République algérienne démocratique et populaire
Ministère de l'enseignement supérieur et de la recherche scientifique
Université Amar Télidji Laghouat


Faculté des sciences

Département d'informatique

Thèse en vue de l'obtention du diplôme de doctorat en sciences
Spécialité : Informatique

Thème :

# Algorithmique des automates d'arbres en vue d'un traitement efficace des grandes masses de données

Présenté par : Guellouma Younes

Composition du jury :

| | | | |
|---|---|---|---|
| M. Yagoubi M.Bachir | Prof. | Université de Laghouat | Président |
| M. Ziadi Djelloul | Prof. | Normandie Université, Rouen | Rapporteur |
| Mme. Bouzouad Hadda | Prof. | Université de Laghouat | Rapporteur |
| Mme. Bensaou Nacera | MCA. | USTHB | Examinateur |
| M. Belbachir Hacéne | Prof. | USTHB | Examinateur |
| M. Ouinten Youcef | MCA. | Université de Laghouat | Examinateur |

# Remerciements

À l'iassue de ce travail de thèse, je voudrais remercier toutes celles et tous ceux qui ont contribué à le rendre possible. Tout d'abord, mes deux directeurs de thèse Professeur Ziadi Djelloul et Professeur Bouzouad Hadda, pour avoir dirigé ce travail de recherche d'un doctorant quelque peu ≪atypique≫, avec leurs conseils, leurs patience, et une constante exigence.

Je remercie très respectueusement l'ensemble des membres du jury qui m'ont fait l'honneur d'évaluer ce modeste travail. Je tiens aussi à exprimer ma reconnaissance à :

- Professeur Bruce William Watson, membre du groupe FASTAR, Stellenbosch, Afrique du sud.
- Dr. Ouali Sebti Nadia et Dr Ludovic Mignot membres du laboratoire LITIS, Rouen, France.
- Professeur Lagraa Nasreddine, directeur du laboratoire LIM, Dr Bensaad Lahcéne, chef de département d'informatique et Dr Kerrache Chaker Abdelazziz pour leur aide dans ma rédaction.
- Dr. Tahar Bendouma, M.Maicha mohamed El Habib, M. Chellama Laradj et M. Rahmoune Abdelaziz.
- M. Bellaouar Slimane, M. Ouled Naoui Slimane, M. Attia Nehar, Mlle. Belabacci Ahlem membres de l'équipe de l'optimisation dans Laboratoire LIM avec lesquels j'ai travaillé étroitement sur plusieurs travaux de recherche.

Je remercie enfin tous ceux et celles qui ont contribué de près ou de loin dans l'accomplissement de ce modeste travail.

# Dédicaces

Je dédie ce modeste travail à la mémoire de mon frère Abdelkader avec lequel je n'aurai pas le plaisir de partagé cet événement mais qui est et qui demeurera toujours dans mon cœur. Je dédie aussi et surtout à ceux qui sont le symbole de courage, ceux à qui je dois tout, ma mère et mon père.

Je le dédie aussi à :

- Ma femme,
- Mes sœurs,
- Ma fille et mon fils,
- Mes oncles et mes tantes,
- A tous mes amis.

Sans oublier tous les enseignants qui ont contribué à ma formation, tous cycles confondus.

# ملخص

تمثل البنى ذاتية التشغيل المخصصة للبنى الشجرية نموذجا نظريا قويا مستعملا في عدة مجالات تتطلب معالجة البيانات الكبيرة مثل تصميم و معالجة اللغات الطبيعية و التحقق الرياضي. من اجل تحقيق التوازن بين الجانبين النظري و التطبيقي، نتناول في هذه المذكرة العديد من الجوانب النظرية للبنى ذاتية التشغيل بالنسبة للبنية الشجرية. بغاية الحد من الفجوة الموجودة بين هذا النموذج النظري و استعمالاته في التطبيقات. سوف نقوم في هذه المذكرة بمعالجة اشكاليتين: الاولى تتمثل في اقتراح خوارزميتين جديدتين لايجاد اصغر بنية مكافئة للبنية ذاتية التشغيل محل البحث. أما الاشكالية الثانية فتتمثل في تحويل البنية ذاتية التشغيل الى عبارة منتظمة مكافئة. الوجه الثاني لاهتماماتنا هو ذو طابع تطبيقي. و في اطاره قمنا بتصميم و إنجاز مجموعة أدوات تعتمد على التصنيف الخوارزمي للبنى الشجرية و البنى ذاتية التشغيل بالإضافة للعبارات المنتظمة.

إهتمامنا في هذا البحث خص محورين أساسين. في الأهتمام الأول، ذي الطابع الخوارزمي النظري، قمنا بمعالجة إشكاليتين هما ايجاد اصغر بنية مكافئة للبنية ذاتية التشغيل محل البحث و تحويل لبنية ذاتية التشغيل لبنية خاصة بالسلاسل الحرفية لعبارات منتظمة. فيما يخص ايجاد اصغر بنية مكافئة للبنية ذاتية التشغيل، قمنا باقتراح طريقتين جديدتين : الاولى تستند على نظرة بيانية للبنية ذاتية التشغيل. هذا يسمح باستغلال خصائص البيانات لتسريع عملية تحسين و تصغير البنى ذاتية التشغيل بالنسبة للبنى الشجرية.
أما بالنسبة للطريقة الثانية، فقد قمنا بتحويل البنية ذاتية التشغيل لبنية خاصة بالسلاسل الرمزية. حيث سنقوم بإثبات أن ايجاد اصغر بنية للبنية الجديدة يوافق نظيرتها الخاصة بالبنى الشجرية. فيما يتعلق بالإشكالية الثانية، فقد قمنا بتعميم ثلاث طرائق خاصة بتحويل لبنية ذاتية التشغيل لبنية خاصة بالسلاسل الحرفية لعبارات منتظمة. هذه الطرائق تتمثل في ايجاد و حل جملة معادلات منهجية و ايجاد و حل جملة مسارات مرمزة و أخيرا مفهوم التكامل للعبارات المنتظمة.

أما فيما يخص الإسهام الثاني المتعلق بتصميم و إنجاز صندوق أدوات برمجية يعتمد تقنية التصنيف الخوارزمي فقد قمنا بتعريف تسلسل هرمي لكل بنية من البنى المستعملة إضافة إلى الخوارزميات المنتمية للتصنيف محل الدراسة و أنجزنا ذلك بالنسبة للإشكاليتين المدروستين و هما ايجاد اصغر بنية مكافئة للبنية ذاتية التشغيل و تحويل البنية ذاتية التشغيل إلى بنية خاصة بالسلاسل الحرفية لعبارات منتظمة.

**الكلمات الدلالية**: بنية شجرية. بنية ذاتية التشغيل. عبارة منهجية. التعقيد الحسابي. التصنيف الخوارزمي. أدوات برمجية.

***Résumé***

Les automates d'arbres constituent un outil théorique puissant utilisé dans plusieurs domaines, exigeant la manipulation de grosses masses de données. Ces derniers incluent la modélisation XML, le traitement des langages naturels et la vérification formelle. Afin de réduire l'écart entre cet outil théorique et son utilisation dans la pratique, nous traitons dans cette thèse différents aspects algorithmiques liés à une manipulation efficace des automates d'arbres.

Nous nous somme focalisé sur deux aspects. Dans le premier, d'ordre algorithmique, nous avons traités deux problèmes majeurs : la minimisation et la transformation d'un automate en une expression rationnelle équivalente. Le second aspect concerne la conception et l'implémentation d'une boite à outils.

Concernant la minimisation d'automates d'arbres, deux techniques ont été proposées afin d'améliorer la complexité temporelle et spatiale. Ces deux techniques sont considérées comme des méta-méthodes vu qu'elles peuvent être personnalisées selon le type de l'automate traité ainsi que la variante de minimisation. La première approche consiste à déployer une représentation utilisant la structure de graphe liée à l'automate en question. Ainsi, des propriétés liées aux graphes sont appliquées pour accélérer le processus de minimisation. La seconde approche transforme l'automate d'arbre en un automate de mots dont la minimisation coïncide avec celle de l'automate d'arbres.

Concernant le problème de transformation d'automates d'arbres en expressions rationnelles, très peu étudié dans la littérature, nous proposons trois variantes représentant des généralisations de techniques analogues à celles des automates de mots. Elles se basent respectivement sur la formulation et la résolution d'un système d'équations rationnelles, d'un système d'étiquetage de chemins d'états et sur la notion d'intégration d'expressions rationnelles.

Quant au deuxième aspect, nous avons conçu une nouvelle boite à outils à base de taxonomies. Cette boite à outils se veut ouverte, extensible et facile à utiliser. Pour cette raison, nous nous sommes basés sur une hiérarchie de structures de données afin de répondre aux exigences des utilisateurs. Une hiérarchie d'algorithmes basés sur les taxonomies est aussi définie. En plus, nous avons décrit l'implémentation de deux taxonomies d'algorithmes visées par notre étude qui sont la minimisation d'automates d'arbres et la transformation d'un automate d'arbres en expression rationnelle.

**Mots Clés :** Automates d'arbres, Expressions rationnelles, Minimisation, Complexité, Boite à outils, Classification d'algorithmes.

### *Abstract*

Tree automata are powerful theoretical tool used in many fields requiring big data analysis. Among those are, for example, XML modeling, natural language processing and model checking. We deal in this thesis with different algorithmic aspects of tree automata manipulation in order to reduce the gap between this theoretical tool and its use in practice.

We are interested in two sides. In the first one, we have treated two main problems : tree automata minimization and the transformation of tree automata into an equivalent rational expression. The second side is to design and implement a taxonomy-based toolkit for trees, tree automata and rational expressions treatment.

Concerning minimization, we proposed two techniques making possible the improvement of time and space complexities. Indeed, these two techniques can be seen as meta-methods given that they can be customized according the tree automata type as well as to the targeted minimization variant. The first proposal is achieved by transforming the tree automaton into a bipartite graph. Thereby, graph properties can be deployed on tree automata in order to accelerate the minimization process. The second one transforms the tree automaton to be minimized into a string one. The minimization is done then through the two new structures.

Regarding the problem of the transformation of tree automata into rational expressions, we proposed three variants which are basically generalizations of analogous techniques on strings. These the proposals are based respectively on formulation and resolution of rational equations system, labeling system and rational expression integration notion.

The second contribution deals with implementation of a new taxonomy-based toolkit. Therefore, we used a hierarchy of data structures and for algorithms belonging to a given taxonomy. We implement two taxonomies about the problems discussed by this dissertation, namely minimization and transformation of tree automata into equivalent rational expressions.


