivation complète de G dont l'arbre t correspondant est dans $E_{X,n}(G)$. Considérons un arbre t' étiqueté dans $\Gamma \times \{0,1,2\} \cup \Sigma$ qui a exactement la même structure (le même ensemble de positions) que t et tel que :

- Si un noeud de t est étiqueté par une lettre de Σ, alors le noeud correspondant (à la même position) dans t' a la même étiquette.
- Si un noeud ρ de t est étiqueté par une lettre $T \in \Gamma$:

- S'il y a au moins un X sur le chemin qui va de la racine jusqu'à ρ, alors le noeud ρ de t' est étiqueté par $(T, 0)$.
- S'il n'y a pas de X sur le chemin qui va de la racine jusqu'à ρ (sauf ρ), et si le sous-arbre dont la racine est ρ contient au moins un X, alors le noeud ρ de t' est étiqueté par $(T, 1)$.
- Sinon – s'il n'y a pas de X sur le chemin qui va de la racine jusqu'à ρ, et si le sous-arbre dont la racine est ρ ne contient pas de X –, le noeud est étiqueté par $(T, 2)$.

On peut vérifier que t' correspond à un arbre de dérivation complet de G_X dont l'image par φ est exactement l'exécution complète correspondant à t, ce qui prouve que φ est surjective, et conclut donc la preuve. □

Exemple 8.5. Illustration de la preuve de la proposition 8.4 *Soit la grammaire* $G = (\{a, b\}, \{S, T, X\}, S, R)$ *de l'exemple 8.3, avec* $R = \{S \to SS, S \to aT, S \to Xb, T \to aa, X, \to T, X \to b\}$. *La figure 8.1 représente l'arbre de dérivation de* $E_{X,19}(G)$, *qui correspond à la dérivation complète suivante :* $S, SS, SSS, aTSS, aaaSS, aaaXbS, aaabbS, aaabbXb, aaabbTXb, aaabbaaXb, aaabbaabb$. *La dérivation correspondante dans* G_X *est* $(S, 1), (S, 1)(S, 1), (S, 2)(S, 1)(S, 1), a(T, 2)(S, 1)(S, 1), aaa(S, 1)(S, 1), aaa(X, 1)b(S, 1), aaabb(S, 1), aaabb(X, 1)b, aaabb(T, 0)(X, 0)b, aaabbaa(X, 0)b, aaabbaabb$, *dont l'arbre de dérivation de* $E_{19}(G_X)$ *est représenté dans la figure 8.2.*

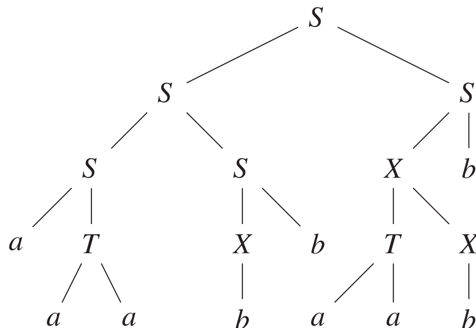


FIGURE 8.1 – Arbre de dérivation de la grammaire G de l'exemple 8.3

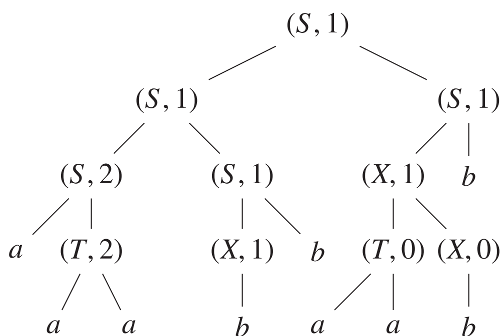


FIGURE 8.2 – Arbre de dérivation de la grammaire G_X de l'exemple 8.3

Grâce à la proposition 8.4 et aux résultats décrits dans dans la section 7.1, on peut calculer $|E_{X,n}(G)|$. Si on note ℓ le nombre maximum d'éléments de Γ (avec la multiplicité) qui apparaissent dans une partie droite de G, alors G_X a $O(2^{\ell}|R|)$ règles, dont les tailles sont limitées par la taille

maximum des règles de G. C'est pourquoi, si ℓ est raisonnable, alors le calcul de $|E_{X,n}(G)|$ est réalisable en pratique, mis à part pour de trop grandes valeurs de n. Comme mentionné ci-dessus, le calcul de $|E_{X,n}(G)|$ donne immédiatement la valeur de $p_{X,n}$. Il est gélament important de noter que G_X permet de générer uniformément des arbres d'exécution de G d'une taille donnée qui couvrent X.

Comme $E_{X,X,n}(G) = E_{X,n}(G)$, le calcul de $|E_{X,X,n}(G)|$ est une application directe des méthodes décrites ci-dessus. Le calcul de $|E_{X,Y,n}(G)|$, avec $Y \neq X$, peut être effectué grâce à une construction quasiment similaire : la différence est que la construction des règles de la grammaire G_{XY}, à partir de la grammaire G_X, doit tenir compte du fait que X et Y doivent tous deux apparaître dans la dérivation. Soit $G_{XY} = (\Sigma, \Gamma \times \{0,1,2\} \times \{0,1,2\}, ((S_0,1),1), R_{XY})$ où $R_{XY} = R_0 \cup R_1 \cup R'_1 \cup R_2$ avec :

- $R_0 = \{((Z,i),0) \to [u]_0 \mid (Z,i) \to u \in R_X\}$,
- $R_1 = \{((Z,i),1) \to u' \mid Z \neq Y$ et $\exists (Z,i) \to u \in R_X$ telle que $u' \in \{u\}_{1,2}\}$,
- $R'_1 = \{((Y,i),1) \to [u]_0 \mid (Y,i) \to u \in R_X\}$,
- $R_2 = \{((Z,i),2) \to [u]_2 \mid (Z,i) \to u \in R_X$ et $Z \neq Y\}$.

Une preuve similaire à celle de la proposition 8.4 permet de démontrer que l'on peut calculer une bijection entre $E_n(G_{XY})$ et $E_{X,Y,n}(G)$. Notons que la taille de G_{XY} est approximativement 4^ℓ fois plus grandes que la taille de G.

8.3/ Expérimentations

Nous avons évalué notre approche sur la grammaire de JSON [1] (pour JavaScript Object Notation), qui est un langage utilisé pour déclarer des objets. Soit la grammaire $G = (\Sigma, \Gamma, Object, R)$ avec Σ composé des huit éléments suivants : $\Sigma = \{, ,\{, :, \}, letter, digit, [,]\}$. L'ensemble Γ des symboles non-terminaux [2] est composé des éléments $''Object''$, $''Members''$, $''Pair''$, $''Array''$, $''Elements''$ and $''Value''$. Et enfin, l'ensemble R est composé des règles suivantes :

- $Object \to \{\} \mid \{Members\}$
- $Members \to Pair \mid Pair, Members$
- $Pair \to letter : Value$
- $Array \to [\ ] \mid [Elements]$
- $Elements \to Value \mid Value, Elements$
- $Value \to letter \mid Object \mid digit \mid Array$

1. http://www.json.org/
2. Pour rendre la spécification plus lisible, la convention qui consiste à utiliser des lettres majuscules pour les symboles non-terminaux n'est pas complètement respectée ici.

Pour optimiser le critère de couverture, on doit résoudre le système suivant : maximiser p qui satisfait

$$\begin{cases}
p \leq \pi_{Object}\frac{p_{Object,Object,n}}{p_{Object,n}} + \pi_{Members}\frac{p_{Members,Object,n}}{p_{Members,n}} \\
\qquad +\pi_{Pair}\frac{p_{Pair,Object,n}}{p_{Pair,n}} + \pi_{Array}\frac{p_{Array,Object,n}}{p_{Array,n}} \\
\qquad +\pi_{Elements}\frac{p_{Elements,Object,n}}{p_{Elements,n}} + \pi_{Value}\frac{p_{Value,Object,n}}{p_{Value,n}} \\
p \leq \pi_{Object}\frac{p_{Object,Members,n}}{p_{Object,n}} + \pi_{Members}\frac{p_{Members,Members,n}}{p_{Members,n}} \\
\qquad +\pi_{Pair}\frac{p_{Pair,Members,n}}{p_{Pair,n}} + \pi_{Array}\frac{p_{Array,Members,n}}{p_{Array,n}} \\
\qquad +\pi_{Elements}\frac{p_{Elements,Members,n}}{p_{Elements,n}} + \pi_{Value}\frac{p_{Value,Members,n}}{p_{Value,n}} \\
p \leq \pi_{Object}\frac{p_{Object,Pair,n}}{p_{Object,n}} + \pi_{Members}\frac{p_{Members,Pair,n}}{p_{Members,n}} \\
\qquad +\pi_{Pair}\frac{p_{Pair,Pair,n}}{p_{Pair,n}} + \pi_{Array}\frac{p_{Array,Pair,n}}{p_{Array,n}} \\
\qquad +\pi_{Elements}\frac{p_{Elements,Pair,n}}{p_{Elements,n}} + \pi_{Value}\frac{p_{Value,Pair,n}}{p_{Value,n}} \\
p \leq \pi_{Object}\frac{p_{Object,Array,n}}{p_{Object,n}} + \pi_{Members}\frac{p_{Members,Array,n}}{p_{Members,n}} \\
\qquad +\pi_{Pair}\frac{p_{Pair,Array,n}}{p_{Pair,n}} + \pi_{Array}\frac{p_{Array,Array,n}}{p_{Array,n}} \\
\qquad +\pi_{Elements}\frac{p_{Elements,Array,n}}{p_{Elements,n}} + \pi_{Value}\frac{p_{Value,Array,n}}{p_{Value,n}} \\
p \leq \pi_{Object}\frac{p_{Object,Elements,n}}{p_{Object,n}} + \pi_{Members}\frac{p_{Members,Elements,n}}{p_{Members,n}} \\
\qquad +\pi_{Pair}\frac{p_{Pair,Elements,n}}{p_{Pair,n}} + \pi_{Array}\frac{p_{Array,Elements,n}}{p_{Array,n}} \\
\qquad +\pi_{Elements}\frac{p_{Elements,Elements,n}}{p_{Elements,n}} + \pi_{Value}\frac{p_{Value,Elements,n}}{p_{Value,n}} \\
p \leq \pi_{Object}\frac{p_{Object,Value,n}}{p_{Object,n}} + \pi_{Members}\frac{p_{Members,Value,n}}{p_{Members,n}} \\
\qquad +\pi_{Pair}\frac{p_{Pair,Value,n}}{p_{Pair,n}} + \pi_{Array}\frac{p_{Array,Value,n}}{p_{Array,n}} \\
\qquad +\pi_{Elements}\frac{p_{Elements,Value,n}}{p_{Elements,n}} + \pi_{Value}\frac{p_{Value,Value,n}}{p_{Value,n}} \\
\pi_{Object} + \pi_{Members} + \pi_{Pair} + \pi_{Array} + \pi_{Elements} + \pi_{Value} = 1
\end{cases}$$

Pour calculer les probabilités $p_{X,n}$ et $p_{X,Y,n}$ pour tout $X, Y \in \Gamma$, nous avons adapté puis utilisé une partie de l'outil Hoa ([End]). Pour $n = 20$, le calcul de l'ensemble des $p_{X,n}$ et $p_{X,Y,n}$ n'a pris que quelques secondes (sur un ordinateur portable Intel Core i5-560M), et on obtient le système suivant : maximiser p qui satisfait

$$\begin{cases}
p \leq \pi_{Object}\frac{12}{12} + \pi_{Members}\frac{12}{12} + \pi_{Pair}\frac{12}{12} + \pi_{Array}\frac{11}{11} \\
\qquad +\pi_{Elements}\frac{8}{8} + \pi_{Value}\frac{12}{12} \\
p \leq \pi_{Object}\frac{12}{12} + \pi_{Members}\frac{12}{12} + \pi_{Pair}\frac{12}{12} + \pi_{Array}\frac{11}{11} \\
\qquad +\pi_{Elements}\frac{8}{8} + \pi_{Value}\frac{12}{12} \\
p \leq \pi_{Object}\frac{12}{12} + \pi_{Members}\frac{12}{12} + \pi_{Pair}\frac{12}{12} + \pi_{Array}\frac{11}{11} \\
\qquad +\pi_{Elements}\frac{8}{8} + \pi_{Value}\frac{12}{12} \\
p \leq \pi_{Object}\frac{11}{12} + \pi_{Members}\frac{11}{12} + \pi_{Pair}\frac{11}{12} + \pi_{Array}\frac{11}{11} \\
\qquad +\pi_{Elements}\frac{8}{8} + \pi_{Value}\frac{11}{12} \\
p \leq \pi_{Object}\frac{8}{12} + \pi_{Members}\frac{8}{12} + \pi_{Pair}\frac{8}{12} + \pi_{Array}\frac{8}{11} \\
\qquad +\pi_{Elements}\frac{8}{8} + \pi_{Value}\frac{8}{12} \\
p \leq \pi_{Object}\frac{12}{12} + \pi_{Members}\frac{12}{12} + \pi_{Pair}\frac{12}{12} + \pi_{Array}\frac{11}{11} \\
\qquad +\pi_{Elements}\frac{8}{8} + \pi_{Value}\frac{12}{12} \\
\pi_{Object} + \pi_{Members} + \pi_{Pair} + \pi_{Array} \\
\qquad +\pi_{Elements} + \pi_{Value} = 1
\end{cases}$$

Le problème de programmation linéaire ci-dessus peut être résolu de façon efficace, à l'aide d'approches basées sur l'algorithme du simplexe. Nous avons utilisé l'outil lp_solve [3], et le résultat est que $p = 1$ si $\pi_{Object} = 0$, $\pi_{Members} = 0$, $\pi_{Pair} = 0$, $\pi_{Array} = 0$, $\pi_{Elements} = 1$, et $\pi_{Value} = 0$. Ce résultat signifie que, pour cet exemple, la meilleure approche pour couvrir l'ensemble des symboles non-terminaux consiste à générer uniquement des arbres de dérivation qui couvrent $Elements$. En effet, dans cette grammaire, la génération d'arbres de dérivation qui couvrent le symbole non-terminal *Elements* produira des arbres qui couvrent également l'ensemble des autres symboles non-terminaux.

3. http://lpsolve.sourceforge.net/

9

Contribution : Application sur les automates à piles

Les automates à pile sont très utilisés en informatique, notamment pour définir des graphes de flot de contrôle qui gèrent les appels/retours de fonctions. Sans cette gestion des appels/retours de fonctions, chaque chemin dans le graphe de flot de contrôle a peu de chances de correspondre à une exécution réelle du système. Les systèmes à pile – qui sont des automates à pile avec uniquement des actions sur la pile ($\Sigma = \{\}$) – sont également très courants.

Nous réutiliserons les notions et notations définies dans le chapitre 2. Les contributions présentées dans ce chapitre de ma thèse sont les suivantes : La première contribution est détaillée dans la section 9.1, et consiste à améliorer l'approche aléatoire proposée dans [DGG+12] en l'étendant aux automates à pile, ce qui permet d'exploiter des abstractions plus fines. Par exemple, les automates à pile sont particulièrement utiles pour coder la pile d'appel d'un programme. La seconde contribution est détaillée dans la section 9.1, et fournit un moyen de résoudre efficacement le problème de la *Génération aléatoire d'un chemin dans un automate à pile déterministe normalisé*, qui est décrit ci-dessous (problème 9.1). La troisième contribution, détaillée dans la section 9.2, concerne la combinaison des critères de couverture le test aléatoire, afin de profiter des avantages de ces deux approches pour évaluer la qualité des suites de tests. Et la quatrième contribution, détaillée dans les sections 9.3 et 9.4, consiste à illustrer l'application des méthodes proposées avec du test structurel, puis avec du test basé sur le modèle.