**Keywords :** Tree automata, Rational expressions, Minimization, Complexity, Algorithm Taxonomy, Tree Automata Toolkit.

# Table des matières

**Remerciements** i

**Abstract** iii

**Table des Figures** ix

**Introduction générale** 1

**1 Généralités sur les arbres** 7
- 1.1 Notions et définitions sur les arbres . . . 7
  - 1.1.1 Alphabet, mots et langages . . . 9
  - 1.1.2 Arbres rangés et non-rangés . . . 10
  - 1.1.3 Étiquetage d'arbres . . . 12
- 1.2 Opérations sur les arbres . . . 14
  - 1.2.1 La substitution . . . 14
  - 1.2.2 L'homomorphisme . . . 15
  - 1.2.3 L'isomorphisme . . . 16
- 1.3 Langages d'arbres . . . 17
  - 1.3.1 Opérations sur les langages d'arbres . . . 17
  - 1.3.2 La fermeture de Kleene . . . 19
- 1.4 Les expressions rationnelles d'arbres . . . 20
- 1.5 Arbres et multiplicités . . . 22
  - 1.5.1 Séries d'arbres . . . 22
  - 1.5.2 Expression rationnelles avec multiplicité . . . 23
- 1.6 Conclusion . . . 23

**2 Automates d'arbres rangés** 25
- 2.1 Automates d'arbres rangés à états finis ascendants . . . 25
  - 2.1.1 Langages reconnaissables . . . 28
  - 2.1.2 Quelques opérations sur les automates . . . 31
  - 2.1.3 Déterminisation . . . 33
- 2.2 Autres types d'automates d'arbres . . . 35
  - 2.2.1 Automates d'arbres rangés à états finis descendants . . . 35
  - 2.2.2 Automates d'arbres non rangés . . . 36
  - 2.2.3 Automates à haie . . . 38
  - 2.2.4 Automate à multiplicités . . . 39

2.3 Conclusion . . . 40

**3 État de l'art sur la minimisation** **41**
3.1 Théorème de Myhill-Nerode pour les arbres . . . 42
3.2 Minimisation standard . . . 46
3.2.1 Algorithme standard . . . 47
3.2.2 Implémentation de Carrasco et al. . . . 48
3.3 Minimisation incrémentale . . . 50
3.4 Hyper-minimisation . . . 52
3.5 Construction incrémentale de l'AAFAD minimal . . . 53
3.6 Conclusion . . . 55

**4 Nouvelles méthodes de minimisation** **56**
4.1 Méthode directe . . . 56
4.1.1 Initialisation efficace . . . 59
4.1.2 Application à la minimisation standard . . . 60
4.1.3 Application à la minimisation incrémentale . . . 62
4.2 Méthode indirecte . . . 67
4.2.1 Calcul de l'alphabet pour l'automate de mots . . . 69
4.2.2 Construction de l'automate de mots associé . . . 72
4.2.3 Application à la minimisation standard . . . 74
4.2.4 Application aux autres variantes de minimisation . . . 75
4.3 Conclusion . . . 78

**5 Automates d'arbres et expressions rationnelles d'arbres** **80**
5.1 Méthode de résolution de système d'équations rationnelles . . . 81
5.1.1 Expression rationnelle d'arbres avec variables . . . 82
5.1.2 Système d'équations rationnelles . . . 85
5.1.3 Système d'équations récursif . . . 88
5.1.4 Construction d'une ERA depuis un AAFA . . . 93
5.2 Méthode de résolution de système d'étiquetage . . . 96
5.3 Intégration rationnelle . . . 100
5.4 Algorithme de passage d'un AAFA vers une ERA . . . 105
5.5 Conclusion . . . 108

**6 Boite à outils** **109**
6.1 Boites à outils existantes . . . 109
6.2 Architecture de notre boite à outils . . . 111
6.2.1 Structure globale . . . 111
6.2.2 Format d'interopérabilité . . . 112
6.2.3 Classes d'objets de base . . . 114
6.2.4 Classes d'algorithmes . . . 116
6.2.5 Génération aléatoire . . . 116
6.3 Détails d'implémentation . . . 117
6.4 Conclusion . . . 118

**Conclusion générale** **119**

**Annexe** **129**
.1 Code *D* pour la génération aléatoire . . . . . 129
.2 Code *D* pour la définition d'alphabets, d'état et d'AAFA . . . . . 131
.3 Code *D* pour la définition des AAFA, AAFA rangés et AAFA rangés avec mutiplicités 131

# Table des figures


1 Classification des techniques de minimisation . . . 4
2 Minimisation d'un automate d'arbres moyennant un automate de mots . . . 5
3 Expression rationnelle d'arbres (ERA) $\leftrightarrow$ Automate d'arbre (AA) . . . 5

1.1 Arborescence de fichiers Windows . . . 8
1.2 Arborescence dans un document XML . . . 8
1.3 Représentation graphique de l'arbre $t = f(g(a), f(a, f(g(b), b)))$ . . . 11
1.4 Exemple d'arbre étiqueté . . . 13
1.5 Exemple de l'homomorphisme d'arbres . . . 16

2.1 Exemple d'AAFA . . . 26
2.2 Reconnaissance d'un arbre par l'AAFA A de l'Exemple 2.1 . . . 28
2.3 Arbre non reconnu par un AAFA . . . 28
2.4 Automate avec états inutiles $\{q_1, q_2\}$ . . . 30
2.5 Pompage d'un arbre à l'aide d'un AADA . . . 33
2.6 Déterminisation d'un AAFA . . . 35
2.7 Exemple de document XML . . . 37

3.1 Classification des techniques de minimisation . . . 42
3.2 Minimisation d'AAFA . . . 46

4.1 Classification des techniques de minimisation avec les deux méta-méthodes proposées 57
4.2 Graphe associé à l'automate de l'Exemple 4.1 . . . 58
4.3 Exemple d'un AAFAD . . . 70
4.4 L'automate acyclique associé à $q_1$ . . . 71
4.5 L'automate minimal reconnaissant $L_\Delta$ . . . 71
4.6 L'automate $M_\mathcal{A}$ associé l'AAFAD de l'Exemple 4.3. . . . 73

5.1 Passage ERA $\leftrightarrow$ AAFA . . . 81
5.2 Transformation d'un AAFA en une ERA via systéme d'équations rationnelles . . . 82
5.3 Exemple d'AAFA. . . . 95

6.1 Extrait des modules de la boite à outils . . . 112
6.2 Document décrivant l'arbre $f(g(b), g(a))$ . . . 113
6.3 Document décrivant un AAFA . . . 113
6.4 Document décrivant l'ERA $f(a, g(a)^{*a}) + b$ . . . 113
6.5 Classes principales de la boite à outils . . . 115
6.6 Hiérarchie de la classe AAFA (FTA) . . . 115
6.7 Hiérarchie des algorithmes de minimisation . . . 116

6.8 Expression rationnelle (ERA) $\leftrightarrow$ Automate d'arbre (AA) . . . . . 120

# Liste des algorithmes


1 Réduction d'AAFA . . . . . 30
2 Déterminisation d'AAFA . . . . . 34
3 Minimisation d'AAFAD . . . . . 48
4 Implementation de Carrasco et al. . . . . 50
5 Fonction d'équivalence d'états . . . . . 51
6 Hyper minimisation . . . . . 53
7 Fonction split . . . . . 54
8 Fonction clone . . . . . 55
9 Construction du graphe associé . . . . . 59
10 Minimisation linéarithmique . . . . . 62
11 Nouvelle fonction d'équivalence . . . . . 65
12 Traitement de $Trace$ . . . . . 66
13 Verify stack . . . . . 66
14 Nouvelle fonction main . . . . . 67
15 Calcul de $\sim$ . . . . . 70
16 Filtre AAFAD vers automate de mots . . . . . 73
17 Nouvelle fonction split . . . . . 77
18 Nouvelle fonction clone . . . . . 77
19 Calcul des cycles et des niveaux . . . . . 106
20 Construction d'une ERA à partir d'un AAFA . . . . . 107
21 Générateur basique des AAFA rangés . . . . . 117

# Introduction générale

Durant les dernières années, les chercheurs et les industriels se sont penchés vers la théorie des langages d'arbres et des automates d'arbres afin de solutionner des problèmes complexes. Cette nouvelle tendance se matérialise dans plusieurs domaines tel que la modélisation XML [ZL03, Chi99], le traitement du langage naturel [Tom06], la vérification formelle et l'analyse des programmes. Conçu pour la première fois dans les années soixante [Don68], ce formalisme avait pour but de donner une généralisation de la théorie des langages pour les arbres, faiblement utilisés, aux domaines concrets. En effet, la seule application majeure était la vérification des circuits intégrés.

Durant les années 70, beaucoup de résultats théoriques concernant les automates d'arbres ont été établis. Mais un écart entre cette théorie en développement et la pratique a commencé à se manifester. En particulier, la complexité croissante des solutions données vis-à-vis du problème de "décidabilité" et l'incohérence entre ses résultats et les ressources matérielles de l'époque. Petit à petit, cette théorie a repris sa place dans la pratique. Le retour considérable vers la structure arborescente a suscité de rediscuter, améliorer et appliquer ce genre d'outils formels qui sont les automates d'arbres.

## Problématique

Notre vision à travers cette étude est de bénéficier d'un fondement théorique puissant dans le but de concevoir une algorithmique plus efficace pour les grandes masses de données (Big Data). En effet, une des questions sur laquelle beaucoup d'effort de recherche se focalise est le traitement des grande masse de donnée arborescentes tel que les entrepôts de données XML, architectures de médiation, explosion de la masse des index d'arbres et leurs traitement efficace surtout sur le Web. Par ailleurs, la théorie des langages d'arbres inclut beaucoup de sous domaines : les grammaires d'arbres pour la génération d'arbres, les expressions d'arbres pour dénoter les langages rationnels, les automates d'arbres pour reconnaître les langages réguliers, les langages d'arbres hors contexte et les automates à piles, etc. L'objectif de cette thèse est de développer une algorithmique efficace des automates d'arbres afin de construire un prototype destiné aux développeurs d'applications. Une tendance vers l'algorithmique déterministe est alors envisagée. Pour cette raison, nous nous focalisons sur les automates d'arbres déterministes qui sont plus efficaces dans la pratique. Cette classe d'automates est *régulière* vu sa capacité à reconnaître un grand nombre d'arbre voir une infinité d'arbres (si cette masse suit un motif précis). Nous nous intéressons aussi à l'étude des expressions rationnelles car c'est le moyen par excellence pour représenter un motif (pattern) d'un ensemble d'arbres réguliers (comme les DTD et les XMLs schémas). En outre, il est prouvé que les automates d'arbres et les expressions rationnelles sont équivalents vis-à-vis des langages réguliers et rationnels.

**Automates d'arbres** Les automates d'arbres sont une généralisation des automates de mots. Ils sont parfaitement adaptés pour la modélisation des structures de données arborescentes (pages Web, XML, vérification des programmes, récursivité) et sont de plus en plus utilisés comme modèles en traitement automatique des langages naturels. Les défis posés par l'analyse de grandes masses de données, en extraction automatique en particulier, ont suscité de nouveaux travaux sur l'inférence grammaticale de ces automates. Les langages réguliers de mots et les automates finis qui les reconnaissent sont des outils formels de base pour traiter les séquences de données.

Afin de prendre en compte l'incertitude des données, on peut associer à chaque mot une probabilité ; le langage est alors dit stochastique. Plus généralement on peut associer à chaque mot un élément pris dans un semi-anneau : nous avons alors une série rationnelle. Cette dimension ajoutée permet aux automates d'arbres de reconnaître des séries rationnelles. Les automates ainsi définis sont dits *à multiplicité*.

Les opérations sur les automates d'arbres, avec ou sans multiplicités, sont ainsi couramment utilisées dans les applications relevant de l'ingénierie du Web, de l'ingénierie des documents digitaux, ou encore du traitement automatique des langages naturels. Dans chacun de ces trois domaines d'application, l'explosion du volume de données à traiter a suscité de nouveaux défis concernant des problèmes comme l'extraction automatique de connaissances, ou encore la classification. Dans le cas des mots, il existe une algorithmique efficace des automates et des automates à multiplicités, et de nombreuses boîtes à outils sont disponibles. Citons par exemple les compilateurs XFST et WFSC (Xerox), ou la bibliothèque OpenFst [RAJ09], qui a contribué aux progrès réalisés en traitement automatique des langages naturels. La situation est loin d'être aussi favorable en ce qui concerne les arbres et en particulier les arbres à multiplicités. Des structures de données spécifiques doivent être conçues en particulier pour éviter les risques d'explosion combinatoire. Des choix restent à faire (automate ascendant / automate descendant par exemple), de nombreux algorithmes restent à écrire ou à consolider.

**Expressions rationnelles** Une expression rationnelle d'arbres appelés aussi motifs d'arbres est aussi une généralisation de celle sur les mots. Elle est une chaîne de caractère incluant des opérateurs fondamentaux permettant de décrire -selon une syntaxe précise- un ensemble d'arbres ou bien un langage d'arbres dit *rationnel*. De la même façon, les langages rationnels d'arbres permettent de manipuler les structures arborescentes de données telles que les pages Web. Nous considérons des séries rationnelles d'arbres en associant à chaque arbre un élément pris dans un semi anneau. Lorsque les multiplicités sont des probabilités, la série est dite stochastique.