Génération aléatoire uniforme d'un chemin dans un automate à pile déterministe normalisé

Le problème de la *génération aléatoire uniforme* sur les automates à pile peut s'écrire ainsi :

Problème 9.1. Génération aléatoire d'un chemin dans un automate à pile déterministe normalisé

Entrée : Un automate à pile déterministe normalisé $\mathcal{A}$, l'état initial v_0, un entier positif n

Question : Générer aléatoirement une "NPDA-trace" dans $\mathcal{A}$ de taille n, qui part de v_0 ? La génération doit être uniforme vis-à-vis de toutes les "NPDA-traces" de taille n dans $\mathcal{A}$.

On utilise les abréviations NPDA pour *Normalised Deterministic Pushdown Automaton*, et DFA pour *Deterministic Finite Automaton*. Pour le problème mentionné supra, une "NPDA-trace" est un chemin dans l'automate fini sous-jacent qui correspond aux opérations effectuées sur la pile. La définition formelle est donnée dans la section 2.2.2.

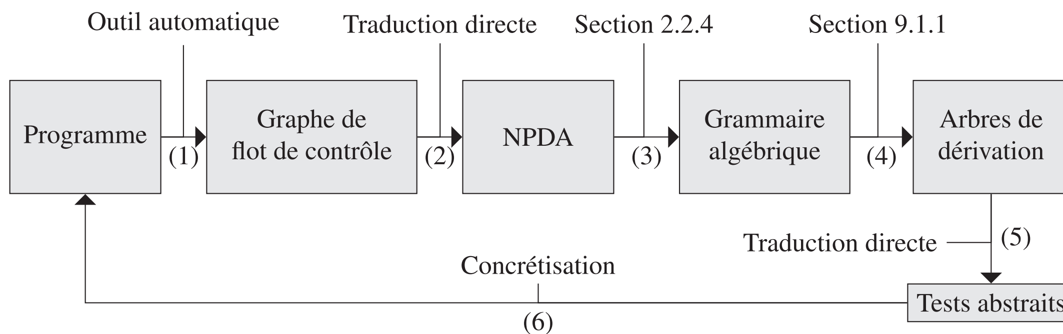


FIGURE 9.1 – Méthode de test aléatoire d'un programme

PRÉSENTATION DE NOTRE APPROCHE

Dans le contexte du *test structurel* – c'est-à-dire lorsque la campagne de tests est produite à partir du code du programme – notre approche pour générer des tests abstraits consiste à :

1. générer le graphe de flot de contrôle (GFC),
2. traduire le GFC en un automate à pile normalisé (NPDA),
3. transformer ce NPDA en une grammaire algébrique (CFG) non ambigue équivalente,
4. générer aléatoire-uniformément des arbres de dérivation de la CFG,
5. traduire ces arbres de dérivation en chemins dans le programme,
6. trouver des valeurs d'entrée du programme, afin que les exécutions correspondent aux chemins générés.

Cette approche est représentée par la figure 9.1. Notons que des outils automatiques existent pour réaliser l'étape (1), à partir de plusieurs langages de programmation fréquemment utilisés (*cf.* [ST12]). L'étape (2) consiste simplement à transformer des appels de fonctions en des opérations dans la pile, et peut aisément être automatisée. L'étape (3) est très basique (*cf.* [Sip96, Section 2.2]). Les tests abstraits obtenus (étape (5) sont des PDA-traces obtenues directement à partir des arbres d'exécution de la grammaire, par la génération/reconnaissance classique des mots par une grammaire. Pour réaliser cette étape, on peut utiliser l'outil GenRgenS [PTD06] ou le paquet CS de Mupad [DDZ98]. Et enfin, la concrétisation (étape (6)) consiste à trouver des valeurs d'entrée du programme, afin que chaque exécution corresponde à une des NPDA-traces générées.

Pour le test à partir de modèle (appelé MBT, pour l'anglais *model-based testing*) – c'est-à-dire lorsque les cas de test sont générés à partir d'un modèle abstrait du système – l'approche est similaire hormis le fait que l'on se passe de l'étape de génération du graphe de flot de contrôle (étape (1)). En effet le programme – ou le système – testé est fourni avec son modèle (dans ce cas un automate à pile). Soulignons le fait que, dans ce contexte, la difficulté de l'étape de concrétisation dépend beaucoup du niveau d'abstraction. Cependant, il s'agit là d'une problématique générale pour toutes les approches de MBT, dont les solutions dépendent de la façon dont le modèle et le système sont liés. Notons que l'étape de concrétisation requiert l'utilisation d'outils spécifiques, comme des solveurs de contraintes, et n'a pas été étudiée dans le cadre de ma thèse qui se concentre sur la façon de générer aléatoirement les tests abstraits.

Le présent document aborde le problème du calcul de NPDA-traces à partir du NPDA (étapes (3), (4) et (5)). La traduction d'un programme en un NPDA (étapes (1) et (2)) et la concrétisation (étape (6)) se situent hors du cadre de ma thèse, et sont seulement illustrées avec quelques exemples. La concrétisation de tests abstraits est un problème théoriquement indécidable, autant dans le cadre

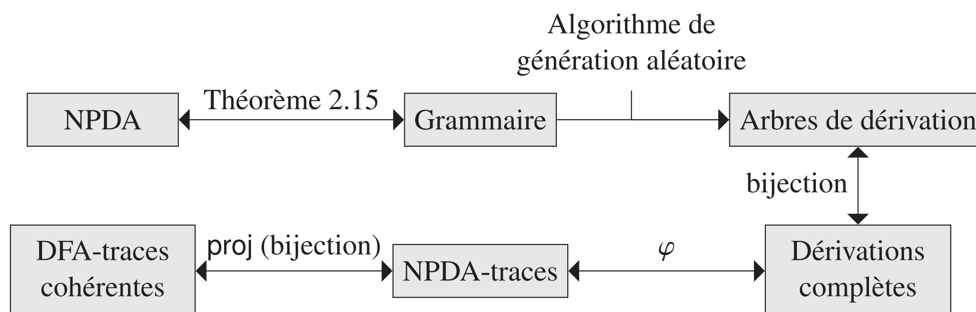


FIGURE 9.2 – Génération aléatoire de DFA-traces cohérentes

du MBT que dans celui du test structurel, et c'est une problématique centrale dans le contexte du test de logiciels [GBR98, GDB05, GKS05, WMMR05].

9.1/ Génération aléatoire-uniforme de DFA-Traces cohérentes

Rappelons que l'objectif principal de cette partie est de générer uniformément, à partir d'un NPDA donné, des traces cohérentes et réussies de longueur fixée dans l'automate fini sous-jacent. Dans la section 2.2.4, on rappelle une connexion bien connue entre les NPDA et les grammaires algébriques. Puis une méthode de génération aléatoire-uniforme de DFA-traces cohérentes est expliquée dans la section 9.1.1.

Notons que dans cette partie, les cas de test sont des DFA-traces cohérentes, ou des NPDA-traces (ce qui est équivalent modulo une bijection triviale).

9.1.1/ Génération aléatoire de DFA-traces cohérentes

Plusieurs algorithmes classiques permettent la transformation d'un NPDA en une grammaire algébrique (*cf.* section 2.2.4). La génération aléatoire de DFA-traces cohérentes d'un NPDA utilise le théorème 2.15. On commence par calculer la grammaire associée G. Puis, on génère aléatoirement des arbres de dérivation : les chemins réussis sont calculés à l'aide de φ. Puisque φ est bijective, si la génération aléatoire d'arbres de dérivation est uniforme, alors la génération aléatoire de DFA-traces cohérentes l'est également (en se servant du fait que proj est bijective). Le schéma général du processus de génération est décrit dans la figure 9.2.

La génération aléatoire d'arbres de dérivation dans une grammaire est décrite dans la section 2.2.3 (problème 2.14).

Considérons par exemple la grammaire décrite dans la figure 2.16. Le calcul prévoit que $T(3) = 1$, $T(9) = 2$, $T(15) = 4$, $T(21) = 8$, et $T(i) = 0$ pour toutes les autres valeurs de i inférieures ou égales à 21. Le résultat $T(9) = 2$ indique qu'il y a deux DFA-traces de taille 9 : $(0, a, 1)(1, c, 5)(5, \text{push}(S), 0)(0, a, 1)(1, b, 2)(2, e, 4)(4, \text{pop}, 6)(6, g, 7)(7, i, 8)$ et $(0, a, 1)(1, c, 5)(5, \text{push}(S), 0)(0, a, 1)(1, b, 2)(2, e, 4)(4, \text{pop}, 6)(6, h, 9)(9, j, 10)$.

À partir d'un NPDA qui a n états, on peut générer k NPDA-traces de taille n en temps $O(n^6 + n^3 k \log(n))$. Notons que cette complexité tient compte du fait que le calcul d'une grammaire à partir d'un PDA nécessite, dans le pire des cas, $\Omega(n^3)$ opérations. En pratique, comme de nombreuses règles de grammaire calculées sont inutiles – c'est-à-dire qu'elles ne peuvent pas être utilisées dans une exécution réussie – il est souvent possible de réduire considérablement la taille de la grammaire, afin de la rendre plus efficace pour la génération aléatoire. L'utilisation de techniques

de génération aléatoires plus avancées, comme les échantillons de Boltzmann, pourrait également améliorer la complexité théorique et/ou la complexité pratique.

9.2/ Critères de couverture d'un PDA

Pour un PDA donné, un critère de couverture est un ensemble C sur ce PDA, ou sur son automate sous-jacent. En général, C est soit l'ensemble des états, soit l'ensemble des transitions, soit l'ensemble des chemins avec une restriction sur les boucles.

9.2.1/ Le critère *Tous les états*

Le critère *Tous les états* – abrégé en *AS*, pour l'anglais All States – est défini par l'ensemble des états du PDA. Dans ce contexte, $q_{AS,1}$ est la probabilité minimale de visiter un état en générant une dérivation de taille n. Si pour tout état q_0 du PDA, on note $pr(q_0)$ la probabilité qu'une trace d'exécution de taille n correponde à un chemin qui visite q_0, alors on a $q_{AS,1} = \min_q\{pr(q)\}$. Comme il existe un nombre fini d'états, on peut déduire $q_{AS,1}$ en calculant toutes les probabilités $pr(q)$. Puisque $pr(q_0)$ est le nombre de NPDA-traces de taille n qui visitent q_0 divisé par le nombre de NPDA-traces de taille n [1], il suffit de calculer ces deux nombres. Le deuxième nombre correspond à la valeur $s(n)$, définie dans la section 2.2.3 (problème 2.14). Il reste à calculer le nombre de NPDA-traces de taille n qui visitent q_0, ce qui est fait en utilisant la construction décrite ci-dessous (qui est une adaptation du produit d'automates classique) :

Soit $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ un NPDA, et $q_0 \in Q$. On défini l'automate $\mathcal{A}^{q_0}$ par $(Q \times \{0, 1\}, \Sigma, \Gamma, \delta', (q_{\text{init}}, 0), F')$, avec $F' = F \times \{1\}$ si $q_0 \notin F$, $F' = F \times \{1\} \cup \{(q_0, 0)\}$ si $q_0 \in F$, et $\delta'(q, a, X)$ défini de la façon suivante :

- Si $q \neq q_0$ et si $\delta(q, a, X) = (p, w)$, alors $\delta'((q, 0), a, X) = ((p, 0), w)$ et $\delta'((q, 1), a, X) = ((p, 1), w)$.
- Si $q = q_0$ et si $\delta(q, a, X) = (p, w)$, alors $\delta'((q_0, 0), a, X) = ((p, 1), w)$ et $\delta'((q_0, 1), a, X) = ((p, 1), w)$.

Intuitivement, le booléen ajouté aux états permet de savoir si l'état q_0 a déjà été visité. Par exemple, si on considère l'automate $\mathcal{A}_{exe}$ dont l'automate sous-jacent est décrit dans la figure 2.14 : L'automate $\mathcal{A}^6_{exe}$ a un automate sous-jacent décrit dans la figure 9.3. Notons que cet automate peut être élagué et simplifié en supprimant les états qui sont inutiles, c'est-à-dire les états $(9, 0)$, $(10, 0)$, $(8, 0)$, $(7, 0)$, $(0, 1)$, $(1, 1)$, $(2, 1)$ et $(5, 1)$.

Soit θ la fonction de l'ensemble des configurations de $\mathcal{A}^{q_0}$ vers l'ensemble des configurations de $\mathcal{A}$. Le résultat suivant est une conséquence directe de la définition de $\mathcal{A}^{q_0}$:

Proposition 9.2 *Pour toute NPDA-trace $C_1a_1C_2a_2\ldots C_na_nC_{n+1}$ de $\mathcal{A}^{q_0}$, la séquence $\theta(C_1)a_1\theta(C_2)a_2\ldots\theta(C_n)a_n\theta(C_{n+1})$ est une NPDA-trace de $\mathcal{A}$ qui visite q_0.*

Preuve. On a $C_1 = ((q_{\text{init}}, 0), \bot)$, donc $\theta(C_1) = ((q_{\text{init}}, \bot)$ est la configuration initiale de $\mathcal{A}$. De même, C_{n+1} est de la forme $((q_f, 1), \bot)$ avec $q_f \in F$, donc $\theta(C_{n+1}) = ((q_f, \bot)$ est dans un état final avec la pile vide.

Par construction de δ' et comme C_i et C_{i+1} sont a_i-consécutives, $\theta(C_i)$ et $\theta(C_{i+1})$ sont également a_i-consécutives. Par conséquent, $\theta(C_1)a_1\theta(C_2)a_2\ldots\theta(C_n)a_n\theta(C_{n+1})$ est une NPDA-trace de $\mathcal{A}$.

Il reste encore à prouver que $\theta(C_1)a_1\theta(C_2)a_2\ldots\theta(C_n)a_n\theta(C_{n+1})$ visite q_0 : Si C_{n+1} est de la forme $((q_f, 1), \bot)$, alors, comme $C_1 = ((q_{\text{init}}, 0), \bot)$, il existe i tel que C_i est de la forme $((q_i, 0), \bot u_i)$ et

1. En supposant qu'il existe au moins une NPDA-trace de taille n ; dans le cas contraire on ne pourra générer aucun test.

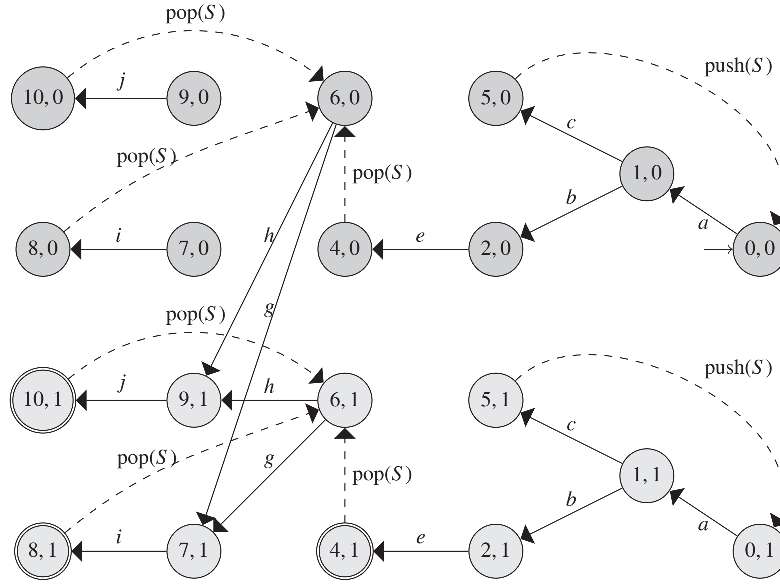


FIGURE 9.3 – Automate sous-jacent de $\mathcal{A}^6_{exe}$

C_{i+1} est de la forme $((q_i, 1), \perp u_{i+1})$. Par construction de δ', $q_i = q_0$. Si $q_0 \in F$ et si C_{n+1} est de la forme $((q_0, 0), \perp)$, alors $\theta(C_{n+1}) = (q_0, \perp)$, ce qui conclut la preuve.

□

A l'inverse, à chaque NPDA-trace de $\mathcal{A}$ qui visite q_0, correspond une unique exécution dans $\mathcal{A}^{q_0}$.

Proposition 9.3 *Pour toute NPDA-trace $C'_1 a_1 C'_2 a_2 \dots C'_n a_n C'_{n+1}$ de $\mathcal{A}$ qui visite q_0, il existe une unique NPDA-trace $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ de $\mathcal{A}^{q_0}$, telle que pour tout i, $\theta(C_i) = C'_i$.*

PREUVE. Soit $C'_i = (q_i, \perp u_i)$ pour tout i. Si l'on suppose que $q_{n+1} \neq q_0$, alors, comme $C'_1 a_1 C'_2 a_2 \dots C'_n a_n C'_{n+1}$ visite q_0, il existe $i_0 = \min\{i \mid q_i = q_0\}$. On fixe $C_i = ((q_i, 0), \perp u_i)$ si $i \leq i_0$, et $C_i = ((q_i, 1), \perp u_i)$ sinon. Par construction de δ', on peut facilement vérifier que $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ est une NPDA-trace de $\mathcal{A}^{q_0}$ et que pour tout i, $\theta(C_i) = C'_i$. Maintenant si $q_0 \in F$ et $C_{n+1} = (q_0, 0)$, alors pour tout i on fixe $C_i = ((q_i, 0), \perp u_i)$. Dans ce cas aussi, $C_1 a_1 C_2 a_2 \dots C_n a_n C_{n+1}$ est une NPDA-trace de $\mathcal{A}^{q_0}$.

Supposons maintenant que $D_1 a_1 D_2 a_2 \dots D_n a_n D_{n+1}$ est une NPDA-trace de $\mathcal{A}^{q_0}$ telle que pour tout i, $\theta(D_i) = C'_i$. Par définition de θ, D_i est de la forme $((q_i, b_i), \perp u_i)$. Tout d'abord, si $C_{n+1} = (q_0, 0)$, alors, par définition de δ', pour $i \leq i_O$, $b_i = 1$, et pour $i \geq i_0$, $b_i = 0$. Il s'ensuit que pour tout i, $D_i = C_i$. De la même façon, si $q_0 \in F$ et $C_{n+1} = (q_0, 0)$, alors, par définition de θ, D_i est de la forme $((q_i, 0), \perp u_i)$, ce qui démontre l'unicité et conclut la preuve. □

Il s'ensuit que θ induit une bijection entre les NPDA-traces de taille n de $\mathcal{A}^{q_0}$ et les NPDA-traces de taille n de $\mathcal{A}$ qui visitent q_0. Grâce à l'approche décrite dans la section 9.1.1 sur $\mathcal{A}^{q_0}$, on peut calculer en même temps le nombre de NPDA-traces dans $\mathcal{A}$ qui visitent q_0, et $pr(q_0)$.

9.2.2/ LE CRITÈRE *Toutes les transitions*

Le critère *Toutes les transitions* – abrégé en *AT*, pour l'anglais All Transitions – est défini par l'ensemble des transitions de l'automate sous-jacent. $q_{AT,1}$ est la probabilité minimale de cou-

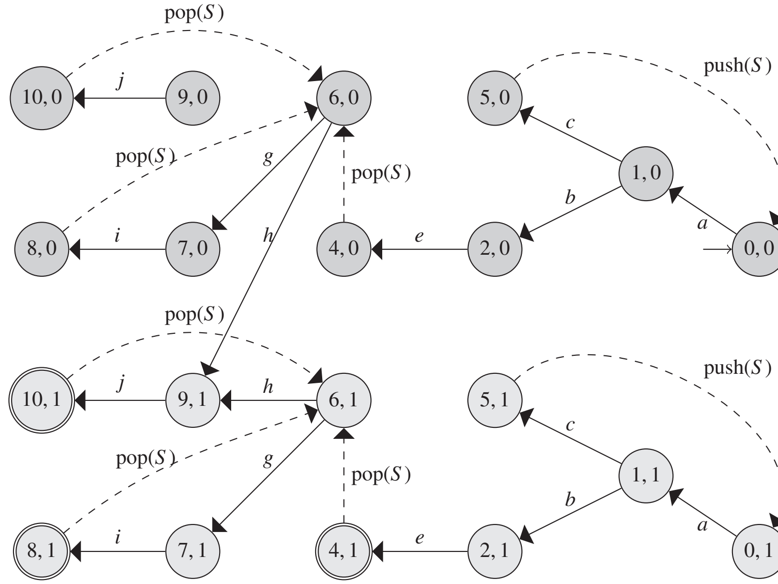


FIGURE 9.4 – Automate sous-jacent de $\mathcal{A}_{exe}^{(6,h,9)}$

vrir une transition en générant une dérivation de taille n. Pour toute transition (q_0, a_0, p_0) de l'automate sous-jacent, notons $pr((q_0, a_0, p_0))$ la probabilité qu'une trace d'exécution de taille n corresponde à un chemin qui visite (q_0, a_0, p_0). Comme pour le critère AS, la probabilité $q_{AT,1} = \min_{(q,a,q')}\{pr((q, a, q'))\}$ peut être déduite en calculant toutes les probabilités $pr((q, a, q'))$. De plus, $pr((q_0, a_0, p_0))$ est le nombre de NPDA-traces de taille n qui visitent (q_0, a_0, p_0) divisé par le nombre de NPDA-traces de taille n (= $s(n)$, qui est censé être strictement positif).

Soit $\mathcal{A} = (Q, \Sigma, \Gamma, \delta, q_{\text{init}}, F)$ un NPDA, et (q_0, a_0, q_1) une transition de l'automate sous-jacent. On défini l'automate $\mathcal{A}^{(q_0,a_0,q_1)}$ par $(Q \times \{0, 1\}, \Sigma, \Gamma, \delta^{\square}, (q_{\text{init}}, 0), F \times \{1\})$, avec $\delta^{\square}(q, a, X)$ défini de la façon suivante :

- Si $a_0 = \text{push(Y)}$ et si $\delta(q, a, X) = (p, w)$, alors
  - si $q \neq q_0$ ou si $\delta(q, \varepsilon, X) \neq (p, XY)$, alors $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), w)$ et $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 0), w)$;
  - si $q = q_0$ et si $\delta(q, \varepsilon, X) = (p, XY)$, alors $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), XY)$ et $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 1), XY)$;
- Si $a_0 = \text{pop(Y)}$ et si $\delta(q, a, X) = (p, w)$, alors
  - si $q \neq q_0$ ou si $\delta(q, \varepsilon, X) \neq (p, XY)$, alors $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), w)$ et $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 0), w)$;
  - if $q = q_0$ et si $\delta(q, \varepsilon, X) = (p, \varepsilon)$, alors $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), \varepsilon)$ et $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 1), \varepsilon)$;
- Si $a_0 \in \Sigma$ et si $\delta(q, a, X) = (p, w)$, alors
  - si $q \neq q_0$ ou si $a_0 \neq a$ or if $\delta(q, a, X) \neq (p, \varepsilon)$, alors $\delta^{\square}((q, 1), \varepsilon, X) = ((p, 1), w)$ et $\delta^{\square}((q, 0), \varepsilon, X) = ((p, 0), w)$;
  - si $q = q_0$ et si $\delta(q, a_0, X) = (p, \varepsilon)$, alors $\delta^{\square}((q, 1), a_0, X) = ((p, 1), \varepsilon)$ et $\delta^{\square}((q, 0), a_0, X) = ((p, 1), \varepsilon)$;

Intuitivement, le booléen ajouté aux états permet de savoir si la transition (q_0, a_0, p_0) a déjà été visitée (il passe de 0 à 1 lorsque la transition (q_0, a_0, p_0) est générée). Par exemple, si on considère à nouveau l'automate $\mathcal{A}_{exe}$: l'automate sous-jacent de $\mathcal{A}_{exe}^{(6,h,9)}$ est décrit dans la figure 9.4. Notons que cet automate peut être élagué et simplifié en supprimant les états inutiles.

D'une manière très proche de celle utilisée pour le critère AS, on peut démontrer qu'il existe une bijection entre les NPDA-traces de taille n dans $\mathcal{A}$ qui visitent (p_0, a_0, q_0) et les NPDA-traces de taille n dans $\mathcal{A}^{(p_0,a_0,q_0)}$, ce qui permet de calculer $pr((p_0, a_0, q_0))$.

9.2.3/ COMBINER TEST ALÉATOIRE ET CRITÈRES DE COUVERTURE

Cette section traite de la combinaison du test aléatoire et de critères de couverture (problème 7.2), dans le contexte des automates à pile. La discussion porte sur le critère AS et exploite les résultats de la section 9.2.1. Cependant, le critère AT peut être traité exactement de la même manière en utilisant les résultats de la section 9.2.2.

Adaptée au critère AS sur les automates à pile, l'approche décrite de façon générale dans la figure 7.6 devient :

1. On calcule tous les $pr(q)$.
2. Si certains d'entre eux sont égaux à 0, alors il n'est pas possible de couvrir tous les états en générant des exécutions de longueur n.
3. Sinon, on choisit aléatoirement un état q qui n'a pas été couvert par les tests précédemment générés (s'il en existe un).
4. On utilise $\mathcal{A}^q$ pour générer un test qui visite q.
5. S'il existe un état qui n'a pas été couvert par les tests précédemment générés, on retourne à l'étape 3.

Et la méthode optimisée (*cf.* figures 7.7 et 7.8) revient à résoudre le problème de programmation linéaire suivant :

maximiser $p_{\min}$ qui satisfait

$$\begin{cases} p_{\min} \leq \sum_{p \in Q} \pi_p \frac{pr(p,q)}{pr(p)} \text{ pour tout } q \in Q \\ \sum_{q \in Q} \pi_q = 1 \end{cases}$$

où $pr(p, q)$ est la probabilité qu'une NPDA-trace de longueur n visite à la fois p et q.

FIGURE 9.5 – Calcul des probabilités π_q, afin d'optimiser la probabilité de couvrir Q

La construction de $\mathcal{A}^q$ peut être adaptée à cet effet : le nouvel ensemble d'états devient $Q \times \{0, 1\} \times \{0, 1\}$ (le premier booléen représente si q a été visité, et le second représente si p a été visité). Les états finaux sont $F \times \{1\} \times \{1\}$, ce qui garantit que p et q ont tous deux été visités. De plus, notons que le problème linéaire ci-dessus doit être résolu avec des nombres réels, ce qui peut être fait efficacement. Par conséquent, il est possible de calculer les π_q afin d'obtenir une génération biaisée qui optimise la qualité probabiliste par rapport au critère AS. La même approche peut être développée pour le critère AT.

9.3/ EXPÉRIMENTATIONS

Le but principal de cette section est de fournir des informations expérimentales quantitatives sur l'approche proposée dans la section 9.2. Dans la section 9.3.2, cette approche est illustrée sur un exemple de deux fonctions mutuellement récursives. Puis, un autre exemple illustratif est présenté dans la section 9.3.3 où une requête XPath est traduite en un PDA. Enfin, la section 9.3.4 rend compte des exigences en matière de temps et d'espace pour les exemples du présent document et issus de la thèse de S.Schwoon [Sch02]. Notons qu'une étude qualitative supplémentaire de l'algorithme de test aléatoire uniforme est présentée dans la section 9.4.

```
int S(int x, int y, int n){
    int z;
    if (y == 1){
        z = M(x,n);
        return z;
    } else {
        z = x + S(x,y-1,n);
        z = M(z,n);
        return z;
    }
}
```

```
int M(int x, int n){
    if (x < n){
        return x;
    } else {
        return M(x-n,n);
    }
}
```

FIGURE 9.6 – Fonctions mutuellement récursives

9.3.1/ INFORMATION TECHNIQUE

Les expérimentations décrites dans cette section ont été obtenues sur un ordinateur personnel Intel Core2 Duo 2GHz avec 4Go de RAM, fonctionnant sur Ubuntu 10.04. Afin de valider l'approche et d'obtenir un ordre de grandeur du temps de fonctionnement et de la taille des grammaires concernées, les algorithmes ont été implémentés dans un prototype codé en Python 2.6. Python est un langage de programmation script/prototypage dont le principal avantage est de permettre de développer rapidement. Cependant, il est généralement admis que des implémentations C/C++ sont (parfois 10 à 100 fois) plus rapides que les mêmes en Python, en particulier pour les programmes qui gèrent de grandes structures de données. Les estimations de temps présentées dans cette section doivent donc être considérées dans ce contexte.

Python est un langage interprété qui utilise plusieurs mécanismes internes qui consomment de l'espace. Le goulot d'étranglement de l'espace nécessaire pour l'approche expérimentée est la taille des grammaires concernées. Par conséquent, afin d'estimer la consommation de l'espace, il semble plus pertinent de fournir les tailles des grammaires concernées plutôt que la mémoire utilisée. Notons également que les expériences soulignent que la ressource critique pour cette approche est le temps (et non l'espace).

9.3.2/ EXEMPLE ILLUSTRATIF

Cette section illustre l'approche proposée de test aléatoire sur le programme de la figure 9.6, qui utilise deux fonctions mutuellement récursives `S` et `M`.

La fonction `M(x,n)` calcule $x\%n$, et la fonction `S(x,y,n)` calcule $(x * y)\%n$. Notre objectif est de tester la fonction `S` à l'aide de l'approche développée dans cette partie. Le NPDA $\mathcal{A}_{\text{modulo}}$ associé à cette paire de fonctions est décrit dans la figure 9.7. La partie du haut du NPDA décrit la fonction `S`, et la partie du bas décrit la fonction `M`. Les appels récursifs à `M` dans `S` sont encodés par les symboles de pile M_1 (pour le cas `y==1`) et M_2 (pour le cas `y!=1`). Le symbole de pile M_3 encode l'appel à `M` dans `M`.

NPDA vs. NFA. À partir du NPDA $\mathcal{A}_{\text{modulo}}$, une grammaire associée qui satisfait les propriétés du théorème 2.15 peut être calculée. La grammaire calculée a 45 symboles de pile et 50 règles. Pour comparer notre approche avec les résultats de A.Denis et al. [DGG+12], l'approche de génération aléatoire associée a été implémentée. Le tableau 9.1 présente les résultats obtenus. La première colonne contient la longueur des traces/chemins générés dans l'automate sous-jacent dans la figure 9.7. Pour chaque longueur, 10 traces sont générées. La seconde colonne indique les NPDA-traces générées : seule la séquence d'états est donnée, et le nombre entre parenthèses

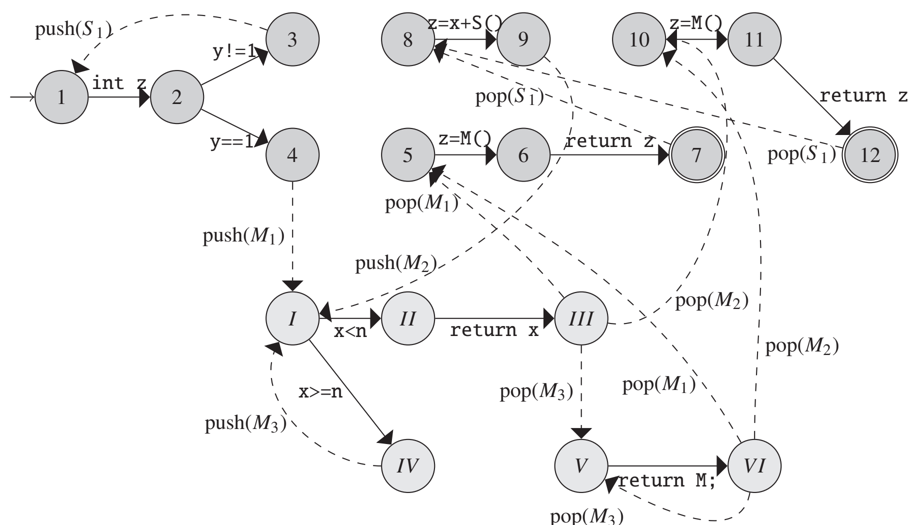


FIGURE 9.7 – $\mathcal{A}_{\text{modulo}}$: NPDA de l'exemple

Taille des traces	Notre approche	Approche de [DGG+12]
de 1 à 7	pas de NPDA-traces de cette longueur	pas de DFA-traces de cette longueur
8	1-2-4-I-II-III-5-6-7 (10)	1-2-4-I-II-III-10-11-12 (6) 1-2-4-I-II-III-5-6-7 (4)
9	pas de NPDA-traces de cette longueur	pas de DFA-traces de cette longueur
10	pas de NPDA-traces de cette longueur	1-2-4-I-IV-I-II-III-5-6-7 (3) 1-2-4-I-IV-I-II-III-10-11-12 (2) 1-2-4-I-II-III-V-VI-10-11-12 (3) 1-2-4-I-II-III-V-VI-5-6-7 (2)
11	pas de NPDA-traces de cette longueur	A-2-3-1-2-4-I-II-III-10-11-12 (6) 1-2-31-2-4-I-II-III-5-6-7 (4)

TABLEAU 9.1 – Résultats expérimentaux qualitatifs

indique le nombre de fois que cette trace a été générée aléatoirement. Le contenu de la dernière colonne est similaire, mais pour les DFA-traces.