## Contributions

Dans ce contexte, nous avons apporté principalement trois contributions liées aux automates d'arbres. Les deux premières contributions -qui constituent la partie majeure de ce mémoire- sont d'ordre algorithmique. La troisième -qui est un résultat direct de l'étude algorithmique de la première contribution- est une conception et une implémentation d'une boite à outils manipulant les arbres, les automates d'arbres, les expressions rationnelles ainsi que les liens qui existent entre les deux structures.

Nous nous intéressons en premier lieu dans cette thèse à la minimisation d'automates d'arbres. En effet, montrer qu'une structure compacte et finie, reconnaissant une infinité d'arbres, est la plus petite serait une chose très intéressante dans le traitement des grandes masses de données. Effectivement, la minimisation touche deux aspects : le premier, le plus connu, est la minimisation du nombre d'états et le deuxième est la réduction du nombre de transitions.

En effet, toutes les techniques de minimisation d'automates d'arbres de référence sont largement inspirées de leurs analogues sur les mots qui sont étudiées pour la première fois par Huffman et Moore [Moo56]. Le meilleur algorithme connu est celui de Hopcroft [Hop71]. La complexité donnée est linéarithmique.

Concernant les automates d'arbres, une panoplie de travaux existe. Ils sont basés sur le type de l'automate traité et sur la base de l'algorithme de minimisation. La Figure 1 résume les travaux faits sur la minimisation. Nous classons les techniques existantes comme suit :

- Minimisation des automates non-rangés : ce type d'automate permet à un type de nœud d'arborescence d'avoir un nombre variable de fils. Nous distinguons deux familles d'algorithmes :
  - Minimisation d'automates à haie : automates opérant sur des transitions sous la forme de *motifs* qui sont dénotés sous forme d'expressions rationnelles de mots (voir Raeymaekers et Bruynooghe [RB04], Cristau et al. [CLT05]).
  - Minimisation d'automates à pas : automates d'arbres qui sont composés d'automates élémentaires de mots. Ces derniers sont utilisés pour identifier les sous arbres d'un nœud donné. (Carme et al. [CNT04]).
- Minimisation d'automates rangés : des automates ayant un nombre fixe de fils pour chaque type de nœuds dans l'arborescence. C'est le type d'automates le plus discuté dans la littérature. Nous classons ses méthodes selon la nature de l'automate produit en deux classes :
  1. Méthodes exactes : toute méthode de minimisation produisant l'automate minimal unique. Nous citons par exemple les travaux de Brainerd [Bra67], Arbib et Give'on [AG68] et Gécseg et al. [GS80].
  2. Méthodes approchées : cette classe contient deux types de méthodes :la première s'intéresse aux techniques produisant un automate qui est plus compacte que l'automate à minimiser mais qui n'est pas nécessairement le minimal. Quant à la deuxième, elle regroupe les techniques qui minimisent un automate mais sans donner des garanties pour que le langage de ce dernier soit exactement le même avec celui de départ. Par exemple, Cleophas et al. [CKSW09] propose un algorithme qui construit un automate avec moins d'états qui reconnaît le même langage avec l'automate en question, mais qui n'est pas forcement l'automate minimal. Par contre, Jez et Maletti [JM12] construisent un automate qui est plus petit que l'automate minimal mais le langage de ce dernier diffère légèrement de celui de l'automate en question.

Nous contribuons dans la conception de deux techniques afin de minimiser un automate d'arbres déterministe. En effet, les deux techniques proposées peuvent être vues comme des *méta-méthodes* pour les techniques de minimisation. Le résultat de ces propositions peut être appliqué et adapté pour produire des algorithmes exacts ou bien approchés.

En effet, la première méthode, dite directe (Minimisation via graphe associé dans la Figure 1), transforme l'automate en un graphe biparti. Nous montrons qu'à partir de ce graphe, on peut extraire

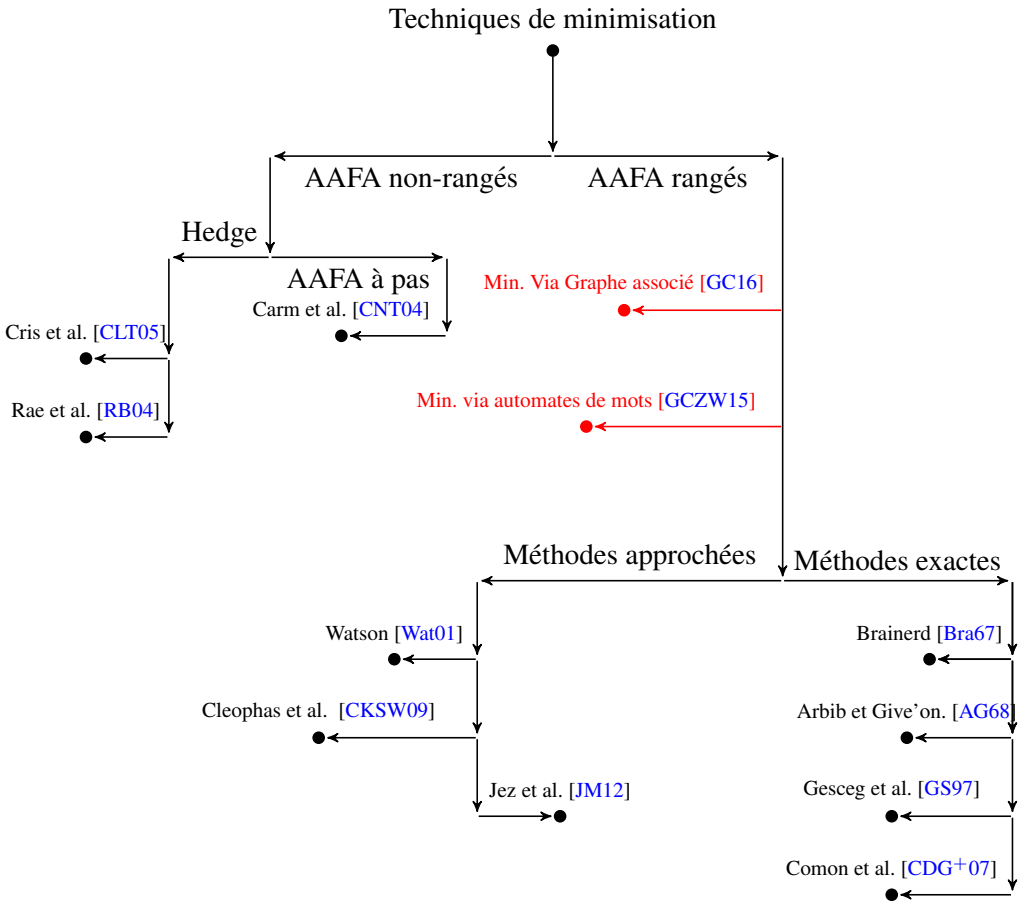



FIGURE 1: Classification des techniques de minimisation

beaucoup de propriétés importantes et ainsi réduire le temps de calcul. Nous montrons que l'utilisation de cette structure produit un algorithme exact linéarithmique et un algorithme approché de l'ordre cubique.

La deuxième technique consiste à bénéficier de l'algorithmique présente dans la littérature afin de la minimisation de mots pour minimiser un automate d'arbres. L'idée consiste à définir un automate de mots à partir de l'automate d'arbres à minimiser. Ensuite, le nouvel automate est minimisé en utilisant une des méthodes définies sur les mots sachant que la minimisation sur les mots est considérée comme étant "bien étudiée". La dernière étape est la déduction directe de l'automate d'arbre minimal à partir de l'automate de mots associé (voir Figure 2). Notons que l'automate résultat est pseudo minimal si la technique utilisée est approchée. Par ailleurs, nous prouvons que malgré le passage par une étape intermédiaire, cette technique donne les meilleures complexités pour la minimisation exacte ou bien la minimisation approchée.

Le deuxième volet abordé par notre étude est le passage depuis les automates d'arbres vers les expressions rationnelles. La Figure 3 résume les travaux existants dans la littérature selon les concepts déployés. Dans notre contribution, nous nous focalisons sur le développement de la branche présentée en pointillé dans la Figure 3.

En effet, les techniques existantes pour la conversion d'une expression rationnelle vers les automates d'arbres sont toutes des généralisations de leurs analogues sur les mots. Premièrement, Kuske

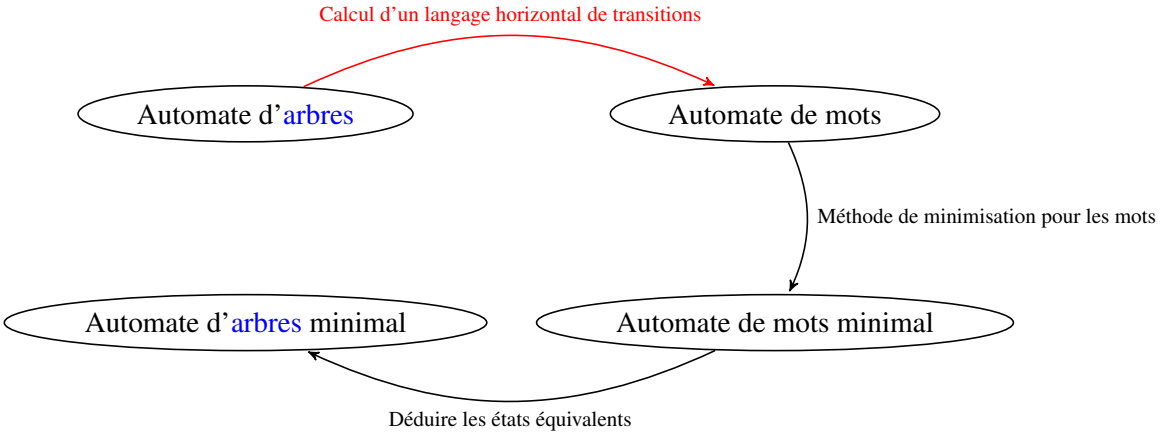



FIGURE 2: Minimisation d'un automate d'arbres moyennant un automate de mots

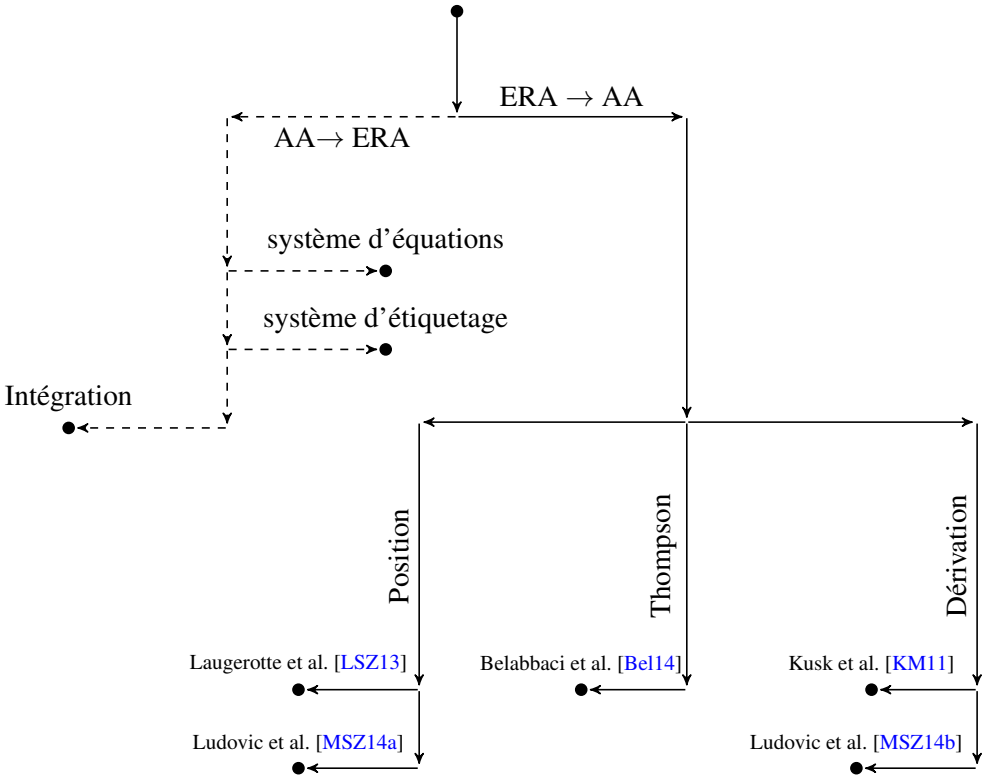



FIGURE 3: Expression rationnelle d'arbres (ERA) ↔ Automate d'arbre (AA)

et Meinecke [KM11] ont généralisé la notion de *dérivées partielles* pour les arbres. Un automate d'équations est alors proposé. Ensuite, Laugerotte et al. [LSZ13] ont travaillé sur la généralisation de l'automate de positions pour les arbres. Enfin, Belabbaci [Bel14] a proposé un automate pour la recherche de motifs d'arbres basé sur la généralisation de l'automate de Thompson. Néanmoins, la technique de résolution de système d'équations (basé sur le lemme d'Arden [Con61]), le système d'étiquetage (connu aussi sous le nom de l'Algorithme de Mcnaughton-Yamada [MY60]) et la notion d'intégration d'expressions rationnelles n'ont pas été bien discutées pour le passage d'un automate d'arbres vers une expression rationnelle. Nous montrons que ces techniques sont envisageables. Ensuite, nous concevons un algorithme qui transforme un automate d'arbres en expression rationnelle.