On peut tout d'abord observer qu'il existe une unique NPDA-trace de taille 8 : 1-2-4-I-II-III-5-6-7, qui est donc évidemment générée à chaque fois. Notons que cette NPDA-trace correspond à l'exécution de `S(3,1,2)`. En revanche, il y a deux DFA-traces de longueur 8 : la première correspond à la NPDA-trace, tandis que la seconde trace 1-2-4-I-II-III-10-11-12 n'est pas compatible avec les appels de pile. Par conséquent, cette DFA-trace non cohérente n'est pas concrétisable (c'est-à-dire qu'il n'y a pas de valeur d'entrée pour S dont l'exécution correspond à cette trace). La même situation se produit pour la génération de traces de longueur 10 : il n'y a aucune DFA-trace cohérente alors qu'il y a quatre DFA-traces.

Ces résultats expérimentaux montrent que pour des traces de longueur 8, 40% des DFA-traces générées par l'approche développée dans le papier de A.Denise et al. [DGG+12] serait incohérentes, ainsi que 100% des DFA-traces de longueur 11. Le tableau 9.2 rend compte du nombre de NPDA-traces – ou de façon équivalente, de DFA-traces cohérentes – et du nombre de DFA-traces pour $\mathcal{A}_{\text{modulo}}$. Il montre que la probabilité d'obtenir une DFA-trace cohérente en générant

n	12	13	14	15	16	17	18	19	20	21	22	23	24	25	26
NPDA	1	0	0	0	1	0	0	1	1	0	0	1	1	0	0
DFA	6	4	10	6	18	10	34	18	64	34	114	64	200	114	356

n	27	28	29	30	31	32	33	34	35	36	37	80
NPDA	3	1	0	1	4	1	0	3	5	1	0	142
DFA	200	640	356	1152	640	2066	1152	3692	2066	6598	3692	$2.4\ 10^9$

TABLEAU 9.2 – NPDA-traces vs. DFA-traces

une DFA-trace devient négligeable lorsque la longueur de la trace augmente. Par exemple, la probabilité pour une DFA-trace de longueur 80 d'être cohérente est $5.9\ 10^{-8}$. De plus, pour plusieurs longueurs, comme 21, 29 ou 37, il n'y a aucune NPDA-traces de cette longueur donc toutes les DFA-traces générées seraient incohérentes.

En ce qui concerne la génération de cas de test de longueur inférieure ou égale à 60, le pré-calcul peut être effectué en moins de 1 seconde. Ensuite, 100 traces peuvent être générées en moins de 0,34 secondes. Plus d'informations sur le temps de calcul pour différentes longueurs sont disponibles dans la section 9.3.4.

Critères de couverture. L'approche développée dans la section 9.2.1 a été appliquée à l'exemple de $\mathcal{A}_{\text{modulo}}$. Pour traiter le critère *All states*, pour tout état q de $\mathcal{A}_{\text{modulo}}$, on a calculé la grammaire qui génère des NPDA-traces de $\mathcal{A}^q_{\text{modulo}}$ – qui est la grammaire qui génère des traces qui visitent q – . Ensuite, une étape de nettoyage a été effectuée. Elle consiste à retirer les symboles non-terminaux inutiles – un symbole non-terminal est inutile lorsqu'il n'existe aucun arbre d'exécution de la grammaire qui l'utilise – . Pour tout état q, le tableau 9.3 donne la taille de la grammaire correspondante : le nombre de symboles non-terminaux et le nombre de règles. Notons que le coté gauche de chaque règle contient au plus 3 symboles. Pour tout état q, le tableau 9.3 fournit également le temps (en secondes) nécessaire pour générer la grammaire et la nettoyer, ainsi que le temps (en secondes) pour effectuer l'étape de pré-calcul pour la génération aléatoire. Cette étape permet à la fois de calculer les probabilités $pr(q)$, et la génération aléatoire de NPDA-traces de $\mathcal{A}_{\text{modulo}}$ qui visitent q. Les deux temps donnés sont donnés respectivement pour la génération de la grammaire (nettoyée) et pour l'étape de pré-calcul pour la génération aléatoire (pour des traces de longueur jusqu'à 60). Par exemple, pour l'état 8, la grammaire nettoyée qui génère des traces de $\mathcal{A}^8_{\text{modulo}}$ a 81 symboles non–terminaux et 90 règles. Elle a été calculée en 250 secondes. L'étape de pré-calcul pour la génération aléatoire a été effectuée en 1.28 secondes.

Le calcul des $pr(p,q)$ demande plus de ressources. Avant de procéder à l'étape de nettoyage, les grammaires liées sont générées en 1 seconde environ, mais chacune d'entre elles a environ 9000 symboles non-terminaux et 70 000 règles. Par exemple, la plus grande avec 10581 symboles non-terminaux et 91572 règles a été générée pour les états $\{7,24\}$. Après l'étape de nettoyage, le nombre de symboles non-terminaux et de règles descend généralement à environ 100. Pour les états $\{7,24\}$ mentionnés ci-dessus, la grammaire nettoyée correspondante a 110 symboles non-terminaux et 124 règles. Pour chaque grammaire, l'étape de nettoyage a été réalisé en environ 420 secondes. À la fin, le calcul de toutes les probabilités du système de programmation linéaire présenté dans la section 9.2.3 a été réalisé en environ 34 heures. Ensuite, la résolution de ce système nécessite moins d'une seconde grâce aux récents outils basés sur les Simplex, comme l'outil `lp_solve` [MBN04]. Enfin, comme expliqué dans la section 9.3.4, la génération aléatoire de centaines de traces se fait en quelques secondes.

état	1	2	3	4	5	6	7	8	9	10	11	12
taille de la grammaire	80+ 88	80+ 89	80+ 87	81+ 90	82+ 91	83+ 92	84+ 93	81+ 90	82+ 91	83+ 92	84+ 93	85+ 92
temps	248 1.19	250 1.18	249 1.19	250 1.23	249 1.23	249 1.23	248 1.24	250 1.28	251 1.35	250 +1.35	249 1.32	251 1.72

état	I	II	III	IV	V	VI
taille de la grammaire	83+ 92	83+ 92	83+ 92	92+ 101	92+ 104	92+ 104
temps	249 1.29	250 1.29	250 1.29	250 1.31	250 1.29	251 1.32

TABLEAU 9.3 – Calcul des probabilités $pr(q)$ pour $n \leq 60$ (en secondes).

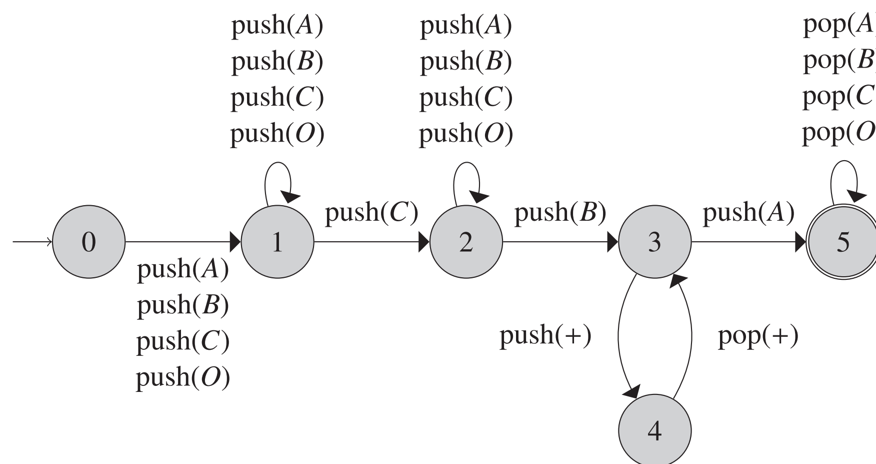


FIGURE 9.8 – Automate $\mathcal{A}_{\text{xpath}}$

En ce qui concerne le critère *AT*, le temps pour calculer les probabilités est très proche. En revanche, puisqu'il y a plus de transitions, le temps total pour calculer le système linéaire est estimé à 216 heures. La section 9.3.4 explique comment ce temps de calcul peut être réduit à un temps attendu de quelques heures.

9.3.3/ Exemple illustratif pour les critères de couverture

Dans cette section, l'approche développée dans la section 9.2 est expérimentée sur un exemple du NPDA $\mathcal{A}_{\text{xpath}}$ de la figure 9.8, qui correspond à une requête XPath. Rappelons qu'à des fins de traitement de flux, les requêtes XPath peuvent être automatiquement traduites dans une sous-classe des PDA, appelée *Visibly pushdown automata* [AM04]. L'exemple de la figure 9.8 est inspiré de l'exemple dans [Cla08, Fig. 5].

Suite aux propositions de la section 9.2, pour une longueur n donnée de traces, les trois algorithmes suivants ont été comparés :

- L'algorithme 1 procède à une génération de NPDA-traces de longueur n, jusqu'à ce que tous les états soient couverts.
- L'algorithme 2 :

  1. génère une NPDA-trace de longueur n aléatoire-uniformément,

n	12	14	16	18	20	22	24	26	28	30	32
Algorithme 1	7.51	6.1	5.0	4.77	4.74	4.24	4.43	4.37	4.34	4.34	4.33
Algorithme 2	1.87	1.84	1.83	1.77	1.8	1.78	1.79	1.77	1.77	1.76	1.77
Algorithme 3	1	1	1	1	1	1	1	1	1	1	1
Algorithme 4	14.1	11.03	10.08	9.27	8.83	9.06	8.64	9.36	9.45	8.59	9.43
Algorithme 5	6.9	6.07	5.57	5.08	4.92	4.68	4.66	4.71	4.51	4.52	4.51
Algorithme 6	14.1	-	-	9.1	-	-	-	-	8.8	-	-

TABLEAU 9.4 – Nombre moyen de traces générées pour couvrir tous les états/toutes les transitions

2. choisit aléatoire-uniformément un état non visité et génère aléatoire-uniformément une trace qui le visite,
3. répète l'étape (2) jusqu'à ce que tous les états aient été visités.

— L'algorithme 3 consiste à résoudre le système de programmation linéaire de la figure 9.5, puis à générer des NPDA-traces avec les probabilités calculées, jusqu'à ce que tous les états aient été visités.

Les algorithmes 4, 5 et 6 correspondent respectivement aux algorithmes 1, 2 et 3, mais pour couvrir toutes les transitions (critère *AT*).

Les grammaires nécessaires pour générer les traces couvrant un état donné ont été calculées en 38 secondes de façon automatique. Le système linéaire décrit dans la section 9.2.3 (figure 9.5) est calculé en 250 secondes, et sa résolution indique que la solution optimale consiste à toujours générer une trace qui visite l'état 4. En effet, un rapide coup d'œil à l'automate montre que toutes les NPDA-traces couvrent tous les états, exception faite de l'état 4 ; une telle trace visite donc tous les états. Les résultats sur le nombre moyen de traces générées pour les six algorithmes mentionnés ci-dessus sont donnés dans le tableau 9.4. Ces valeurs moyennes ont été obtenues en répétant chaque expérience 100 fois. L'algorithme 6 a été expérimenté pour les tailles 12, 18 et 28. Pour l'algorithme 6 l'approche nécessite plusieurs transformations, qui sont faites pour le moment de façon manuelle mais qui peuvent être automatisées.

Pour cet exemple, l'algorithme 3 est sans surprise le meilleur puisqu'une unique NPDA-trace suffit pour visiter tous les états.

Pour l'algorithme 1, les résultats obtenus sont conformes aux attentes théoriques. En effet, pour cet algorithme, des NPDA-traces sont générées jusqu'à en avoir une qui visite l'état 4. Théoriquement, cela nécessite un nombre attendu de $1/p$ générations, où p est la probabilité qu'un chemin qui visite l'état 4 soit généré. Par exemple, la probabilité théorique qu'une NPDA-trace de longueur 14 visite l'état 4 est d'environ 0.16. Par conséquent, le nombre théoriquement attendu de NPDA-traces à générer pour en obtenir une qui visite l'état 4 est proche de $1/0.16 \simeq 6.25$. Ce qui explique le résultat expérimental de 6.1, obtenu pour l'algorithme 1. De même, il y a 6372 NPDA-traces de longueur 16 et 1252 d'entre elles visitent l'état 4. Donc la probabilité théorique qu'une trace de longueur 16 visite l'état 4 est $\frac{1252}{6372} \simeq 0.196$. Ainsi, le nombre théoriquement attendu de traces nécessaires pour visiter l'état 4, et par conséquent tous les états, est $\frac{6372}{1252} \simeq 5.1$, ce qui est proche de la valeur 5.0 obtenue expérimentalement.

Pour l'algorithme 2, une seule itération suffit si le premier chemin généré visite 4 (avec une probabilité de 0.16). Dans le cas contraire, il faut deux itérations. Par exemple, pour des NPDA-traces de longueur 14, le nombre théoriquement attendu de traces générées pour couvrir tous les états avec cet algorithme est $0.16 * 1 + (1 - 0.16) * 2 = 1.84$, ce qui est exactement la valeur obtenue expérimentalement.

Des remarques et comparaisons avec les résultats théoriques similaires peuvent être faîtes pour les algorithmes 4 et 5. De plus, l'algorithme 5 fonctionne (sans surprise) beaucoup mieux que l'algorithme 4.

Pour le critère *AT*, contrairement au critère *AS*, l'algorithme 6 ne fonctionne pas mieux que l'algorithme 5. En fait, l'optimisation induite par la probabilité minimale de visiter une transition est légère. Par exemple, pour des traces de longueur 18, la probabilité minimale de visiter une transition est de 0.2 en générant aléatoire-uniformément une NPDA-trace. Cette probabilité minimale passe à 0.25 en utilisant l'optimisation de l'algorithme 5. Pour les traces de longueur 28, la probabilité minimale passe de 0.22 à 0.25. Sur cet exemple, l'algorithme 5 n'apporte donc pas de très gros bénéfice. En comparaison, pour le critère *AS*, la probabilité minimale de visiter un état passe de 0.22 avec l'algorithme 1, à 1.0 avec l'algorithme 3.

9.3.4/ Temps d'exécution sur plus d'exemples

Cette section est consacrée à l'étude expérimentale de la durée et de l'espace requis par les algorithmes proposés dans cette partie. Les NPDA utilisés pour ces expérimentations sont les automates $\mathcal{A}_{\text{modulo}}$ (figure 9.7), $\mathcal{A}_{\text{xpath}}$ (figure 9.8), $\mathcal{A}_{\text{power}}$ (figure 2.14), $\mathcal{A}_{\text{SY}}$ qui encode l'*algorithme Shunting-Yard* décrit dans la section 9.4, et l'automate $\mathcal{A}_{\text{plotter}}$ de la thèse de S.Schwoon [Sch02], qui décrit un algorithme sur les graphes. Notons que les NPDA peuvent être calculés automatiquement à partir du code source C ou Java en utilisant l'outil JimpleToPDSolver [2] [HO10] ou l'outil PuMoc [3] [ST12]. Les NPDA utilisés dans cette section ont des tailles comparables à la taille moyenne des systèmes à pile qui correspondent à des pilotes Windows donnés dans le papier [ST11] de F.Song et T.Touili.