La dernière contribution est le développement d'une boite à outils orientée applications dont l'intérêt est de favoriser le développement de nombreux projets de recherche. Plus précisément, nous avons considéré les étapes suivantes :

- Analyse des besoins dans les principaux domaines applicatifs.
- Sélection des briques de base.
- Choix des structures de données les plus pertinentes.
- Mise au point d'algorithmes de base. Prototypage.
- Conception des algorithmes de calcul : conversions, opérations, réduction, minimisation. Prototypage.
- Optimisation des complexités.

## Plan de la thèse

Omit cette introduction, ce mémoire de thèse est organisé en six chapitres. Le premier chapitre présente les définitions de notions et notations utiles pour la compréhension des arbres, langages d'arbres et expressions rationnelles qui dénotent la classe des langages réguliers. Le deuxième chapitre est dédié à l'introduction les automates d'arbres. Les différentes notions théoriques liées sont présentées. Nous entamons la minimisation des automates d'arbres dans le troisième chapitre. Le quatrième chapitre est réservé à notre première contribution qui se résume en une proposition de deux approches pour améliorer la complexité du processus de minimisation. Le cinquième chapitre traite la relation entre les automates d'arbres et les expressions rationnelles. Dans le dernier chapitre, nous présentons une description générale de l'outil développé pour la manipulation des automates d'arbres et des expressions rationnelles. Finalement, nous présentons nos conclusions et perspectives de recherche dans la conclusion.

# Chapitre 1

# Généralités sur les arbres

La technologie de l'information (information Technology ou IT) est devenue le centre du développement de l'humanité au vingt et unième siècle. Pour les gouvernements, les entreprises, la société et les individus, l'information est devenue primordiale pour la vie de notre civilisation. L'arrivée de nouvelles technologies tel que le le cloud computing nécessite des avancées technologiques dans le stockage (mémoire et structures de données), le traitement (super calculateurs, indexation, parallélisme) et la transmission (équipements de télécommunication, bande passante). L'information quant à elle est de plus en plus complexe. L'avènement de technologies d'agrégation et de fusion de données hétérogènes comme les ontologies et plus généralement les architectures de médiation et l'avancée des systèmes décisionnels en vue de travailler avec des données hétérogènes a changé la vision envers les structures de données.

Étant donnée une structure naturelle, l'arbre qui est une structure de données classique et ancienne dans l'algorithmique est largement utilisé dans le monde de l'IT. La structure arborescente permet de stocker une donnée d'une manière naturelle, équilibrée en mémoire, favorisant un temps d'accès raisonnable.

Comme premier exemple classique de l'usage des arbres, nous considérons les systèmes d'exploitation. En effet, ils organisent leurs systèmes de fichiers dans des structures arborescente (voir Figure 1.1)

Un deuxième exemple plus significatif est l'utilisation de XML (eXtended Markup Language). Un document XML est présenté sous une forme arborescente (Voir Figure 1.2).

La structure arborescente est une structure fondamentale en algorithmique [Cor02]. Elle n'est pas utilisée seulement (B-arbres par exemple [Bay71]) pour représenter les données mais aussi pour accélérer les temps de calcul, optimiser la recherche d'information (algorithme Branch and Bound [LD60]), trouver des heuristiques etc.

## 1.1 Notions et définitions sur les arbres

Dans cette section, nous allons introduire les notions et les définitions nécessaires pour la bonne compréhension du reste du mémoire.

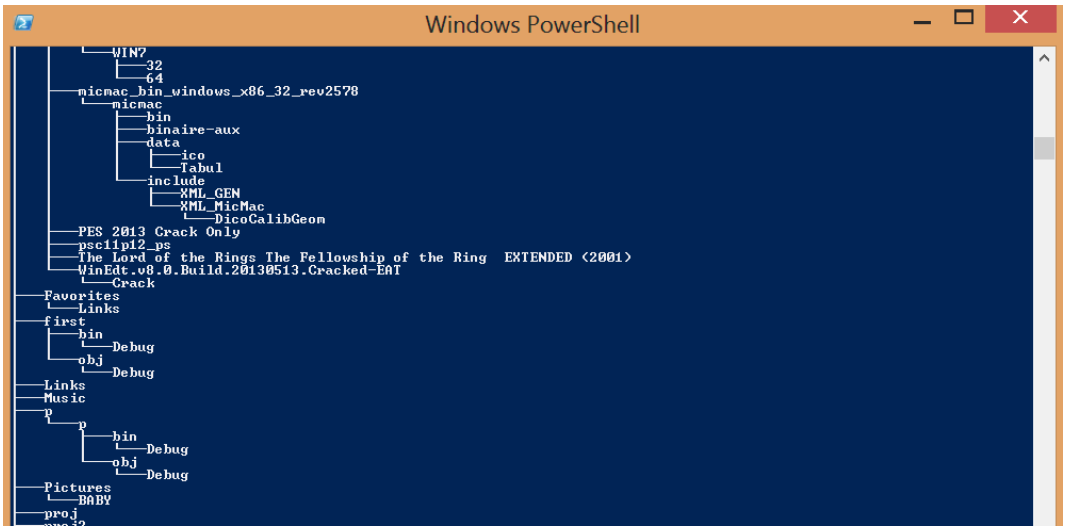



FIGURE 1.1: Arborescence de fichiers Windows

```
<parent>
  <sibling1> ... </sibling1>
  <sibling2> ... </sibling2>
  <self>
    <child1> ... <desc1></desc1> ... <desc2></desc2> ... </child1>
    <child2> ... </child2>
    <child3> ... <desc3><desc4> ... </desc4></desc3> ... </child3>
  </self>
  <sibling3> ... </sibling3>
</parent>
```

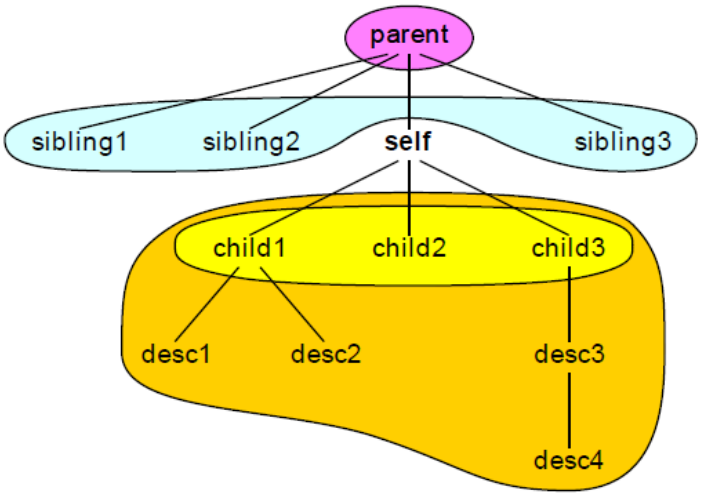



FIGURE 1.2: Arborescence dans un document XML

Omit nos propres définitions, la plupart des notions, notations et définitions présentées dans ce chapitre sont principalement compilées à partir du document de référence “tata” [CDG+07]. D’autres référence sont aussi utilisées comme [Cle08, Mal09, K.S10, Pun09].

### 1.1.1 Alphabet, mots et langages

Afin de faire l’analogie avec la théorie des langages, nous commençons par rappeler quelques notions basiques mais nécessaire pour normaliser les formalismes [K.S10, Pun09].

*Un alphabet* est un ensemble fini de symboles. Un symbole est appelé “lettre”.
*Un mot* est une suite finie (éventuellement vide) de lettres.
*Le mot vide* est noté par convention $\varepsilon$.
*L’étoile de Kleene* $\Sigma^*$ (resp. $\Sigma^+$) est l’ensemble de tous les mots (resp. l’ensemble de tous les mots non-vides) qu’on peut construire à partir de l’alphabet $\Sigma$. En effet, l’étoile de Kleene peut être construite comme suit :

- $\varepsilon \in \Sigma^*$ est un symbole spécial appelé le mot vide,
- $\Sigma \subset \Sigma^*$,
- si $u, v \in \Sigma^*$ alors $uv, vu \in \Sigma^*$.

*La longueur d’un mot* $u$ notée $|u|$ est égale au nombre total des lettres de $u$. Par conséquent, $|\ |$ est une application de $\mathbb{N} \to \Sigma^*$ qui est définie par induction comme suit :

- $|\varepsilon| = 0$,
- $\forall a \in \sigma, |a| = 1$,
- Soient $u, v \in \Sigma^*$ alors $|uv| = |u| + |v|$.

*L’opération de concaténation* de deux mots $u, v \in \Sigma^*$ produit un nouveau mot $uv$ constitué par la juxtaposition de lettres de $u$ et de lettres de $v$. Nous avons alors $|uv| = |u| + |v|$. La concaténation est une opération interne de $\Sigma^*$ ; elle est associative, mais pas commutative (sauf dans le cas dégénéré où $\Sigma$ ne contient qu’un seul symbole). $\varepsilon$ est l’élément neutre pour la concaténation : $u\varepsilon = \varepsilon u = u$. Si $u$ se factorise sous la forme $u = xy$, alors on écrira $y = x^{-1}u$.
*La position* $\text{Pos}^u$ d’un nombre entier $i \in \mathbb{N}$ est une application $\mathbb{N} \to \Sigma$ qui retourne le symbole se trouvant à la $i-$éme position d’un mot $u$. *Un langage* est un sous ensemble de $\Sigma^*$ pour un alphabet $\Sigma$ donné. Un langage peut être fini ou infini.

**Exemple 1.1.** *Soit* $\Sigma = \{a, b, c, d\}$. *Alors :*

- $\Sigma^* = \{\varepsilon, a, b, c, ab, ba, ac, ca, abc, acb, \ldots\}$, *soit l’ensemble de tous les mots composés de lettres de* $\Sigma$.
- $\{\varepsilon, a, b, c, d\}, \{a, ab, abc, abcd\}$ *sont deux langages finis.*
- $\{a, aa, aaa, aaaa, \ldots\}$ *est un langage infini qui contient tous les mots composés à partir de la lettre* $a$.
- $|\varepsilon| = 0, |abcd| = |aaaa| = 4$
- $Pos^{abcd}(2) = b$

### 1.1.2 Arbres rangés et non-rangés

Les arbres se distinguent par la nature des nœuds qui les composent. Nous définissons dans cette partie les différentes formes d'arbres présentes dans la littérature.

**Définition 1.1.** *(Arbre non rangé) Soit un alphabet $\Sigma$, l'ensemble de tous les arbres non-rangés $N_\Sigma$ est défini comme suit :*
$\Sigma \subset N_\Sigma$ *et* $\forall f \in \Sigma, t_1, \ldots, t_n \in N_\Sigma, n \in \mathbb{N} : f(t_1, \ldots, t_n) \in N_\Sigma$.

Néanmoins, la définition d'un rang (degré) pour un arbre nécessite l'extension de l'alphabet qui le compose.