Test aléatoire-uniforme. Les résultats expérimentaux sur l'algorithme de test aléatoire-uniforme de la section 9.1 sont présentés dans le tableau 9.5. Dans ce tableau, la seconde colonne donne la taille (nombre d'états + nombre de transitions) du NPDA considéré. Ici le nombre entre parenthèses indique le nombre de transitions du NPDA considéré comme un PDA ; par exemple pour la transition $(1, \texttt{intz}, 2)$ de la figure 9.7, il y a dans le PDA autant de transitions que de symboles de la pile X, de la forme $\delta(1, \texttt{intz}, X) = (2, X)$. La troisième colonne du tableau indique la taille de la grammaire générée (nombre de symboles + nombre de règles) avant de la nettoyer, et la quatrième colonne donne le temps de calcul pour cette génération. Les deux colonnes suivantes donnent respectivement la taille de la grammaire après le nettoyage, et le temps nécessaire pour ce nettoyage. Rappelons que l'étape de nettoyage consiste à enlever les symboles non-terminaux inutiles et les règles inutiles. L'avant-dernière colonne donne les détails sur le temps de pré-calcul pour la génération aléatoire correspondant au calcul des $s(i)$ tel que défini dans la section 9.1.1 : le premier nombre entre parenthèses est la taille maximale utilisée pour arrêter l'étape de pré-calcul, et le second nombre est le temps de calcul. Par exemple, pour l'automate $\mathcal{A}_{\text{xpath}}$, (30) 0.57 signifie que le temps de pré-calcul pour générer des traces de taille au plus 30 est de 0.57 secondes. La dernière colonne donne le temps pour générer 100 traces de la taille indiquée par le nombre entre parenthèses. Par exemple, pour la première ligne de $\mathcal{A}_{\text{modulo}}$, (8) 0.07 indique que 0.07 secondes sont nécessaires pour générer 100 traces de longueur 8. Notons que le temps est donné pour la taille 8 car il n'y a pas de NPDA-trace de taille 10. On peut noter que le nettoyage de la grammaire est une étape coûteuse. L'étape de pré-calcul devient significative pour de très longues traces. Cependant, une fois le pré-calcul effectué, la génération aléatoire de NPDA-traces ne coûte quasiment

2. Un outil pour évaluer des formules en mu-calcul sur des systèmes à pile, et des jeux de parité à pile : http://www.cs.ox.ac.uk/matthew.hague/pdsolver.html

3. Un model-checker CTL pour les systèmes à pile et les programmes séquentiels : http://www.liafa.univ-paris-diderot.fr/~song/PuMoC/

	Taille du NPDA	Taille de la grammaire	Temps de génération de la grammaire	Taille de la grammaire nettoyée	Temps de génération de la grammaire nettoyée	Temps de pré-calcul	Temps de génération 100 traces
$\mathcal{A}_{\text{xpath}}$	6+21 (102)	217+ 3463	0.03	27+87	0.38	(10) 0.07 (20) 0.22 (30) 0.57 (100) 17.21 (200) 141	(10) 0.09 (20) 0.17 (30) 0.63 (100) 1.1 (200) 2.83
$\mathcal{A}_{\text{power}}$	10+12 (24)	201+369	0.02	27+31	0.05	(100) 0.98 (200) 5.63 (500) 78.96	(100) 0.41 (196) 0.94 (496) 3.94
$\mathcal{A}_{\text{modulo}}$	18+24 (90)	1621+ 7572	0.07	45+50	6.07	(10) 0.03 (20) 0.08 (50) 0.41 (100) 2.29 (200) 14.86	(8) 0.07 (21) 0.14 (49) 0.32 (99) 0.28 (199) 1.4
$\mathcal{A}_{\text{SY}}$	23+ 36 (102)	1937+ 16735	0.13	143+ 261	16.61	(10) 0.16 (20) 0.58 (30) 1.46 (100) 39.21 (200) 326.28	(10) 0.13 (20) 0.24 (30) 0.34 (100) 1.14 (200) 2.35
$\mathcal{A}_{\text{plotter}}$	1+22	21+24	0.01	21+24	0.01	(20) 0.05 (30) 0.11 (100) 2.21 (200)15.23 (300) 52.37 (500) 255.18	(17) 0.07 (30) 0.11 (100) 0.44 (200) 1.01 (300) 1.7 (500) 3.41

TABLEAU 9.5 – Expérimentations de l'algorithme de test aléatoire-uniforme (temps en secondes)

rien.

Test aléatoire biaisé. Pour générer des NPDA-traces d'une longueur donnée, la première étape consiste à calculer pour chaque état q, une grammaire qui génère des DFA-traces cohérentes (ce qui revient à générer des NPDA-traces) qui visitent q. Cette étape permet à la fois de calculer la probabilité $pr(q)$ qu'une NPDA-trace visite q, et de générer uniformément les traces en question. Le tableau 9.6 donne pour chaque exemple, le temps nécessaire pour calculer toutes les probabilités $pr(q)$. Il fournit également les tailles moyennes (nombre de symboles + nombre de règles) des grammaires liées. Notons que pour l'algorithme Shunting-Yard, les grammaires générées – calculées en 25 secondes environ – ont environ 17000 symboles non terminaux, et 400000 règles. Le temps pour $\mathcal{A}_{\text{plotter}}$ n'est pas indiqué car ce NPDA a un seul état. Ces résultats montrent que l'approche proposée au début de la section 9.2.3 peut facilement être appliquée à nos exemples. Les deux première lignes du tableau 9.6 correspondent au calcul des probabilités de couvrir les états, et les deux suivantes correspondent au calcul des probabilités de couvrir les transitions.

La première ligne du tableau 9.7 donne le temps moyen expérimental pour calculer une probabilité $pr(p,q)$ (pour les états). Pour une paire de transitions, il n'y a pas de ligne spéciale dans le tableau 9.7 car le temps moyen calculé est vraiment proche de celui des états, et les automates liés

	$\mathcal{A}_{\text{xpath}}$	$\mathcal{A}_{\text{power}}$	$\mathcal{A}_{\text{modulo}}$	$\mathcal{A}_{\text{SY}}$	$\mathcal{A}_{\text{plotter}}$
Temps moyen (état)	38s	5.25s	22min	136min	-
Taille grammaire moyenne	31.3+98.8	32.2+36.8	65.2+72.9	342.7+692.9	26+31
Temps moyen (transition)	91s	6.3s	33min	219min	15.3s
Taille grammaire moyenne	30.5+94.8	30.7+34.3	66.2+73.7	98.3+205.6	23.8+27.3

TABLEAU 9.6 – Temps de calcul des probabilités $pr(q)$ (pour des traces de taille 60)

	$\mathcal{A}_{\text{xpath}}$	$\mathcal{A}_{\text{power}}$	$\mathcal{A}_{\text{modulo}}$	$\mathcal{A}_{\text{SY}}$	$\mathcal{A}_{\text{plotter}}$
Temps moyen (une paire d'états)	13.4s	0.9s	408s	110min	1.16s
Total (état)	7min19s	79s	34h12min	847h (*)	-
Total (transition)	25min	81s	60h39min	1200h (*)	9min
Total (simplifié, état)	1min	-	9h (*)	120h (*)	-
Total (simplifié, transition)	8min (*)	-	18h (*)	350h (*)	-

TABLEAU 9.7 – Temps de calcul des probabilités $pr(p,q)$ et des systèmes linéaires (pour des traces de taille 60)

sont de taille et de structure similaire. Ce tableau indique également la durée totale pour calculer le système de programmation linéaire (*cf.* section 9.2.3). La notation (*) est utilisée pour signaler que la valeur indiquée est un temps estimé, calculé à partir d'un échantillon de probabilités calculées. Pour $\mathcal{A}_{\text{plotter}}$, les probabilités pour les états ne sont pas pertinentes car cet automate possède un seul état. Les deux dernières lignes de tableau 9.7 indiquent le temps estimé pour calculer le système linéaire (*cf.* section 9.2.3) en quelques certaines simplifications. Par exemple, pour $\mathcal{A}_{\text{power}}$ représenté dans la figure 2.14, toutes les NPDA-traces qui visitent l'état 9 visitent également l'état 10, et inversement. Par conséquent, $pr(9,q) = pr(10,q)$ pour tout état q. De plus, toutes les NPDA-traces visitent l'état 1 ; il s'ensuit que $pr(1,q) = pr(q)$ pour tout état q. Avec ce genre d'observations, le nombre de probabilités $pr(p,q)$ à calculer diminue. Des remarques similaires peuvent être faites pour les transitions. Avec ces observations, le tableau 9.7 montre que le temps de calcul (estimé) est nettement meilleur. Pour $\mathcal{A}_{\text{plotter}}$ et $\mathcal{A}_{\text{power}}$, le temps n'est pas fourni car le temps de calcul par force brute est déjà faible. Avant d'en terminer avec le tableau 9.7, notons que le calcul du système linéaire pour $\mathcal{A}_{\text{SY}}$ pour les deux critères semble être intraitable avec le prototype implémenté. De plus, les simplifications possibles pour calculer les probabilités (pour les différents exemples) n'ont pas été implémentées dans le prototype ; cependant, plusieurs dépendances simples pourraient être détectées automatiquement. Les probabilités estimées ont été obtenues en comptant le nombre de $pr(p,q)$ suffisantes pour obtenir le système linéaire à l'aide de ces simplifications ; et en multipliant ce nombre par le temps moyen pour calculer une probabilité $pr(p,q)$. Les valeurs indiquées donnent donc l'ordre de grandeur d'un temps prévu.

Analyse des résultats expérimentaux. Premièrement, on peut noter que les exemples utilisés pour les expérimentations sont de taille comparable à la taille de systèmes à pile qui modélisent des programmes récursifs, comme indiqué dans le papier [ST11] de F.Song et T.Touili sur la vérification de pilotes Windows : les PDA concernés ont en général moins de 10 états et quelques dizaines de transitions.

Deuxièmement, comme mentionné précédemment, il est communément admis que des implémentations en C/C++ sont 10 à 100 fois plus rapides que la version codée en Python, en particulier pour les programmes qui gèrent de grandes structures de données. Par conséquent, un outil codé en C/C++ peut rendre notre approche très utile pour des applications pratiques, avec des temps de

calcul traitables.

Troisièmement, l'analyse des résultats expérimentaux montre que le calcul de la grammaire nettoyée pour l'approche aléatoire biaisée est une étape coûteuse. Cependant chaque grammaire (simplifiée) peut être calculée indépendamment, donc l'approche peut être distribuée sur plusieurs ordinateurs de façon triviale (par exemple un ordinateur par probabilité calculée). Par exemple, pour $\mathcal{A}_{\text{xpath}}$ et pour le critère AT, il y a environ 200 probabilités $pr(t_1, t_2)$ à calculer (rappelons que $pr(t_1, t_2) = pr(t_2, t_1)$). L'utilisation de 10 ordinateurs en parallèle peut diviser le temps de calcul par 10 : les expérimentations montrent que chaque probabilité est calculée dans un temps relativement similaire, ce qui rend la parallélisation très efficace.

Pour conclure, les résultats expérimentaux montrent que l'approche de test aléatoire-uniforme (qui correspond aux algorithmes 1 et 4 de la section 9.3.3) est facilement applicable, et que des applications pratiques sont traitables. Le test aléatoire biaisé basé sur les états ou les transitions déjà visités (qui correspond aux algorithmes 2 et 5 de la section 9.3.3) peut également être facilement réalisé, même avec un prototype codé en Python (*cf.* tableau 9.6). L'optimisation de la qualité de l'approche de test (qui correspond aux algorithmes 3 et 6 de la section 9.3.3) nécessite davantage de ressources. Excepté pour $\mathcal{A}_{\text{SY}}$, le test avec le critère AS est traitable, mais pour le critère AT, des optimisations – comme des simplification de calcul, un outil codé en C++, ou encore du calcul distribué – doivent être étudiées pour obtenir des temps de calcul raisonnables. Notons que pour $\mathcal{A}_{\text{SY}}$, utiliser à la fois de simplifications de calcul et une implémentation en C++ devrait fournir un temps de calcul attendu raisonnable de l'ordre de quelques heures.

9.4/ Cas d'étude : l'algorithme Shunting-Yard

Dans cette section, l'application des techniques de génération aléatoire est présentée dans le cadre du MBT (Model-Based Testing). Á cet effet, une implémentation de l'algorithme Shunting-Yard disponible sur internet [4] est analysée.

9.4.1/ Description

Cette section décrit l'algorithme Shunting-Yard [Dij61] proposé par Dijkstra pour convertir les expressions mathématiques de la notation infixée (usuelle), dans la notation post-fixée. Le nom de cet algorithme est dû à son fonctionnement rappelant celui d'une gare de triage ferroviaire (shunting yard en anglais). La notation post-fixée est une notation basée sur la pile, proche de la notation par arbre syntaxique, et utilisée par exemple par les calculatrices de poche HP. Par exemple, les expressions $3 + 4$ ou encore $3 + 4 * (2 - 1)$ deviennent respectivement $3\ 4\ +$ et $3\ 4\ 2\ 1\ -\ *\ +$ en notation post-fixée.

Comme l'algorithme Shunting-Yard est basé sur la pile, il est tout à fait adapté à une modélisation sous forme d'un PDA. La conversion utilise deux variables de type texte (chaînes de caractère) pour l'entrée et la sortie, et une pile qui contient les opérateurs qui n'ont pas encore été traités. Pour convertir, l'algorithme lit un symbole sur l'entrée, puis il le traite en fonction du symbole lu ainsi que de l'opérateur qui se trouve au sommet de la pile. La figure 9.9 représente une description textuelle de l'algorithme Shunting-Yard, simplifié en supprimant les tokens de fonction ainsi que l'opérateur ˆ.

9.4.2/ Automate sous-jacent pour l'algorithme Shunting-Yard

Comme expliqué précédemment, la modélisation par un PDA correspond bien aux besoins de l'algorithme Shunting-Yard, qui exploite une pile. À partir de sa description (figure 9.9), l'algorithme

4. http://en.wikipedia.org/wiki/Shunting-yard_algorithm

Shunting-Yard
Entrée : une liste e de caractères qui est une expression arithmétique dans la forme usuelle. Chaque élément de e est soit un nombre dans $\{0, \ldots, 9\}$, soit une parenthèse (gauche/ouvrante ou droite/fermante), soit un symbole opérateur.
Sortie : une liste t qui correspond à l'expression entrée dans la notation post-fixée.

Algorithme :
t est la liste vide.
s est une pile locale vide.
Tant que e n'est pas vide **faire**
c prend la valeur du premier élément de e.
retirer le premier élément de e.
Si c est un nombre, **alors** l'ajouter à la fin de t. **FinSi**
Si c est un opérateur, **alors**
Tant que le sommet de la pile s contient un opérateur o avec une précédence supérieure ou égale
à la précédence de c **faire**
pop o du sommet de s
ajouter o à la fin de t.
FinTantQue
push c au sommet de s.
FinSi
Si c est une parenthèse gauche **alors** push c au sommet de s. **FinSi**
Si c est une parenthèse droite **alors**
Tant que le sommet de la pile s contient un élément o qui n'est pas une parenthèse gauche **faire**
pop o du sommet de s
ajouter o à la fin de t.
FinTantQue
pop le sommet de s (qui est une parenthèse gauche)
FinSi
FinTantQue
TantQue s n'est pas vide **faire** pop l'élément au sommet de s et l'ajouter à la fin de t. **FinTantQue**
Retourner t.

Figure 9.9 – Algorithme Shunting-Yard

Shunting-Yard est modélisé par un PDA dont l'automate sous-jacent est décrit dans la figure 9.10. Pour des raisons de lisibilité, le modèle ne prend en compte que les opérateurs "+" et "*". Notons toutefois que l'utilisation de ces opérateurs est suffisante pour illustrer et valider les propositions faîtes dans cette partie. Dans cet automate, les symboles de la pile sont Z (pour la pile vide), $X_+, X_($ et X_* pour coder le fait que la pile contient, respectivement, +, (et $*$. Les transitions *read* représentent une lecture de l'entrée, et les transitions *write* représentent une écriture dans la sortie. Notons que les x utilisés dans les transitions entre q_0 et q_d peuvent être remplacés par n'importe quel chiffre de $\{0, 1, \ldots, 9\}$. Les action *EOI* encodent le fait qu'il n'y a plus rien à lire sur l'entrée. Les pairs de transitions de la forme $(q_+, \text{pop}(Z), q_1)(q_1, \text{push}(Z), q_{+end})$ représentent le fait de contrôler si la pile (dans l'implémentation) est vide (sur le modèle, on regarde si la pile contient uniquement le symbole Z).