**Définition 1.2.** *(Alphabet rangé) Un alphabet $\Sigma$ est dit rangé ou gradué s'il existe une application $rang : \Sigma \to \mathbb{N}$ qui associe à chaque symbole de $\Sigma$ un rang (une arité) . Le rang de $f \in \Sigma$ sera noté $\hat{f}$). La relation de $\Sigma^2$ définie par* rang *est appelée signature. Le rang maximal de $\Sigma$ noté $\hat{r}$ (à ne pas confondre avec le rang d'un symbole $r$) est :*

$$\hat{r} = \max_{f \in \Sigma} \hat{f}$$

*$\Sigma_p, p \in \mathbb{N}$ est l'ensemble de tous les symboles qui ont exactement $p$ fils :*

$$\Sigma_p = \{f \in \Sigma \mid \hat{f} = p\}$$

*Les symboles ayant le rang $0, 1, 2, \ldots, p$ sont respectivement des constantes (feuilles), des symboles unaires, symboles binaires,..., symboles $p$-aires.*

**Exemple 1.2.** *Soit $\Sigma = \{f, g, a, b\}$ un alphabet. La signature $\{(f, 2), (g, 1), (a, 0), (b, 0)\}$ définit le rang sur $\Sigma$. Nous avons :*

- $\hat{f} = 2, \hat{g} = 1, \hat{a} = \hat{b} = 0$.
- $\hat{r} = 2$.

Dans la littérature, nous trouvons diverses notations pour indiquer le rang d'un symbole donné : certains utilisent un indice sur un symbole $p$-aire $f$ qui est son rang ($f_p$). D'autres écrivent respectivement $f, f(), \ldots, f(,\ldots,)$ $p$ fois pour indiquer une constante, un symbole unaire, ..., un symbole $p$-aire respectivement. Dans ce mémoire, nous optons pour la seconde écriture.

**Définition 1.3.** *(Arbre rangé) Soit $\Sigma$ un alphabet rangé. L'ensemble des arbres rangés $T_\Sigma$ définis sur l'alphabet rangé $(\Sigma, rang)$ est le plus petit ensemble défini par :*

1. $\Sigma_0 \subseteq T_\Sigma$,
2. *Si* $p \geq 1, f \in \Sigma_p, t_1, \ldots, t_p \in T_\Sigma$ *alors* $f(t_1, \ldots, t_p) \in T_\Sigma$.

$T_\Sigma$ est l'analogue de $\Sigma^*$ dans les mots.

Nous définissons des fonctions générales liées aux arbres rangés. Ces mêmes fonctions peuvent être généralisées aux arbres non-rangés. Soit $\Sigma$ un alphabet rangé et $t = f(t_1, \ldots, t_{\hat{f}})$ un arbre rangé.

- *La racine* de $t$ est $\mathcal{R}(t) = f$

- *La taille* $\|t\|$ est définie comme suit :
$$\|t\| = \begin{cases} 1 & \text{si } \hat{f} = 0 \\ 1 + \sum_{i=1}^{\hat{f}} \|t_i\| & \text{sinon} \end{cases}$$
La taille d'un arbre est le nombre de ses éléments.
- *La profondeur* de $t$ notée $\mathcal{P}(t)$ est définie comme suit :
$$\mathcal{P}(t) = \begin{cases} 1 & \text{si } \hat{f} = 0 \\ 1 + \max_{i=1,\ldots,\hat{f}}(\mathcal{P}(t)) & \text{sinon} \end{cases}$$
La profondeur d'un arbre est égale à longueur de la branche ayant la taille maximale.
- *L'ensemble des sous arbres* $St(t)$ est défini par : $St(t) = \{t\} \cup \bigcup_{k=1}^{\hat{f}} St(t_k)$.

**Exemple 1.3.** *Soit* $\Sigma = \{f(,), g(), a, b\}$ *un alphabet rangé d'arbres contenant deux constantes* $a, b$*, un symbole unaire* $g$ *et un symbole binaire* $f$*.*

*L'ensemble de tous les arbres construits à partir de* $\Sigma$ *est* $T_\Sigma = \{a, b, g(a), g(b), g(g(a)), g(g(b)), f(a,a), f(a,b), f(b,a), f(b,b), f(g(a),a), f(g(a),b,\ldots\}$

*Soit l'arbre suivant* $t = f(g(a), f(a, f(g(b), b)))$*. La représentation graphique de* $t$ *est schématisée par la Figure 1.3 :*
*Nous avons :*

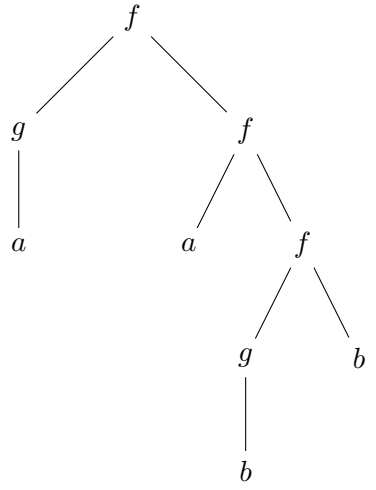



FIGURE 1.3: Représentation graphique de l'arbre $t = f(g(a), f(a, f(g(b), b)))$

- $\mathcal{R}(t) = f$
- $\|t\| = 1 + \|g(a)\| + \|f(a, f(g(b), b))\| = 1 + 1 + \|a\| + 1 + \|a\| + \|f(g(b), b)\| = 9$,
- $\mathcal{P}(t) = 1 + \max(\mathcal{P}(g(a)), \mathcal{P}(f(a, f(g(b), b)))) = 5$.
- $St(t) = \{t, g(a), a, f(a, f(g(b), b)), f(g(b), b), g(b), b\}$.

Nous remarquons que les arbres ainsi définis sont ordonnés. Par conséquent, cette définition peut être généralisée et des arbres non ordonnés peuvent être ainsi définis. Le couple $(A, m)$ avec $m : A \to \mathbb{N}$ est appelé *Multi-ensemble* où $A$ est un ensemble quelconque appelé *support* et $m$ une application de $A$ vers l'ensemble des entiers naturels. Un multi-ensemble est noté en utilisant les accolades doubles

$\{\!|\}$ où chaque élément est répété est répété autant de fois que son image par $m$. Grâce à la notion de multi-ensembles, nous pouvons définir les arbres non ordonnés.

**Définition 1.4.** *Soit $\Sigma$ un alphabet fini. L'ensemble des arbres non-ordonnés $M_\Sigma$ est le plus petit ensemble défini par :*

1. $(f, \{\!|\}) \in M_\Sigma$ *pour tout* $f \in \Sigma_0$
2. $\forall t \subseteq M_\Sigma, m, f \in \Sigma : (f, t) \in M_\Sigma$.

On peut définir une application entre les arbres ordonnés et non-ordonnés :

**Définition 1.5.** *Soit $\Sigma$ un alphabet quelconque. Nous définissons l'application $\alpha : T_\Sigma \to M_\Sigma$ qui relie chaque arbre ordonné à un arbre non-ordonné tel que pour un arbre $t \in T_\Sigma$ nous avons :$\forall f(t_1, \ldots, t_n) \in St(t), t_1, \ldots, t_n \in T_\Sigma : (f, \{\!|t_1, \ldots, t_n|\!\}) \in \alpha(t)$.*

**Exemple 1.4.** *Prenant l'arbre illustré par la Figure 1.3. Cet arbre est vu comme un arbre non-ordonné si nous supposons que la permutation de ses branches est permise. L'arbre s'écrit formellement comme suit :*

$$(f, \{\!|(g, \{\!|(a, \{\!|\}\!)|\}\!), (f, \{\!|(a, \{\!|\}\!), (f, \{\!|(g, \{\!|(b, \{\!|\}\!)|\}\!), (b, \{\!|\}\!)|\}\!)|\}\!)$$

### 1.1.3 Étiquetage d'arbres

L'étiquetage est l'opération qui consiste à associer à chaque nœud de l'arbre un identifiant unique. Ce dernier doit permettre de retrouver la position du nœud qui lui est associé dans l'arbre. Dans ce qui suit, nous introduisons un étiquetage hiérarchisé appelé *domaine d'arbres*.

Vu comme une généralisation de la notion des positions dans les mots, l'étiquetage de l'arbre n'est pas une application vers $\mathbb{N}$ car l'arborescence suit une hiérarchie bien connue contrairement aux mots qui ne sont qu'une suite de symboles vue comme un tableau qui peut être indexé par des entiers.

Pour étiqueter un arbre, nous définissons une construction hiérarchisée dans $\mathbb{N}$ appelée *Domaine d'arbres*.

**Définition 1.6.** *Un domaine d'arbres $D \subseteq \mathbb{N}^*$ satisfait :*

1. *Soit $u = u'v \in D$ tel que $v, u' \in \mathbb{N}^*$ alors $u' \in D$*
2. *Si $uj \in D$ tel que $j \in \mathbb{N}$ alors pour tout $i \in \{1 \ldots, n\}, ui \in D$*

*Il nécessaire d'utiliser la notation $(u, v)$ au lieu de $uv$ pour décrire la concaténation des entier. Ceci permet de distinguer entre la concaténation de $12$ avec $1$ et de $1$ avec $21$.*

**Exemple 1.5.**
1. *L'ensemble $D = \{0, 1, (0, 0), (0, 1)\}$ est un domaine d'arbre*
2. *L'ensemble $D' = \{0, (0, 0), (1, 1)\}$ n'est pas un domaine d'arbre ($(1, 1) \in D'$ mais $1 \notin D'$.*

Grâce aux domaines d'arbres, nous définissons l'étiquetage d'un arbre.

**Définition 1.7.** *Un arbre $t \in T_\Sigma$ est étiqueté par un domaine d'arbres $D$ en définissant une application $Pos^t : D \cup \{\varepsilon\} \to T_\Sigma$ tel que :*

**(i)** $Pos^t(\varepsilon) = t$,

**(ii)** *Si* $t' = f(s_1, \ldots, s_{i-1}, s, s_{i+1}, \ldots, s_{\hat{f}}) \in St(t)$ *et* $Pos^t(d) = t', d \in D$ *alors* $Pos^t(d(i-1)) = s$.

*Ensuite,* $D^t$ *est construit comme suit :*

$$D^t = \{(f, d) \mid \exists t' \in St(t) : Pos^t(d) = t' \text{ et } \mathcal{R}(t') = f\}$$

La fonction définie récursivement $Cons : D^t \to T_\Sigma$ reconstitue à partir d'un étiquetage $D^t$ l'arbre correspondant $t$ comme suit : $Cons(D^t) = f(Cons(D^{t_1}), \ldots, Cons(D^{t_{\hat{f}}}))$ tel que $(f, \varepsilon) \in D^t$) et $D^{t_i} = \{(g, j) \mid (g, ij) \in D^t\}$

Le rôle de cette fonction est exactement l'inverse du rôle de $Pos$.

Nous rappelons que $\varepsilon$ est l'élément neutre de la concaténation de mots.

D'après la définition 1.7 $Pos$ est une injection de $D$ dans $\Sigma$. La deuxième propriété exige que les positions d'un arbre ne puissent pas dépasser le rang maximal de l'alphabet.

**Exemple 1.6.** *Nous allons étiqueter, grâce à* $Pos$*, l'arbre* $t = f(g(a), f(a, f(g(b), b)))$

1. $Pos^t(\varepsilon) = t$,
2. $Pos^t(0) = g(a)$ *(*$0$ *est vu comme* $\varepsilon$*),*
3. $Pos^t(1) = f(a, f(g(b), b))$ *(*$Pos^t(2)$ *n'existe pas car* $\hat{f} = 2$*).*
4. *etc.*

*Enfin, nous aurons l'arbre étiqueté présenté par Figure 1.4.*

$D^t = \{(f, \varepsilon), (g, 0), (a, 0.0), (f, 1), (a, 1.0), (f, 1.1), (g, 1.1.0), (b, 1.1.0.0), (b, 1.1.1)\}$

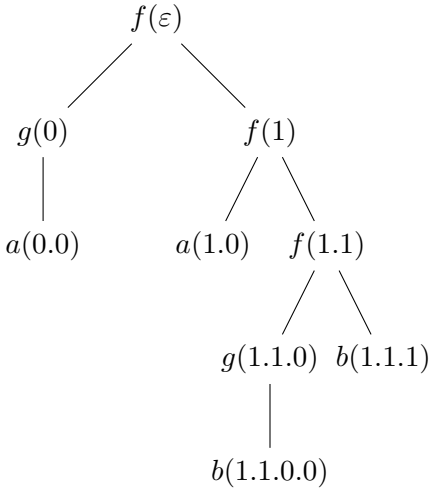



FIGURE 1.4: Exemple d'arbre étiqueté

**Théorème 1.** *Chaque arbre rangé possède un étiquetage unique.*

*Démonstration.* La preuve se fait par induction sur la construction de l'étiquetage. Soit $t \in T_\Sigma$ un arbre rangé. La position de $t$ est égale toujours à $\varepsilon$. Soit $d \in D$ tel que $Pos(d) = f(s_1, \ldots, s_{\hat{r}}) \in$

$St(t)$ alors par définition nous avons $Pos^t(di) = s_i, i = 1 \ldots, \hat{r}$. Comme $d$ est unique (hypothèse d'induction) alors la position des racines des sous arbres $s_i, i = 1 \ldots, \hat{r}$ sont uniques. □

La notion d'étiquetage et son unicité permettent de représenter la topologie d'un arbre donné d'une façon unique. Cet étiquetage peut être utilisé dans plusieurs traitements comme l'indexation des arbres, l'isomorphisme d'arbres, le parcours, etc.

**Étiquetage des sous arbres** L'étiquetage d'un sous arbre peut être directement déduit à partir de l'arbre original en supprimant simplement la position de la racine du sous arbres des positions de ses nœuds. Soit $t \in T_\Sigma$ un arbre rangé et $s \in St(t)$ alors l'étiquetage $D^s$ de $s$ se calcule via $Pos^s$ comme suit :

- $Pos^s(\varepsilon) = s$
- $Pos^s(i) = Pos^t(di)$ tel que $Pos^t(d) = s$.

## 1.2 Opérations sur les arbres

Il existe plusieurs opérations utiles pour les arbres. Deux parmi elles ont déjà été citées. Il s'agit de l'opération $\mathcal{R}$ qui retourne le symbole de la racine d'un arbre et $St$ qui retourne l'ensemble des sous arbres. Dans cette section, nous allons présenter d'autres opérations très intéressantes : substitution, concaténation, homomorphisme et isomorphisme.

### 1.2.1 La substitution

La substitution peut être vue comme un remplacement d'un sous arbre d'un arbre cible par un autre arbre donné. Selon [Cle08], il en existe trois types de substitutions.

**Définition 1.8.** *(Substitution d'une seule occurrence d'un sous arbre) Soit $s, t \in T_\Sigma$. $t[\overset{d}{\leftarrow} s]$ dénote l'arbre dans lequel l'arbre $u = Pos^t(d)$ est remplacé par $s$.*

**Définition 1.9.** *(Substitution de toutes les occurrences d'un sous arbres) Soient $s, t \in T_\Sigma$. $t[u \leftarrow s]$ dénote l'arbre dans lequel toutes les occurrences de l'arbre $u$ sont remplacées par l'arbre $s$. Cette substitution est définie comme suit :*

- $u[u \leftarrow s] = s$,
- $t[u \leftarrow s] = t$ *si* $s \notin St(t)$,
- $f(t_1, \ldots, t_{\hat{f}})[u \leftarrow s] = f(t_1[u \leftarrow s], \ldots, t_{\hat{f}}[u \leftarrow s])$.

#### 1.2.1.1 Substitution élémentaire

Dans le contexte de notre étude, nous définissons une substitution spéciale (notée $\Leftarrow$) :

**Définition 1.10.** *Un arbre $u \in t(r \Leftarrow s)$ est obtenu en remplaçant une seule occurrence du symbole $r$ par l'arbre $s$.*

**Example 1.1.** *Soit* $t = f(a, g(b, a))$ *un arbre.* $t(a \leftarrow g(b)) = f(g(b), g(b, g(b)))$ *mais* $t(a \Leftarrow g(b)) = \{f(g(b), g(b, a)), f(a, g(b, g(b)))\}$. *Comme le montre cet exemple, cette substitution produit un ensemble.*

#### 1.2.1.2 Concaténation d'arbres

Étant un cas spécial de la substitution, la concaténation d'arbres est une opération très utilisée car elle participe dans la définition d'une classe de régularité pour les arbres [Eng75]. Il s'agit d'une substitution ou l'arbre à substituer est une constante.

**Définition 1.11.** *(Concaténation d'arbres) Soit* $s, t \in T_\Sigma$. $t._c s$ *dénote l'arbre où toutes les occurrences de* $c \in \Sigma_0$ *sont remplacées par l'arbre* $s$.

- $c._c s = s$,
- $d._c s = d$ *si* $d \neq c$,
- $f(t_1, \ldots, t_{\hat{f}})._c s = f(t_1._c s, \ldots, t_{\hat{f}}._c s)$.

La concaténation est une loi de composition interne dans $T_\Sigma$ satisfaisant :

- Elle n'est pas commutative : $t._c u \neq u._c t$,
- Elle est associative : $s._c(t._c u) = (s._c t)._c u$,
- L'élément neutre de $._c$ est $c$.

### 1.2.2 L'homomorphisme

L'homomorphisme est une application entre deux ensembles qui permet de respecter une structure bien définie. Concernant les arbres, l'homomorphisme permet de comparer la structure de deux arbres et tester une éventuelle équivalence entre eux. Un exemple type de l'homomorphisme est la transformation d'un arbre n-aire en un arbre binaire.

**Définition 1.12.** *Soit* $T_\Sigma, T_{\Sigma'}$ *les ensembles des arbres rangés définis respectivement sur* $\Sigma$ *et* $\Sigma'$ *et* $X$ *un ensemble de constantes (feuilles) disjoint de* $\Sigma$ *et* $\Sigma'$. *L'homomorphisme d'arbres* $\phi$ *est une application* $\phi : T_\Sigma \to T_{\Sigma'}$ *définie comme suit.*

- $\phi(f) = t_f$ *si* $f \in \Sigma_0$ *(avec* $t_f \in \Sigma'$*),*
- $\phi(f(t_1, \ldots, t_{\hat{f}})) = t_f[x_1 \leftarrow \phi(t_1)] \ldots [x_{\hat{f}}] \leftarrow \phi(t_{\hat{f}})]$ *sinon (avec* $t_f \in T_{\Sigma' \cup X}$*)* .

Nous remarquons que $t_f$ n'est pas dans $T_{\Sigma'}$ mais ceci ne fausse pas la définition car $t_f[x_1 \leftarrow \phi(t_1)] \ldots [x_{\hat{f}}] \leftarrow \phi(t_{\hat{f}})] \in T_{\Sigma'}$.

**Exemple 1.7.** *Soit* $\Sigma = \{f(,,), a, b\}$ *et* $\Sigma' = \{g(,), a, b\}$. *Nous définissons l'homomorphisme suivant (voir Figure 1.5).*

- $\phi(f(t_1, t_2, t_3) = g(\phi(t_1), g(\phi(t_2), \phi(t_3)))$.
- $\phi(a) = a, \phi(b) = b$.

*Soit* $t = f(a, f(b, b, b), a) \in T_\Sigma$.

$$\begin{aligned}\phi(t) &= g(\phi(a), g(\phi(f(b,b,b)), \phi(a))) \\ &= g(a, g(g(\phi(b), g(\phi(b), \phi(b)))), a) \\ &= g(a, g(g(b, g(b, b)), a))\end{aligned}$$

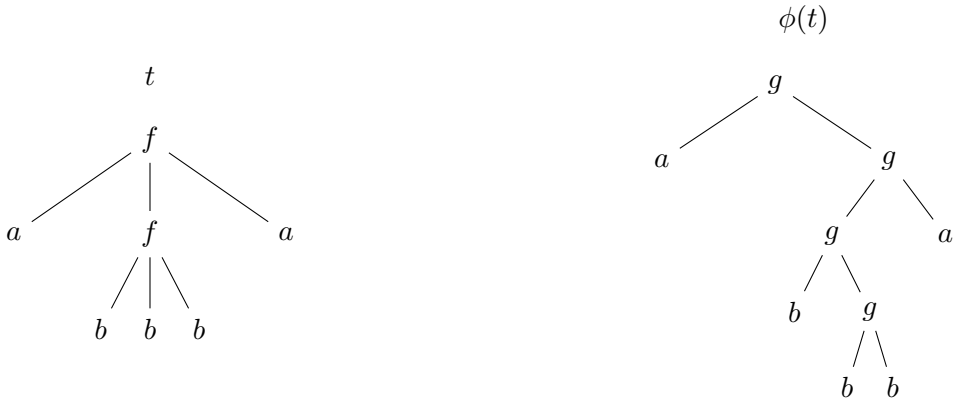



FIGURE 1.5: Exemple de l'homomorphisme d'arbres

### 1.2.3 L'isomorphisme

L'isomorphisme consiste en une application qui permet de préserver la topologie de l'arbre. Autrement dit si un nœud est en relation avec un autre nœud (père, fils à la position $i$) alors leurs images le sont aussi. Néanmoins, l'isomorphisme d'arbres est étudié comme un sous problème d'isomorphisme de graphes où un arbre est vu comme un graphe connexe et sans cycles [Ull76]. En effet, la notion de rang n'existe pas dans ce contexte.

Ici, nous définissons l'isomorphisme d'arbre respectant le rang des nœuds comme suit.

**Définition 1.13.** *Soit $T_\Sigma, T_{\Sigma'}$ les ensembles des arbres rangés définis respectivement sur $\Sigma$ et $\Sigma'$. L'isomorphisme d'arbres $\overline{\phi}$ est une application $\overline{\phi} : T_\Sigma \to T'_\Sigma$ définie comme suit.*

- $\overline{\phi}(f) = f'$ *si* $f \in \Sigma_0$ *avec* $f' \in \Sigma'_0$,
- $\overline{\phi}(f(t_1, \ldots, t_{\hat{f}})) = f'(\overline{\phi}(t_1), \ldots \overline{\phi}(t_{\hat{f}})$ *tel que* $f' \in \Sigma'_{\hat{f}}$.

Une remarque très importante présentée dans le théorème suivant est que deux arbres sont isomorphes si et seulement si l'un peut être obtenu en renommant les nœuds de l'autre. En d'autres termes si et seulement si leurs étiquetage est le même

**Théorème 2.** *Soit $t \in T_\Sigma, t' \in T_{\Sigma'}$ deux arbres rangés. $t$ et $t'$ sont isomorphes par $\overline{\phi}$ si et seulement si $(f, d) \in D^t \Leftrightarrow (\overline{\phi}(f), d) \in D^{t'}$.*

## 1.3 Langages d'arbres

La notion de langages d'arbres est très utile dans le traitement des grandes masses d'informations car elle donne la possibilité de décrire/reconnaître des familles d'arbres qui ont des caractéristiques spécifiques. Dans cette section, nous nous intéressons aux fondements formels des langages d'arbres.

**Définition 1.14.** *Soit $T_\Sigma$ l'ensemble des arbres rangés définis sur $\Sigma$. Un langage d'arbres $L$ est un sous-ensemble de $T_\Sigma$. ($L \subset T_\Sigma$).*

Vue que les langages d'arbres sont des ensembles, on peut utiliser les opérations usuelles tel que l'union de deux langages d'arbres $K$ et $L$ ($K \cup L$), l'intersection ($K \cap L$), le complément $\complement(L)\ldots$

En raison de simplicité, nous définissons quelques notations nécessaires pour la lecture du reste du mémoire.