Par exemple, si on considère la chaîne d'entrée $3*4+2$. Le chemin correspondant dans l'automate

sous-jacent est :

$(q_{\text{init}}, \text{push}(Z), q_0)(q_0, \text{read } 3, q_d)(q_d, \text{write } 3, q_0)(q_0, \text{read } *, q_*)(q_*, \text{pop}(Z), q_4)(q_4, \text{push}(Z), q_{*end})$
$(q_{*end}, \text{push}(X_*), q_0)(q_0, \text{read } 4, q_d)(q_d, \text{write } 4, q_0)(q_0, \text{read } +, q_+)(q_+, \text{pop}(X_*), q_{+*})$
$(q_{+*}, \text{write } *, q_+)(q_+, \text{pop}(Z), q_1)(q_1, \text{push}(Z), q_{+end})(q_{+end}, \text{push}(X_+), q_0)(q_0, \text{read } 2, q_d)$
$(q_d, \text{write } 2, q_0)(q_0, EOI, q_8)(q_8, \text{pop}(X_+), q_6)(q_6, \text{write } +, q_8)(q_8, \text{pop}(Z), q_f)$

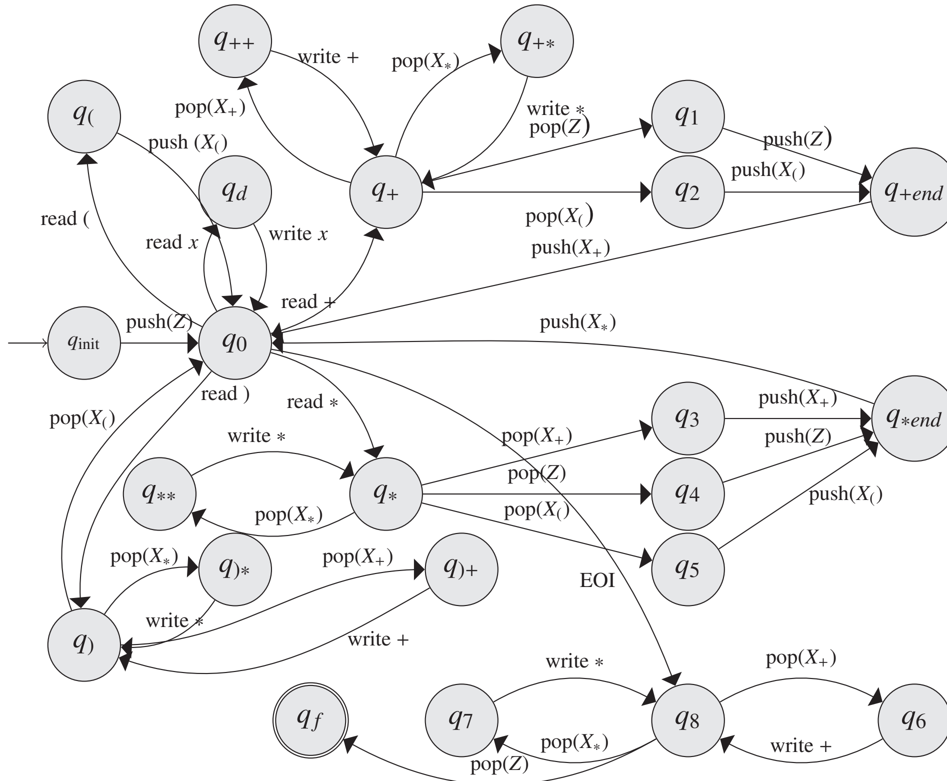


FIGURE 9.10 – Automate sous-jacent pour l'algorithme Shunting-Yard

9.4.3/ EXPÉRIMENTATIONS

Cette section rend compte des expérimentations réalisées sur une implémentation en C de l'algorithme Shunting-Yard disponible sur internet [5] et légèrement simplifié pour gérer les opérateurs "+" et "*". Ce code simplifié est donné dans les figures 9.11 et 9.12. Dans un premier temps, l'approche est évaluée en mesurant la proportion de lignes de code exécutées, et la proportion de transitions visitées. Deuxièmement, l'approche est également évaluée à l'aide d'opérateurs de mutation : 10 morceaux de code modifiés ont été générés à l'aide d'un outil disponible gratuitement, appelé Mutate [6]. Suivant la classification classique des mutants ([ADH⁺89]), quatre des dix mutants sont de type Oido type (incrémente/décrémente), un est de type Varr (remplacement de référence de tableau), un est de type SSWM (mutation d'un bloc switch), et quatre sont de type ONLG (négation logique). Notons que le code testé implémente la détection d'erreurs de syntaxe dans le fichier d'entrée, et que les parties du code qui gèrent les entrées erronées ne sont pas traitées par le modèle puisque notre PDA est conçu pour des entrées valides. Par conséquent, le test

5. http://en.wikipedia.org/wiki/Shunting-yard_algorithm
6. Son code est accessible, par exemple, à l'adresse http://members.femto-st.fr/pierre-cyrille-heam/mutatepy

```
#include <string.h>
#include <stdio.h>
#define bool int
#define false 0
#define true 1
int op_preced(const char c){
    switch(c)    { case '*': return 3; case '+': return 2;}
    return 0;}
bool op_left_assoc(const char c){
    switch(c)    {case '*':  return true; case '+': return true;  }
    return false;
}
#define is_operator(c)  (c == '+' || c == '*')
#define is_ident(c)     (c >= '0' && c <= '9')

bool shunting_yard(const char *input, char *output){
    const char *strpos = input, *strend = input + strlen(input);
    char c, *outpos = output;
    char stack[32];       // operator stack
    unsigned int sl = 0;  // stack length
    char     sc;          // used for record stack element
    while(strpos < strend)   {
        c = *strpos;
        if(c != ' ')    {
  if(is_ident(c))  {
    *outpos = c; ++outpos;
  }
  else if(is_operator(c))  {
    while(sl > 0)    {
      sc = stack[sl - 1];

    if(is_operator(sc)&&((op_left_assoc(c)&&(op_preced(c)<=op_preced(sc)))||(op_preced(c)<op_preced(sc)))){
                    // Pop op2 off the stack, onto the output queue;
                    *outpos = sc;
                    ++outpos;
                    sl--;}
                else   {
                    break;}
                }

stack[sl] = c;
                ++sl;}

    else if(c == '(')   {
                stack[sl] = c;
                ++sl; }

    else if(c == ')')    {
                bool pe = false;
while(sl > 0)     {
                    sc = stack[sl - 1];
                    if(sc == '(')    {
                        pe = true;
                        break;}
                    else  {
                        *outpos = sc;
                        ++outpos;
                        sl--;}
                }

if(!pe)  {
                    printf("Error: parentheses mismatched\n");
                    return false;}
sl--;
if(sl > 0)   {
                    sc = stack[sl - 1];}
            }
            else  {
                printf("Unknown token %c\n", c);
                return false; // Unknown token
          }
        }
        ++strpos;
    }
    while(sl > 0)  {
        sc = stack[sl - 1];
        if(sc == '(' || sc == ')')   {
            printf("Error: parentheses mismatched\n");
            return false;}
        *outpos = sc;
        ++outpos;
        --sl;}
    *outpos = 0; // Null terminator
    return true;
}
```

FIGURE 9.11 – Implémentation en C de l'algorithme Shunting-Yard, que nous avons testée 1/2

```
int main (int argc, char *argv[],char **envp) {
    const char *input = argv[1];
    char output[128];
    if (argc != 2) {
      printf("Error:%d arguments expected",argc-1);
      if (argc > 1) {
printf(" ( ");
int i;
for(i=1;i<argc-1;i++) {
  printf("%s , ",argv[i]);   }
printf("%s ) ",argv[argc-1]);
      }
     printf("Error: a unique argument is expected\n");
     return -1;   }
    if(shunting_yard(input, output))    {
        printf("input: %s\n", input);
        printf("output: %s\n", output);    }
    return 0;
}
```

FIGURE 9.12 – Implémentation en C de l'algorithme Shunting-Yard, que nous avons testée 2/2

aléatoire ne peut pas atteindre toutes les lignes du code. Les mutations présentées ci-dessus se trouvent toutes sur des parties du code qui peuvent être exécutées avec une entrée valide.

Comme dans la section 9.3, le prototype développé a été utilisé pour les expérimentations. À nouveau, on utilise ce prototype pour transformer un PDA en NPDA, qui est utilisé pour produire automatiquement une grammaire algébrique. Puis la grammaire en question est utilisée pour générer automatiquement des exécutions, grâce à l'outil GenRGenS.

Notre prototype calcule, pour chaque exécution, la séquence d'entrée qui correspond pour le programme (la lit sur l'automate) et compare la sortie de l'automate avec la sortie du programme. Si elles sont égales, le test réussi, sinon il échoue : les sorties de l'automate constituent donc l'oracle pour la campagne de test. La couverture des lignes du code source C et la couverture des transitions de l'automate sont calculées également. Enfin, le prototype rend compte également du nombre de mutants détectés.

Pour la première expérience, le but est de détecter tous les mutants, avec le protocol expérimental suivant : pour chaque mutant, un test signalant l'erreur l'erreur de code associée doit être généré. Le but de la seconde expérience est de générer des tests aléatoirement jusqu'à ce que toutes les transitions soient couvertes par la suite de tests. Pour la dernière expérience, le but est de couvrir la plupart des lignes de code. Toutefois, comme expliqué précédemment, les lignes qui gèrent des entrées incorrectes ne peuvent pas être atteintes avec cette méthode. Par conséquent l'objectif est de couvrir 100% des lignes de code qui sont accessibles à partir d'une entrée valide, ce qui représente environ 79% des lignes de code de l'application. Chaque expérience a été réalisée 20 fois. Le nombre minimum, le nombre maximum, et le nombre moyen de tests nécessaires pour atteindre l'objectif sont donnés dans le tableau 9.8. Chaque résultat a été obtenu en quelques secondes à partir d'un ordinateur portable commercial classique.

En conclusion, les expériences montrent que la technique de test proposée est très efficace avec une taille de tests raisonnable et un petit nombre de tests.

	Détection de tous les mutants			Couverture de toutes les transitions			Couverture de 79% du code		
longueur des chemins	min	max	moy.	min	max	moy.	min	max	moy.
10	2	16	7.8	7	48	15.3	3	17	6.9
15	1	7	3.7	4	19	8.8	2	7	4.5
20	2	11	3.2	3	10	5.4	2	6	2.9

TABLEAU 9.8 – Résultats expérimentaux pour l'algorithme Shunting-Yard

9.5/ DISCUSSION

Choisir les paramètres de test. Le cadre de test proposé utilise deux paramètres : le nombre et la longueur des cas de test. Pour le nombre de cas de test, plusieurs approches peuvent être adoptées. Par exemple, il est possible de générer des cas de test jusqu'à ce qu'un critère de couverture donné soit complètement atteint (comme on le fait avec les algorithmes 1 et 4 de la section 9.3.3. Dans ce cas, le nombre de cas de test est $\frac{1}{p_{\min}}$, où $p_{\min}$ est la probabilité minimale de couvrir un élément du critère de couverture que l'on souhaite atteindre. D'après le papier [DGG+12] de A.Denise et al., le nombre de cas de test peut également être fixé *a priori* – en utilisant la formule $N \geq \frac{\log(1-p_{\text{quality}})}{\log(1-p_{\min})}$ – afin d'obtenir une probabilité p_{quality} de couvrir tous les éléments du critère de couverture visé. La même formule peut être adoptée pour les approches fondées sur la résolution du système linéaire. Pour une approche aléatoire biaisée (comme les algorithmes 4 et 6 de la section 9.3.3), le nombre moyen de tests pour assurer une qualité définie est difficile à calculer car il y a de nombreuses dépendances. Il est toutefois borné par le nombre de tests requis par l'approche aléatoire non biaisée. Bien sûr, le nombre de cas de test générés peut également dépendre du contexte : pour une campagne de test nécessitant des manipulations manuelles, il doit être réduit. Inversement, pour du test massif, comme le Fuzz testing, il doit être très grand.

Le choix de la longueur n des traces générées (cas de test) dépend du système à tester ainsi que du contexte. Pour du test de performance, on peut choisir plusieurs longueurs différentes afin d'observer l'évolution de la consommation des ressources. Pour du test de robustesse, on choisira plutôt des longueur très importantes pour observer le comportement du système dans des cas extrêmes. Pour du test fonctionnel, l'objectif est de trouver une longueur n qui permet de couvrir tous les états (ou toutes les transitions). Notons que plusieurs longueurs peuvent être choisies : par exemple, lorsqu'un état ne peut être visité que par des traces de longueur impaire alors qu'un autre ne peut être visité qu'avec des traces de longueur paire. Un moyen pratique et fondé sur l'expérience pourrait être de calculer les probabilités (et les grammaires liées) de couvrir chaque état (ou transition), pour des longueurs allant jusqu'à dix fois le nombre d'états du NPDA. Un autre moyen de définir la longueur n pourrait être, par exemple, d'utiliser la procédure suivante (donnée pour le critère AS mais facilement adaptable au critère AT) :

1. Calculer l'ensemble Q_{reach} des états qui peuvent être visités par une NPDA-trace ; certains états peuvent ne pas être accessibles (ils peuvent représenter du code mort, par exemple).
2. Pour tout état $q \in Q_{\text{reach}}$, calculer la longueur n_q de la plus petite NPDA-trace qui visite q.
3. Effectuer l'étape de pré-calcul (de la procédure de génération aléatoire) pour tous les états et tous les k inférieur ou égal à $n = \max\{n_q \mid q \in Q_{\text{reach}}\}$. Prendre, pour valeur de n, la longueur maximale des cas de test.

L'étape (1) peut être effectuée efficacement et de manière automatique à l'aide de travaux existants, comme [BEF+00]. L'étape (2) peut être faite à la volée en même temps que l'étape (1), ou en adaptant l'approche développée dans l'article [BSLS03] de S.Basu et al.. Par conséquent, n est calculable efficacement.

Comparaison avec la génération aléatoire de DFA-traces. Dans le papier [DGG+12] de A.Denise et al., l'approche de test aléatoire sans critère de couverture est appliquée à des graphes qui possèdent quelques milliers d'états et pour des traces de longueurs comparables. Des graphes plus grands ne peuvent pas être traités directement avec cette approche, en raison des ressources mémoire importantes qui sont nécessaires pour stocker la table de pré-calcul. Pour résoudre ce problème, l'approche a été adaptée pour gérer implicitement de grands graphes définis par une synchronisation de petits graphes. L'approche à base de NPDA proposée dans ce papier ne peut

pas traiter de modèles aussi grands. Cependant, comme expliqué dans la section 9.3, sur les modèles à pile l'approche de A.Denise et al. ([DGG+12]) qui ne tient pas compte des opérations sur la pile produit de nombreux chemins non-cohérents. Un modèle à pile peut être étendu à un graphe dans lequel tous les chemins sont correctes par rapport aux opérations de la pile : il *suffit* de calculer le graphe des configurations NPDA tel que défini dans la section 2.2.2. Toutefois, ce graphe est généralement infini. Il peut être approximé en limitant la profondeur de la pile. Cependant, pour $\mathcal{A}_{\text{modulo}}$, avec 4 symboles de pile et 18 états, et avec une profondeur de pile limitée à 5, le graphe correspondant peut avoir jusqu'à $18 * 4^5 = 18432$ nœuds. Même si plusieurs de ces nœuds/configurations sont inaccessibles, la technique consistant à limiter la profondeur de la pile – ou, de façon équivalente pour le test structurel, à limiter le nombre d'appels de fonctions – conduit à d'énormes graphes qui ne peuvent pas être traités directement dans le cadre de l'approche [DGG+12] de A.Denise et al..