### 1.3.1 Opérations sur les langages d'arbres

Les mêmes opérations de substitution sur les arbres peuvent être étendues sur les langages d'arbres

**Définition 1.15.** *Soit, $f \in \Sigma_n, L, L_1, \ldots, L_n \in T_\Sigma$.*

- $f(L_1, \ldots, L_n) = \{f(t_1, \ldots, t_n) \mid t_i \in L_i, i = 1, n\}$ *(forêt de langages),*
- $L[\overset{d}{\leftarrow} s] = \{t[\overset{d}{\leftarrow} s] \mid t \in L\}$*(Substitution d'une seule occurrence d'un sous arbre dans un langage),*
- $L[u \leftarrow s] = \{t[u \leftarrow s] \mid t \in L\}$ *(substitution d'un arbre dans un langage),*
- $L_1 ._c L_2 = \{t_1 ._c t_2 \mid t_1 \in L_1, t_2 \in L_2\}$ *(concaténation de deux langages ou produit-c).*

Le produit-$c$ est aussi connu sous le nom de *out-inside* (OI) substitution [ES77]. Nous rappelons quelques propriétés algébriques liées à cette notion (voir [ES77, Esi10, BN98]).

Comme dans le cas des mots, le produit-$c$ est distributif sur l'union :

**Lemme 1.1.** *Soient $L_1$, $L_2$ et $L_3$ trois langages d'arbres à travers $\Sigma$. Soit $c$ un symbole dans $\Sigma_0$. Alors :*

$$(L_1 \cup L_2) \cdot_c L_3 = (L_1 \cdot_c L_3) \cup (L_2 \cdot_c L_3)$$

*Démonstration.* Soit $t$ un arbre dans $T_\Sigma$. Alors :

$$\begin{aligned} t \in (L_1 \cup L_2) \cdot_c L_3 &\Leftrightarrow \exists u \in L_1 \cup L_2, \exists v \in L_3, t = u \cdot_c v \\ &\Leftrightarrow (\exists u \in L_1, \exists v \in L_3, t = u \cdot_c v) \vee (\exists u \in L_2, \exists v \in L_3, t = u \cdot_c v) \\ &\Leftrightarrow t \in (L_1 \cdot_c L_3) \cup (L_2 \cdot_c L_3) \end{aligned}$$

□

Une autre propriété commune est l'associativité.

**Lemme 1.2.** *Soient $t$ et $t'$ deux arbres dans $T_\Sigma$. Soit $L$ un langage d'arbres à travers $\Sigma$ et soit $c$ un symbole dans $\Sigma_0$. Alors :*

$$t \cdot_c (t' \cdot_c L) = (t \cdot_c t') \cdot_c L$$

*Démonstration.* Par induction sur la structure de $t$.

1. $t = c$. Alors $t \cdot_c (t' \cdot_c L) = t' \cdot_c L = (t \cdot_c t') \cdot_c L$.
2. $t \in \Sigma_0 \setminus \{c\}$. Alors $t \cdot_c (t' \cdot_c L) = t = (t \cdot_c t') \cdot_c L$.
3. $t = f(t_1, \ldots, t_n)$ avec $n > 0$. Alors :

$$\begin{aligned} f(t_1, \ldots, t_n) \cdot_c (t' \cdot_c L) &= f(t_1 \cdot_c (t' \cdot_c L), \ldots, t_n \cdot_c (t' \cdot_c L)) \\ &= f((t_1 \cdot_c t') \cdot_c L, \ldots, (t_n \cdot_c t') \cdot_c L) \quad \text{(Hypothèse d'induction)} \\ &= f(t_1 \cdot_c t', \ldots, t_n \cdot_c t') \cdot_c L \\ &= (f(t_1, \ldots, t_n) \cdot_c t') \cdot_c L \end{aligned}$$

□

**Corollaire 1.1.** *Soient $L$, $L'$ et $L''$ trois langages à travers $\Sigma$, $c \in \Sigma_0$. Alors :*

$$L \cdot_c (L' \cdot_c L'') = (L \cdot_c L') \cdot_c L''$$

Ainsi, nous pouvons conclure que le produit-$c$ est compatible avec l'inclusion.

**Lemme 1.3.** *Soit $t$ un arbre à travers $\Sigma$, et $L, L'$ deux langages de $T_\Sigma$ tel que $L \subset L'$. Alors :*

$$t \cdot_c L \subset t \cdot_c L'$$

Par conséquent :

**Corollaire 1.2.** *Soient $L$, $L' \subset L''$ trois langages $T_\Sigma$ et $c \in \Sigma_0$. Alors :*

$$L \cdot_c L' \subset L \cdot_c L''$$

Une nouvelle propriété qui diffère du cas des mots est présentée dans le lemme suivant.

**Lemme 1.4.** *Soient $t_1$, $t_2$ et $t_3$ trois arbres de $T_\Sigma$. Soient $a$ et $b$ deux symboles distincts dans $\Sigma_0$ tel que $a$ n'apparaît pas dans $t_3$. Alors :*

$$(t_1 \cdot_a t_2) \cdot_b t_3 = (t_1 \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3)$$

*Démonstration.* Par induction sur $t_1$.

1. Si $t_1 = a$, alors

$$(t_1 \cdot_a t_2) \cdot_b t_3 = t_2 \cdot_b t_3 = (t_1 \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3)$$

2. Si $t_1 = b$, alors

$$(t_1 \cdot_a t_2) \cdot_b t_3 = t_3 = (t_1 \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3)$$

3. Si $t_1 = c \in \Sigma_0 \setminus \{a, b\}$, alors

$$(t_1 \cdot_a t_2) \cdot_b t_3 = t_1 = (t_1 \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3)$$

4. Si $t_1 = f(u_1, \ldots, u_n)$ avec $n > 0$, Alors

$$\begin{aligned}(t_1 \cdot_a t_2) \cdot_b t_3 &= (f(u_1 \cdot_a t_2, \ldots, u_n \cdot_a t_2)) \cdot_b t_3 \\ &= f((u_1 \cdot_a t_2) \cdot_b t_3, \ldots, (u_n \cdot_a t_2) \cdot_b t_3) \\ &= f((u_1 \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3), \ldots, (u_n \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3)) \quad \text{(Hypothése d'induction)} \\ &= f(u_1 \cdot_b t_3, \ldots, u_n \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3) \\ &= (f(u_1, \ldots, u_n) \cdot_b t_3) \cdot_a (t_2 \cdot_b t_3)\end{aligned}$$

□

### 1.3.2 La fermeture de Kleene

L'une des notions les plus importantes dans la théorie des langages formels est la fermeture (étoile) de Kleene [HU79]. Cette notion a été définie sur les langages formels de mots pour la première fois par Kleene [Kle56] et puis généralisée aux arbres [GS97].

Avant de définir la fermeture de Kleene, nous présentons la notion d'exponentiation des langages d'arbres.

**Définition 1.16.** *L'exponentiation d'un langage d'arbres $L$ est une séquence d'itérations successives du produit $\cdot_c$ définies comme suit.*

$$L^{0,c} = \{c\} \tag{1.1}$$
$$L^{n+1,c} = L^{n,c} \cup L \cdot_c L^{n,c} \tag{1.2}$$

Nous définissons la fermeture de Kleene comme suit :

**Définition 1.17.**

$$L^{*c} = \bigcup_{i=0}^{\infty} L^{i,c} \tag{1.3}$$
$$L^{+c} = L^{*c} - \{c\} \tag{1.4}$$

**Exemple 1.8.** *Soit $L = \{f(a, b)\}$ alors*

- $L^{0,a} = \{a\}$,
- $L^{1,a} = \{a\} \cup \{f(a, b)\} \cdot_a \{a\} = \{f(a, b), a\}$,
- $L^{2,a} = \{f(f(a, b), b), f(a, b), a\}$.

- ...

*Nous aurons* $L^{*a} = \{f(f(a,b)), f(a,b), a, \ldots\}$.

Nous remarquons que contrairement au cas de mots, les produits peuvent commuter avec la fermeture dans certains cas :

**Lemme 1.5.** *Soient* $L_1$ *et* $L_2$ *deux langages dans* $T_\Sigma$. *Soient* $a$ *et* $b$ *deux symboles distincts dans* $\Sigma_0$ *tel que* $L_2 \subset T_{\Sigma\setminus\{a\}}$. *Alors :*

$$L_1^{*_a} \cdot_b L_2 = (L_1 \cdot_b L_2)^{*_a}$$

*Démonstration.* Nous prouvons par récurrence que pour un entier $n$, $L_1^{n,a} \cdot_b L_2 = (L_1 \cdot_b L_2)^{n,a}$.

1. Si $n = 0$, alors, en se référant à l'Équation (1.1)) :

$$L_1^{0,a} \cdot_b L_2 = \{a\} = (L_1 \cdot_b L_2)^{0,a}$$

2. Si $n > 0$, Alors, suivant l'Équation (1.2)) :

$$\begin{aligned}
L_1^{n+1,a} \cdot_b L_2 &= (L_1^{n,a} \cdot_a L_1 \cup L_1^{n,a}) \cdot_b L_2 \\
&= (L_1^{n,a} \cdot_a L_1) \cdot_b L_2 \cup (L_1^{n,a}) \cdot_b L_2 && \text{(Lemme 1.1)} \\
&= ((L_1^{n,a} \cdot_b L_2) \cdot_a (L_1 \cdot_b L_2)) \cup (L_1^{n,a}) \cdot_b L_2 \\
&= ((L_1 \cdot_b L_2)^{n,a} \cdot_a (L_1 \cdot_b L_2)) \cup (L_1 \cdot_b L_2)^{n,a} && \text{(Hypothese d'induction)} \\
&= (L_1 \cdot_b L_2)^{n+1,a}
\end{aligned}$$

Par conséquent, $L_1^{*_a} \cdot_b L_2 = (L_1 \cdot_b L_2)^{*_a}$.

□

## 1.4 Les expressions rationnelles d'arbres

Une autre façon de représenter les langages d'arbres est d'utiliser *les expressions régulières* ou *rationnelles* (ER). En effet, ce type d'expressions est utilisé, comme dans le cas des mots, pour *décrire* une famille de langages dite rationnelle. Dans ce qui suit, nous rappelons les expressions rationnelles sur les mots.

Une expression rationnelle (ER) est une suite de caractères qui ont pour rôle de dénoter ou décrire un langage régulier fini ou infini. On peut définir par induction une ER sur un alphabet $\Sigma$ comme suit :

**Définition 1.18.**

- $0$ *est une ER qui dénote le langage vide* $\emptyset$,
- $1$ *est une ER qui dénote le mot vide* $\{\varepsilon\}$,
- $a \in \Sigma$ *est une ER qui dénote le langage* $\{a\}$ ,
- *Si* $E$ *et* $F$ *sont deux ER alors* $E + F$ *et* $E.F$ *sont des ER qui dénotent respectivement les langages* $L_E \cup L_F$ *et* $L_E L_F$ ($L_E$ *et* $L_F$ *sont les langages qui dénotent les ER* $E$ *et* $F$ *respectivement),*

- *Si $E$ est une ER alors $E^*$ l'est aussi.*

Nous définissons en premier lieu l'ensemble des langages rationnels d'arbres définis sur un alphabet donné.

**Définition 1.19.** *(langage rationnel).* $\mathrm{Rat}(\Sigma)$ *est le plus petit ensemble de parties de $T_\Sigma$ qui contient les parties finies de $T_\Sigma$ et qui est fermé pour l'union, la concaténation et la fermeture de Kleene.*

**Définition 1.20.** *Une* Expression Rationnelle d'Arbres (ERA) *$E$ à travers un alphabet $\Sigma$ est un terme défini par induction comme suit :*

$$E = 0 \tag{1.5}$$
$$E = f(E_1, \ldots, E_n) \tag{1.6}$$
$$E = E_1 + E_2 \tag{1.7}$$
$$E = E_1 \cdot_c E_2 \tag{1.8}$$
$$E = E_1^{*_c} \tag{1.9}$$

*avec $f \in \Sigma_n$, $c \in \Sigma_0$ et $E_i$ est une ERA pour $i = 1 \ldots n$.*

Afin d'extraire la sémantique d'une ERA (le langage dénoté par une ERA) nous introduisons la définition suivante :

**Définition 1.21.** *La* sémantique *d'une ERA $E$ est le langage d'arbres $[\![E]\!]$ défini par induction comme suit :*

$$[\![0]\!] = \emptyset \tag{1.10}$$
$$[\![a]\!] = \{a\} \tag{1.11}$$
$$[\![E_1 + E_2]\!] = [\![E_1]\!] \cup [\![E_2]\!] \tag{1.12}$$
$$[\![E_1 \cdot_c E_2]\!] = [\![E_1]\!] \cdot_c [\![E_2]\!] \tag{1.13}$$
$$[\![E^{*_c}]\!] = [\![E]\!]^{*_c} \tag{1.14}$$
$$[\![f(E_1, \ldots, E_n)]\!] = \{f(s_1, \ldots, s_n) \mid s_1 \in [\![E_1]\!], \ldots, s_n \in [\![E_n]\!]\} \tag{1.15}$$

*où $f \in \Sigma_n$, $c \in \Sigma_0$ et $E_i$ est une ERA pour $i = 1 \ldots n$.*

On dit que deux ERA $E$ et $F$ sont *équivalentes* ($E \equiv F$) si et seulement si $[\![E]\!] = [\![F]\!]$.