Limites et forces des approches proposées. Les résultats expérimentaux présentés dans la section 9.3 montrent que pour la génération aléatoire uniforme des cas de test, l'étape coûteuse est le calcul de la grammaire algébrique nettoyée. Pour l'automate $\mathcal{A}_{\text{SY}}$, le calcul d'une probabilité $pr(p, q)$ peut être fait en environ 110 minutes. L'automate lié $\mathcal{A}_{\text{SY}}^{p,q}$ possède 96 états et 144 transitions. Même si ce temps de calcul peut être amélioré grâce à une meilleure implémentation, la taille maximale d'un NPDA, pour qu'il soit exploitable dans un temps réduit avec l'approche de test aléatoire uniforme, est d'approximativement une centaine d'états et de transitions.

Pour la méthode de test aléatoire biaisé (algorithmes 2 ou 5 de la section 9.3.3), le calcul d'une grammaire – pour chaque état ou chaque transition – est nécessaire. C'est possible, pour tous les exemples étudiés dans la section 9.3, même si la taille de $\mathcal{A}_{\text{SY}}$ semble être proche de la taille maximum acceptable pour un traitement en temps raisonnable. Notons également que cette limite correspond à la taille de NPDA construits automatiquement dans des travaux récents ([ST11]) qui utilisent l'outil PuMoX [ST12], pour de nombreux exemples industriels.

L'approche qui consiste à résoudre un système de programmation linéaire afin d'optimiser la probabilité de satisfaire totalement un critère de couverture prend beaucoup de temps. Les résultats expérimentaux montrent qu'elle est limitée au critère *AS* eu aux automates avec quelques états, comme $\mathcal{A}_{\text{xpath}}$. Notons tout de même que plusieurs exemples industriels étudiées dans la littérature (par exemple dans [ST11]) ont des tailles de cet ordre. De plus, plusieurs optimisations peuvent être envisagées afin de résoudre ce problème avec des NPDA plus grands. On peut donc espérer pouvoir traiter des NPDA de la taille de $\mathcal{A}_{\text{modulo}}$, mais pour cela il y a encore du travail à faire.

En utilisant des approches aléatoires biaisées qui consistent à générer des traces qui visitent un élément non encore couvert (algorithmes 2 et 4 de la section 9.3.3) permet de réduire significativement la taille des suites de tests. De plus, ces deux algorithmes ne nécessitent pas de temps de calcul coûteux. Le fait d'optimiser la probabilité minimale de couvrir un élément en résolvant un système linéaire est certes plus coûteux, mais lorsque les probabilités de couvrir les éléments ne sont pas bien réparties (par exemple, pour les états de $\mathcal{A}_{\text{xpath}}$), cette approche peut être vraiment fructueuse. Inversement, lorsque les probabilités de couvrir les éléments sont bien réparties, le bénéfice est négligeable (par exemple, pour les transitions de $\mathcal{A}_{\text{xpath}}$). L'efficacité de cette approche dépend donc fortement de la topologie du NPDA.

10

Conclusion de la partie III

Dans cette partie nous avons tout d'abord présenté et développé une méthode qui permet d'exploiter à la fois critère de couverture et test aléatoire, dans le contexte de la génération de tests à partir de grammaire. Cette méthode automatique consiste à construire une grammaire, puis à résoudre un système à contraintes linéaire, ce qui peut être effectué par des outils existants, y compris pour de grandes valeurs. Á l'avenir, cette méthode pourrait être étendue à d'autres critères de couverture comme la couverture de règles. Elle pourrait également être étendue à d'autres types de grammaires comme des grammaires attribuées, afin d'ajouter de la sémantique aux langages algébriques.

Nous avons ensuite présenté et développé une approche de test aléatoire à partir de modèles à pile. Bien que cette approche n'est pas aussi efficace que l'approche proposée dans le papier [DGG+12], de A.Denise et al., elle reste traitable (complexité polynomiale). De plus, elle augmente fortement les chances de calculer des chemins qui correspondent à de vraies exécutions. Dans le contexte du test structurel, un exemple montre l'importance de traiter les opérations de la pile afin de modéliser les appels récursifs de fonctions. Dans le contexte du MBT, l'approche est fructueuse pour les programmes ou les systèmes qui gèrent une pile, comme l'algorithme Shunting-Yard. Nous avons fourni une étude qualitative et quantitative des procédures de test, et expliqué comment utiliser de la génération aléatoire biaisée afin de combiner test aléatoire et critères de couverture de façon optimisée. Une perspective intéressante serait d'adapter la méthode et les outils développés afin de générer aléatoirement des cas de tests avec une distribution donnée par des informations statistiques, autrement dit de faire du test statistique.

IV

Conclusions et perspectives

11

Conclusions et perspectives

Nous présentons dans ce chapitre les conclusions sur le travail effectué au cours de cette thèse, ainsi que les pistes de recherche qui en découlent.

Dans la partie II (Approximations régulières pour l'analyse d'accessibilité), nous avons défini et présenté deux nouvelles techniques d'approximation pour le model-checking régulier (*cf.* problème 3.3, page 40). Ces techniques utilisent des algorithmes polynomiaux – à condition d'utiliser les algorithmes récents pour tester l'inclusion des langages de deux automates, comme dans [DR10, ACH$^+$10, BP12] –, et ne sont pas liées au spécificités d'un domaine en particulier (elles sont applicables à tous types de problèmes réguliers). Nos propositions – contrairement à [BJNT00, BHV04] dont elles sont inspirées – n'exigent pas de minimiser/déterminiser l'automate obtenu à chaque étape du model-checking régulier. Nous avons publié ces travaux dans [DHK13a]. Comme une direction possible de ces travaux, nous pouvons envisager de les adapter aux langages d'arbres, comme l'ont fait par exemple les auteurs de [BJNT00, BHV04] ; car après avoir défini une méthode de vérification qui fonctionne sur les mots, vient naturellement l'envie de la confronter à un modèle plus puissant comme les arbres. Mais avant cela, nous n'avons pas épuisé les idées d'améliorations de notre méthode sur les mots... Nous pensons que notre approche de raffinement doit pouvoir être améliorée, sur la précision des approximations comme sur le temps de calcul. Par exemple, pour la technique dont la fonction d'approximation est l'automate C, nous raffinons actuellement en dupliquant simplement l'état p de C qui correspond à la classe d'équivalence des états que l'on ne souhaite plus fusionner. Au lieu de ça, nous pourrions essayer de dupliquer l'ensemble de l'automate C, afin de séparer également les états de C qui suivent l'état p – nous avons nommé cette idée *raffinement par couleur*. Une autre idée que nous n'avons pas eu le temps d'approfondir est d'éviter de fusionner entre eux des états qui étaient déjà présents dans l'automate précédent, afin que l'automate $\mathcal{A}_i$ ressemble davantage à l'automate $\mathcal{A}_{i-1}$, et d'augmenter ainsi les chances d'atteindre un point fixe. Et ensuite, une autre direction envisagée est de généraliser encore davantage nos mécanismes d'approximation, afin de les appliquer à d'autres problèmes de model-checking régulier, comme les systèmes à compteur(s) (*cf.* [BFL04]) ou encore les systèmes à pile (*cf.* [EHRS00]).

Dans la partie III (Combiner génération aléatoire et critère de couverture, à partir d'une grammaire algébrique ou d'un automate à pile), nous avons présenté et développé deux méthodes automatiques qui permettent d'exploiter à la fois critère de couverture et test aléatoire. La première permet de générer des tests à partir d'une grammaire, et la seconde à partir d'un automate à pile. L'avantage de générer des tests à partir d'une grammaire algébrique ou d'un automate à pile, plutôt qu'à partir d'un graphe, est que les chances de calculer des chemins qui correspondent à de vraies exécutions dans le système réel augmentent fortement. La méthode de génération de tests à partir d'une grammaire consiste à construire une grammaire, puis à résoudre un système à contraintes linéaire, ce qui peut être effectué par des outils existants, y compris pour de grandes valeurs. Le

critère de couverture traité est *Tous les symboles non-terminaux* de la grammaire. Á l'avenir, cette méthode pourrait être étendue à d'autres critères de couverture comme la couverture de règles, ou à d'autres types de grammaires comme des grammaires attribuées, afin d'ajouter de la sémantique aux langages algébriques. La génération de tests à partir d'un automate à pile s'appuie sur la génération de tests à partir d'une grammaire, grâce à la transformation de l'automate à pile en une grammaire, et à une bijection entre l'ensemble des arbres de dérivation d'une grammaire et l'ensemble des traces d'un automate à pile (*cf.* théorème 2.15, page 33). Il est à noter que dans ce cas, les critères de couverture traités sont *Tous les états* de l'automate à pile, et *Toutes les transitions* de l'automate à pile. Une perspective intéressante de ces travaux serait d'adapter la méthode et les outils développés afin de générer aléatoirement des cas de tests avec une distribution donnée par des informations statistiques, autrement dit de faire du test statistique.

Bibliographie

[ACH+10] P.A. Abdulla, Y.-F. Chen, L. Holík, R. Mayr, and T. Vojnar. When simulation meets antichains. In Esparza and Majumdar [EM10], pages 158–174.

[ADH+89] H. Agrawal, R. Demillo, R. Hathaway, W. Hsu, W. Hsu, E. Krauser, R. J. Martin, A. Mathur, and E. Spafford. Design of mutant operators for the c programming language. Technical report, Citeseer, 1989.

[AJMd02] P.A. Abdulla, B. Jonsson, P. Mahata, and J. d'Orso. Regular tree model checking. In E. Brinksma and K. G. Larsen, editors, *Proceedings of the 14th international conference on Computer Aided Verification, CAV'02*, volume 2404 of *LNCS*, page 452–466. Springer, 2002.

[AJNd03] P.A. Abdulla, B. Jonsson, M. Nilsson, and J. d'Orso. Algorithmic improvements in regular model checking. In Hunt and Somenzi [HS03], pages 236–248.

[AM04] R. Alur and P. Madhusudan. Visibly pushdown languages. In *Proceedings of the 36th Annual ACM Symposium on Theory of Computing, STOC'04*, pages 202–211. ACM, 2004.

[AO08] P. Ammann and J. Offutt. *Introduction to Software Testing*. Cambridge University Press, UK, 2008.

[AV08] T.L. Alves and J. Visser. A case study in grammar engineering. In D. Gasevic, R. Lämmel, and E. Van Wyk, editors, *SLE*, volume 5452 of *LNCS*, pages 285–304, Toulouse, France, 2008. Springer.

[BCHK08] Y. Boichut, R. Courbis, P.-C. Héam, and O. Kouchnarenko. Finer is better : Abstraction refinement for rewriting approximations. In A. Voronkov, editor, *Proceedings of the 19th international conference on Rewriting Techniques and Applications, RTA'08*, volume 5117 of *LNCS*, pages 48–62. Springer, 2008.

[BEF+00] A. Bouajjani, J. Esparza, A. Finkel, O. Maler, P. Rossmanith, B. Willems, and P. Wolper. An efficient automata approach to some problems on context-free grammars. *Information Processing Letters (IPL)*, 74(5-6) :221–227, June 2000.

[Bei95] B. Beizer. *Black-Box Testing : Techniques for Functional Testing of Software and Systems*. John Wiley & Sons, New York, USA, 1995.

[BF12] A. Bauer and Y. Falcone. Decentralised LTL monitoring. In D. Giannakopoulou and D. Méry, editors, *FM*, volume 7436 of *LNCS*, pages 85–100. Springer, 2012.

[BFL04] S. Bardin, A. Finkel, and J. Leroux. Faster acceleration of counter automata in practice. In K. Jensen and A. Podelski, editors, *Proceedings of the 10th international conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'04*, volume 2988 of *LNCS*, pages 576–590. Springer, 2004.

[BFLP03] S. Bardin, A. Finkel, J. Leroux, and L. Petrucci. FAST : Fast Acceleration of Symbolic Transition systems. In Hunt and Somenzi [HS03], pages 118–121.

[BHV04] A. Bouajjani, P. Habermehl, and T. Vojnar. Abstract regular model checking. In *Proceedings of the 16th international conference on Computer Aided Verification, CAV'04*, volume 3114 of *LNCS*, page 378–379, 2004.

[BJNT00] A. Bouajjani, B. Jonsson, M. Nilsson, and T. Touili. Regular model checking. In Emerson and Sistla [ES00], page 403–418.

[BKe08] C. Baier, J.P. Katoen, and Inc ebrary. *Principles of model checking*, volume 950. MIT press, 2008.

[BLW03] B. Boigelot, A. Legay, and P. Wolper. Iterating transducers in the large. In Hunt and Somenzi [HS03], page 223–235.

[Boi12] B. Boigelot. Domain-specific regular acceleration. *International Journal on Software Tools for Technology Transfer, STTT*, 14(2) :193–206, 2012.

[BP12] F. Bonchi and D. Pous. Checking NFA equivalence with bisimulations up to congruence. Technical report, January 2012. 13p.

[BSLS03] S. Basu, D. Saha, Y.-J. Lin, and S. A. Smolka. Generation of all counter-examples for push-down systems. In *Proceedings of the 23rd IFIP WG 6.1 international conference, FORTE'03*, volume 2767 of *LNCS*, pages 79–94. Springer, 2003.

[BT11] A. Bouajjani and T. Touili. Widening techniques for regular tree model checking. *International Journal on Software Tools for Technology Transfer, STTT*, page 1–21, 2011.

[CGJ+00] E.M. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith. Counterexample-guided abstraction refinement. In Emerson and Sistla [ES00], pages 154–169.

[CGN+05] C. Campbell, W. Grieskamp, L. Nachmanson, W. Schulte, N. Tillmann, and M. Veanes. Testing concurrent object-oriented systems with spec explorer. In J. Fitzgerald, I. J. Hayes, and A. Tarlecki, editors, *FM'05*, volume 3582 of *LNCS*, pages 542–547, Newcastle, UK, July 18-22 2005. Springer.

[CGP00] E.M. Clarke, O. Grumberg, and D. Peled. *Model Checking. 2000.* MIT press, 2000.

[CL05] D. Coppit and J. Lian. Yagg : an easy-to-use generator for structured test inputs. In Redmiles et al. [REZ05], pages 356–359.

[Cla08] R. Clark. Querying streaming xml using visibly pushdown automata. Technical Report UIUCDCS-R-2008-3008, University of Illinois at Urbana-Champaign, 2008.

[DDGM07] B. Daniel, D. Dig, K. Garcia, and D. Marinov. Automated testing of refactoring engines. In *Proceedings of the ACM SIGSOFT symposium on the Foundations of Software Engineering, ESEC/FSE'07*, New York, NY, USA, September 2007. ACM Press.

[DDZ98] A. Denise, I. Dutour, and P. Zimmermann. Cs : a mupad package for counting and randomly generating combinatorial structures. In *Proceedings of the 10-th international conference on Formal Power Series and Algebraic Combinatorics, FPSAC'98*, pages 195–204, Toronto, Canada, 1998.

[DGG+12] A. Denise, M.-C. Gaudel, S.-D. Gouraud, R. Lassaigne, J. Oudinet, and S. Peyronnet. Coverage-biased random exploration of large models and application to testing. *International Journal on Software Tools for Technology Transfer, STTT*, 14(1) :73–93, 2012.

[DHK13a] A. Dreyfus, P.-C. Héam, and O. Kouchnarenko. Enhancing approximations for regular reachability analysis. In *Implementation and Application of Automata*, pages 331–339. Springer, 2013.

[DHK13b] A. Dreyfus, P.-C. Héam, and O. Kouchnarenko. Random grammar-based testing for covering all non-terminals. In *Software Testing, Verification and Validation Workshops (ICSTW), 2013 IEEE Sixth International Conference on*, pages 210–215. IEEE, 2013.

[DHKM14] A. Dreyfus, P.-C. Héam, O. Kouchnarenko, and C. Masson. A random testing approach using pushdown automata. *Software Testing, Verification and Reliability*, 2014.

[Dij61] E. W. Dijkstra. Algol 60 translation : An Algol 60 translator for the x1 and Making a translator for Algol 60. Technical Report 35, Mathematisch Centrum, Amsterdam, 1961.

[DLH09] F. Dadeau, J. Levrey, and P.-C. Héam. On the use of uniform random generation of automata for testing. *Electr. Notes Theor. Comput. Sci.*, 253(2) :37–51, 2009.

[DLS01] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. In *Proceedings of the 13th international conference on Computer Aided Verification, CAV'01*, page 286–297, 2001.

[DLS02] D. Dams, Y. Lakhnech, and M. Steffen. Iterating transducers. *Journal of Logic and Algebraic Programming*, 52 :109–127, 2002.

[DN81] J.W. Duran and S. Ntafos. A report on random testing. In *Proceedings of the 5th international conference on Software engineering, ICSE'81*, pages 179–183, Piscataway, NJ, USA, 1981. IEEE Press.

[DR10] L. Doyen and J.-F. Raskin. Antichain algorithms for finite automata. In Esparza and Majumdar [EM10], pages 2–22.

[DZ99] A. Denise and P. Zimmermann. Uniform random generation of decomposable structures using floating-point arithmetic. *Theor. Comput. Sci.*, 218(2) :233–248, 1999.

[EDGB12] I. Enderlin, F. Dadeau, A. Giorgetti, and F. Bouquet. Grammar-based testing using realistic domains in php. In G. Antoniol, A. Bertolino, and Y. Labiche, editors, *Proceedings of the 5th International Conference on Software Testing, ICST'12*, pages 509–518. IEEE, 2012.

[EHRS00] J. Esparza, D. Hansel, P. Rossmanith, and S. Schwoon. Efficient algorithms for model checking pushdown systems. In *Proceedings of the 12th international conference on Computer Aided Verification, CAV'00*, page 232–247, 2000.

[EM10] J. Esparza and R. Majumdar, editors. *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems, TACAS'10*, volume 6015 of *LNCS*. Springer, 2010.

[End] I. Enderlin. Hoa project, a set of php libraries. *URL : http ://hoa-project.net*.

[ES00] E. A. Emerson and A. P. Sistla, editors. *Computer Aided Verification, 12th International Conference, CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proceedings*, volume 1855 of *LNCS*. Springer, 2000.

[FPPS11] F. Fioravanti, A. Pettorossi, M. Proietti, and V. Senni. Program specialization for verifying infinite state systems : An experimental evaluation. *Logic-Based Program Synthesis and Transformation*, page 164–183, 2011.

[FS08] P. Flajolet and R. Sedgewick. *Analytic Combinatorics*. Cambridge University Press, UK, 2008.

[FZC94] P. Flajolet, P. Zimmermann, and B. Van Cutsem. A calculus for the random generation of labelled combinatorial structures. *TCS*, 132(2) :1–35, 1994.

[Gau14] M.-C. Gaudel. Le test de logiciel : pourquoi et comment. *1024 – Bulletin de la société informatique de France*, May 2014. Available *http://www.societe-informatique-de-france.fr/bulletin/1024-3*.

[GBR98] A. Gotlieb, B. Botella, and M. Rueher. Automatic test data generation using constraint solving techniques. In *Proceedings of the ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA'98*, pages 53–62, Clearwater Beach, FL, USA, 1998.

[GDB05] A. Gotlieb, T. Denmat, and B. Botella. Constraint-based test data generation in the presence of stack-directed pointers. In Redmiles et al. [REZ05], pages 313–316.

[GGP08] A. Cano Gómez, G. Guaiana, and J.-E. Pin. When does partial commutative closure preserve regularity ? In L. Aceto, I. Damgård, L. Ann Goldberg, M. M. Halldórsson, A. Ingólfsdóttir, and I. Walukiewicz, editors, *Proceedings of the 35th International Colloquium on Automata, Languages and Programming, ICALP'08*, volume 5126 of *LNCS*, pages 209–220, Reykjavik, Iceland, 2008. Springer.

[GJ08] A. Groce and R. Joshi. Random testing and model checking : building a common framework for nondeterministic exploration. In *Proceedings of the 6th international Workshop on Dynamic Analysis, WODA'08*, pages 22–28, New York, NY, USA, 2008. ACM.

[GKL08] P. Godefroid, A. Kiezun, and M.Y. Levin. Grammar-based whitebox fuzzing. In R. Gupta and S. P. Amarasinghe, editors, *PLDI*, pages 206–215, Tucson, AZ, USA, 2008. ACM.

[GKS05] P. Godefroid, N. Klarlund, and K. Sen. DART : directed automated random testing. In *PLDI'05 : Proceedings of the 2005 ACM SIGPLAN conference on Programming language design and implementation*, pages 213–223, New York, NY, USA, 2005. ACM.

[Gra08] C. Grandpierre. *Stratégies de génération automatique de tests à partir de modèles comportementaux UML/OCL.* PhD thesis, PhD thesis, LIFC, Université de Franche-Comté, Besancon, 2008.

[Ham94] R. Hamlet. Random testing. In *Encyclopedia of Software Engineering*, pages 970–978. Wiley, 1994.

[HC83] T. J. Hickey and J. Cohen. Uniform random generation of strings in a context-free language. *SIAM J. Comput.*, 12(4) :645–655, 1983.

[HN11] P.-C. Héam and C. Nicaud. Seed : An easy-to-use random generator of recursive data structures for testing. In *Proceedings of the IEEE 4th International Conference on Software Testing, verification and validation, ICST'11*, pages 60–69. IEEE Computer Society, 2011.

[HO10] M. Hague and C.-H. Luke Ong. Analysing mu-calculus properties of pushdown systems. In *Proceeding of the 17th international SPIN workshop, Enschede, SPIN'10*, volume 6349 of *LNCS*, pages 187–192. Springer, 2010.

[HS03] W. A. Hunt and F. Somenzi, editors. *Computer Aided Verification, 15th International Conference, CAV 2003, Boulder, CO, USA, July 8-12, 2003, Proceedings*, volume 2725 of *LNCS*. Springer, 2003.

[JJ04] C. Jard and T. Jéron. Tgv : theory, principles and algorithms, a tool for the automatic synthesis of conformance test cases for non-deterministic reactive systems. *International Journal on Software Tools for Technology Transfer, STTT*, 6, October 2004.

[JN00] B. Jonsson and M. Nilsson. Transitive closures of regular relations for verifying infinite-state systems. *TACAS*, page 220–235, 2000.

[KMM+97] Y. Kesten, O. Maler, M. Marcus, A. Pnueli, and E. Shahar. Symbolic model checking with rich assertional languages. In O. Grumberg, editor, *Proceedings of the 9th*

International Conference on Computer Aided Verification, CAV'97, volume 1254 of *LNCS*, page 424–435. Springer, 1997.

[Läm01] R. Lämmel. Grammar testing. In H. Hußmann, editor, *FASE*, volume 2029 of *LNCS*, pages 201–216, Genova, Italy, 2001. Springer.

[Leg12] A. Legay. Extrapolating (omega-) regular model checking. *International Journal on Software Tools for Technology Transfer, STTT*, 14(2) :119–143, 2012.

[LGJJ06] T. Le Gall, B. Jeannet, and T. Jéron. Verification of communication protocols using abstract interpretation of fifo queues. In *Algebraic Methodology and Software Technology*, pages 204–219. Springer, 2006.

[LS06] R. Lämmel and W. Schulte. Controllable combinatorial coverage in grammar-based testing. In M. Uyar, A.Y. Duale, and M.A. Fecko, editors, *TestCom*, volume 3964 of *LNCS*, pages 19–38, New York, NY, USA, 2006. Springer.

[LY96] D. Lee and M. Yannakakis. Principles and methods of testing finite state machines - a survey. In *Proceedings of the IEEE*, pages 1090–1123, 1996.

[Mau92] P.M. Maurer. The design and implementation of a grammar-based data generator. *Softw., Pract. Exper.*, 22(3) :223–244, 1992.

[MBN04] K. Eikland M. Berkelaar and P. Notebaert. lp_solve : a mixed interger linear program, 2004. http://lpsolve.sourceforge.net/5.5/.

[McK97] B. McKenzie. Generating string at random from a context-free grammar. Technical Report TR-COSC 10/97, Univ ersity of Canterbury, 1997.

[MX07] R. Majumdar and R.-G. Xu. Directed test generation using symbolic grammars. In R. E. K. Stirewalt, A. Egyed, and B. Fischer 0002, editors, *ASE*, pages 134–143, Atlanta, GA, USA, 2007. ACM.

[Ori05] C. Oriat. Jartege : A tool for random generation of unit tests for java classes. In R. Reussner, J. Mayer, J.A. Stafford, S. Overhage, S. Becker, and P.J. Schroeder, editors, *QoSA/SOQUA*, volume 3712 of *LNCS*, pages 242–256. Springer, 2005.

[OXL99] A.J. Offutt, Y. Xiong, and S. Liu. Criteria for Generating Specification-Based Tests. In *Proceedings of the 5th international conference on Engineering of Complex Computer Systems, ICECCS'99*, pages 119–129, Las Vegas, NV, USA, Oct 1999. IEEE Computer Society.

[PTD06] Y. Ponty, M. Termier, and A. Denise. Genrgens : Software for generating random genomic sequences and structures. *Bioinformatics*, 22(12) :1534–1535, 2006.

[Pur72] P. Purdom. A sentence generator for testing parsers. *BIT*, 12(3) :366–375, 1972.

[REZ05] D. F. Redmiles, Th. Ellman, and A. Zisman, editors. *20th IEEE/ACM International Conference on Automated Software Engineering (ASE 2005), November 7-11, 2005, Long Beach, CA, USA*. ACM, 2005.

[Sch02] S. Schwoon. *Model-Checking Pushdown Systems*. Ph.D. Thesis, Technische Universität München, June 2002.

[Sip96] M. Sipser. *Introduction to the theory of computation*, volume 27, pages 27–29. ACM, New York, NY, USA, 1996.

[ST11] F. Song and T. Touili. Efficient CTL model-checking for pushdown systems. In *CONCUR'11 : Proceedings of the 22nd International Conference on Concurrency Theory*, volume 6901 of *LNCS*, pages 434–449. Springer, 2011.

[ST12] F. Song and T. Touili. PuMoC : a CTL model-checker for sequential programs. In *ASE'12 : Proceeding of the IEEE/ACM International Conference on Automated Software Engineering*, ACM, pages 346–349, Essen, Germany, 2012.

[TF89] P. Thévenod-Fosse. Software validation by means of statistical testing : Retrospect and future direction. Technical Report 89043, Rapport L.A.A.S., 1989. International Working Con- ference on Dependable Computing for Criti- cal Applications.

[Tou01] T. Touili. Regular model-checking using widening techniques. In *VEPAS*, volume 50 of *ENTCS*, pages 342–356, 2001.

[Tre04] J. Tretmans. Model based testing - property checking for real. In *Keynote address at the international workshop for Construction and Analysis of Safe Secure, and Interoperable Smart devices, CASSIS'04*, 2004. Available *http://www-sop.inria.fr/everest/events/cassis04/program.html*.

[UPL12] M. Utting, A. Pretschner, and B. Legeard. A taxonomy of model-based testing approaches. *Software Testing, Verification and Reliability*, 22(5) :297–312, 2012.

[WB98] P. Wolper and B. Boigelot. Verifying systems with infinite but regular state spaces. In A. J. Hu and M. Y. Vardi, editors, *Proceedings of the 10th international conference on Computer Aided Verification, CAV'98*, volume 1427 of *LNCS*, pages 88–97. Springer, 1998.

[WMMR05] N. Williams, B. Marre, P. Mouy, and M. Roger. Pathcrawler : Automatic generation of path tests by combining static and dynamic analysis. In M. Dal Cin, M. Kaâniche, and A. Pataricza, editors, *EDCC*, volume 3463 of *LNCS*, pages 281–292, Budapest, Hungary, 2005. Springer.

[XZC10] Z. Xu, L. Zheng, and H. Chen. A toolkit for generating sentences from context-free grammars. In *Software Engineering and Formal Methods*, IEEE, pages 118–122, Pisa, Italy, 2010.

[YBI11] F. Yu, T. Bultan, and O. Ibarra. Relational string verification using multi-track automata. *IJFCS*, 22 :290–299, 2011.

[ZW09] L. Zheng and D. Wu. A sentence generation algorithm for testing grammars. In S.I. Ahamed, E. Bertino, C.K. Chang, V. Getov, L. Liu, H. Ming, and R. Subramanyan, editors, *COMPSAC (1)*, pages 130–135, Seattle, Washington, 2009. IEEE Computer Society.

Résumé :

Les travaux de cette thèse contribuent au développement de méthodes automatiques de vérification et de validation de systèmes informatiques, à partir de modèles. Ils sont divisés en deux parties : vérification et génération de tests.

Dans la partie vérification, pour le problème du model-checking régulier indécidable en général, deux nouvelles techniques d'approximation sont définies, dans le but de fournir des (semi-)algorithmes efficaces. Des sur-approximations de l'ensemble des états accessibles sont calculées, avec l'objectif d'assurer la terminaison de l'exploration de l'espace d'états. Les états accessibles (ou des sur-approximations de cet ensemble d'états) sont représentés par des langages réguliers, ou automates d'états finis. La première technique consiste à sur-approximer l'ensemble des états atteignables en fusionnant des états des automates, en fonction de critères syntaxiques simples, ou d'une combinaison de ces critères. La seconde technique d'approximation consiste aussi à fusionner des états des automates, mais à l'aide de transducteurs. De plus, pour cette seconde technique, nous développons une nouvelle approche pour raffiner les approximations, qui s'inspire du paradigme CEGAR (CounterExample-Guided Abstraction Refinement). Ces propositions ont été expérimentées sur des exemples de protocoles d'exclusion mutuelle.

Dans la partie génération de tests, une technique qui permet de combiner la génération aléatoire avec un critère de couverture, à partir de modèles algébriques (des grammaires algébriques, des automates à pile) est définie. Générer les tests à partir de ces modèles algébriques (au lieu de le faire à partir de graphes) permet de réduire le degré d'abstraction du modèle et donc de générer moins de tests qui ne sont pas exécutables dans le système réel. Ces propositions ont été expérimentées sur la grammaire de JSON (JAvaScript Object Notation), ainsi que sur des automates à pile correspondant à des appels de fonctions mutuellement récursives, à une requête XPath, et à l'algorithme Shunting-Yard.

Mots-clés : model-checking régulier, approximations, modèles algébriques, génération de tests aléatoire, critères de couverture

Abstract:

The thesis contributes to development of automatic methods for model-based verification and validation of computer systems. It is divided into two parts: verification and test generation.

In the verification part, for the problem of regular model checking undecidable in general, two new approximation techniques are defined in order to provide efficient (semi-)algorithms. Over-approximations of the set of reachable states are computed, with the objective of ensuring the termination of the exploration of the state space. Reachable states (or over-approximations of this set of states) are represented by regular languages or, equivalently, by finite-state automata. The first technique consists in over-approximating the set of reachable states by merging states of automata, based on simple syntactic criteria, or on a combination of these criteria. The second approximation technique also merges automata states, by using transducers. For the second technique, we develop a new approach to refine approximations, inspired by the CEGAR paradigm (for Counter-Example-Guided Abstraction Refinement). These proposals have been tested on examples of mutual exclusion protocols.

In the test generation part, a technique that combines the random generation with coverage criteria, from context-free models (context-free grammars, pushdown automata) is defined. Generate tests from these models (instead of doing from graphs) reduces the model abstraction level, and therefore allows having more tests executable in the real system. These proposals have been tested on the JSON grammar (JavaScript Object Notation), as well as on pushdown automata of mutually recursive functions, of an XPath query, and of the Shunting-Yard algorithm.

Keywords: regular model-checking, approximations, context-free models, random tests generation, coverage criteria

■ École doctorale SPIM 16 route de Gray F - 25030 Besançon cedex

■ tél. +33 (0)3 81 66 66 02 ■ **ed-spim@univ-fcomte.fr** ■ www.ed-spim.univ-fcomte.fr

UFC UNIVERSITÉ DE FRANCHE-COMTÉ