Quelques propriétés usuelles sont résumées dans le lemme suivant.

**Lemme 1.6.** *Soient $E_1, E_2, E_3$ Trois ERA à travers $\Sigma$. Soit $c \in \Sigma_0$. Alors nous avons :*

$$E_1 + E_1 \equiv E_1 \tag{1.16}$$
$$E_1 + 0 \equiv E_1 \textbf{ (Element neutre } +\textbf{)} \tag{1.17}$$
$$E_1 + E_2 \equiv E_2 + E_1 \textbf{ (Commutativité de } +\textbf{)} \tag{1.18}$$
$$(E_1 + E_2).{}_cE_3 \equiv E_1.{}_cE_3 + E_2.{}_cE_3 \textbf{ (distributivité à droite)} \tag{1.19}$$
$$E_1.{}_cE_1^{*c} \equiv E_1^{*c}.{}_cE_1 \equiv E_1^{*c} \tag{1.20}$$
$$E_1^{*c} + E_1 \equiv E_1^{*c} \tag{1.21}$$
$$\textit{si } c \notin [\![E_1]\!] \textit{ alors } E_1.{}_cE_2 \equiv E_1 \tag{1.22}$$

Chaque propriété sous la forme $F \equiv G$ pour deux ERA $F$ et $G$ peut être facilement démontrée en prouvant que $[\![F]\!] = [\![G]\!]$.

## 1.5 Arbres et multiplicités

Il est très utile d'ajouter des *Multiplicités* (poids) aux arbres. Les séries d'arbres sont le formalisme le plus général pour représenter cette notion. Premièrement, nous rappelons la notion de semi-anneau [Mal05] :

Soit $\mathbb{K}$ un ensemble quelconque. Un semi anneau est une structure $\Phi = (\mathbb{K}, \oplus, \otimes, 0, 1)$ tel que :

- $(\mathbb{K}, \oplus, 0)$ est un monoïde commutatif
- $(\mathbb{K}, \otimes, 1)$ est un monoïde
- $\otimes$ est distributive sur $\oplus$ :
  $\forall i, j, k \in \mathbb{K}, i \otimes (j \oplus k) = (i \otimes j) \oplus (i \otimes k)$ et $(i \oplus j) \otimes k = (i \otimes k) \oplus (j \otimes k)$
- $0$ est un élément absorbant pour $\otimes$ : $\forall k \in \mathbb{K}, k \otimes 0 = 0 \otimes k = 0$

Le semi anneau est dit commutatif si le monoïde $(\mathbb{K}, \otimes, 1)$ est commutatif.

On utilise souvent $\sum_{i \in \mathbb{N}} a_i$ (resp. $\Pi_{i \in \mathbb{N}} a_i$ ) pour dénoter la somme (resp. le produit) des familles d'objets $(a_i)_{i \in \mathbb{N}} a_i$ où $a_i \in \mathbb{K}, i \in \mathbb{N}$.

### 1.5.1 Séries d'arbres

Soit $S$ un ensemble quelconque. Une *série formelle* est une application $\varphi : S \to \mathbb{K}$. Une série formelle peut s'écrire comme suit

$$\sum_{s \in S} \varphi(s) \otimes s$$

Le *support* de la série formelle $\varphi$ est : $supp(\varphi) = \{s \in S \mid \varphi(s) \neq 0\}$. Les séries avec un support non-nul sont aussi appelées *polynômes*.

En effet, les *séries d'arbres* sont considérées comme des séries formelles opérant sur des arbres :

**Définition 1.22.** *Soit $\Phi = (\mathbb{K}, \oplus, \otimes, 0, 1)$ un semi anneau défini sur un ensemble quelconque $\mathbb{K}$. Soit $T_\Sigma$ l'ensemble des arbres à travers un alphabet $\Sigma$. Une série d'arbres est une série formelle $\varphi : T \to \mathbb{K}$ où $T \subseteq T_\Sigma$. L'ensemble de toutes les séries d'arbres est noté $\mathbb{K}\langle\langle T \rangle\rangle$.*

### 1.5.2 Expression rationnelles avec multiplicité

Nous pouvons définir la famille des langages rationnels à multiplicité pour une série d'arbres $\mathbb{K}\langle\langle T \rangle\rangle$. En effet, $\mathrm{Rat}(\Sigma, \mathbb{K})$ est le plus petit ensemble de parties de $\mathbb{K}\langle\langle T \rangle\rangle$ qui contient les parties finies de $\mathbb{K}\langle\langle T \rangle\rangle$ et qui est fermé pour l'union, la concaténation et la fermeture de Kleene.

**Définition 1.23.** *Soit $\Sigma$ un alphabet. Soit $\Phi = (\mathbb{K}, \oplus, \otimes, 0, 1)$ un semi anneau. L'ensemble des expressions rationnelles de séries d'arbres (ERSA) à travers $(\Sigma, \mathbb{K})$ est défini par induction comme suit.*

$$E = 0 \tag{1.23}$$
$$E = f(E_1, \ldots, E_n) \tag{1.24}$$
$$E = E_1 + E_2 \tag{1.25}$$
$$E = E_1 \cdot_c E_2 \tag{1.26}$$
$$E = E_1^{*c} \tag{1.27}$$
$$E = kE_1 \tag{1.28}$$

*avec $f \in \Sigma_n$, $c \in \Sigma_0$, $k \in \mathbb{K}$ et $E_i$ est une ERA pour $i = 1 \ldots n$.*

Il est clair que la sémantique d'une ERSA donne des série d'arbres comme langages de sortie, mais avant de définir la sémantique, nous définissons une fonction élémentaire $top_{f \in \Sigma}$.

**Définition 1.24.** *Soit $\Sigma$ un alphabet quelconque. Soit $\Phi = (\mathbb{K}, \oplus, \otimes, 0, 1)$ un semi anneau. La* top-concaténation *avec $f \in \Sigma$ est une application $top_f : \mathbb{K}\langle\langle T_\Sigma \rangle\rangle^{\hat{f}} \to \mathbb{K}\langle\langle T_\Sigma \rangle\rangle$ tel que pour chaque $\varphi_1, \ldots, \varphi_{\hat{f}} \in \mathbb{K}\langle\langle T_\Sigma \rangle\rangle$ et $t \in T_\Sigma$, si $t = f(s_1, \ldots, s_{\hat{f}}), s_1, \ldots, s_{\hat{f}} \in T_\Sigma$ alors :*

$$top_f(\varphi_1(t), \ldots, \varphi_{\hat{f}}(t)) = \varphi_1(s_1) \otimes \ldots \otimes \varphi_{\hat{f}} s_{\hat{f}}$$

La sémantique d'une ERSA est définie par induction comme suit :

**Définition 1.25.** *avec $f \in \Sigma, c \in \Sigma_0, n = \hat{f}$ et $k \in \mathbb{K}$.*

## 1.6 Conclusion

Nous avons résumé dans ce chapitre les différentes notions et notations formelles liées à notre étude. Premièrement, nous avons présenté les définitions formelles d'un arbre, langage d'arbres et les différentes opérations qui leurs sont associées. L'ensemble de ces concepts est utile tout au long du mémoire. La définition des séries d'arbres sert à ajouter la dimension multiplicité aux langages, automates ainsi qu'aux expressions rationnelles d'arbres. La partie liée aux ERA sera utilisée dans Chapitre 5. Dans le prochain chapitre, nous définissons les différents types d'automates d'arbres et

nous nous focalisons sur un type particulier : les automates d'arbres rangés déterministes ascendants -objets de notre étude- et nous introduisons les différents formalismes et algorithmes liées.

# Chapitre 2

# Automates d'arbres rangés

Comme dans le cas des mots, un automate d'arbres est un type de *Machine à états* qui effectue la lecture des arbres au lieu de mots. Bien qu'ils soient considérés comme une généralisation des automates de mots, plusieurs différences existent. A titre d'exemple, la notion de successeur dans les mots est remplacée par la relation père-fils. Ainsi dans les automates d'arbres rangés, on ajoute la position du fils dans l'arborescence. Un autre exemple est l'absence de la notion de *miroir* dans les arbres.

Par conséquent, on peut tirer deux types d'automates d'arbres selon le sens de lecture : les automates d'arbres ascendants, et les automates d'arbres descendants.

Dans ce chapitre, nous introduisons les automates d'arbres à états finis qui sont des représentations compactes pour reconnaître des ensembles d'arbres pouvant être infini.

Avant la présentation des automates d'arbres descendants, nous nous intéressons à la version ascendante. Puisque nous allons montrer que la version descendante est moins performante que son homologue ascendant [CDG+07].

## 2.1 Automates d'arbres rangés à états finis ascendants

Le type ascendant pour les automates d'arbres rangés à états finis est le type plus utilisé dans la littérature. En effet, le processus de lecture (bas vers le haut) est plus naturel. De plus, la classe de langages reconnus par les automates ascendants est bien définie (régularité) [CDG+07].

**Définition 2.1.** *Un automate d'arbres rangés à états finis ascendant AAFA à travers un alphabet rangé $\Sigma$ est le quadruplet $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ où :*

1. *$Q$ est un ensemble de symboles appelés* états*,*
2. *$Q_f \subseteq Q$ est un ensemble d'états finaux,*
3. *L'ensemble finis de transitions :$\Delta \subseteq \bigcup_{n\geq 0} \Sigma_n \times Q^{n+1},\ n \in \mathbb{N}$.*

Un AAFA n'a pas d'états initiaux. En effet, le calcul des arbres se fait à partir de ses feuilles.

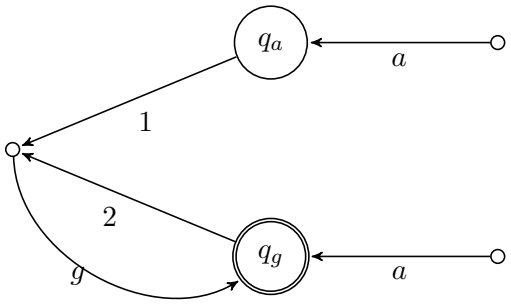



FIGURE 2.1: Exemple d'AAFA

**Définition 2.2.** *(Taille de l'automate) Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA. La taille d'une transition $\rho = (f, q_1, \ldots, q_n), f \in \Sigma,\ q_1, \ldots, q_n \in Q$ est $|\rho| = n + 1$. On peut définir la taille d'un automate à partir des tailles de ses transitions.*

$$|\mathcal{A}| = \sum_{\rho \in \Delta} |\rho|$$

*La taille d'un automate peut être majorée par :*

$$|\mathcal{A}| < (\hat{r} + 1) \times |\Delta|$$

*où $\hat{r}$ est le rang maximum de l'alphabet $\Sigma$.*

## Représentation graphique

Un automate est représenté comme un graphe orienté étiqueté. Les états sont des ronds à l'intérieur desquels on porte le nom de l'état. Une transition $(f, q_1, \ldots, q_n, q)$ est représentée par des flèches qui partent de chaque état de départ $q_i$ et pointe sur un nœud spécial étiqueté $f$. Chaque flèche est étiquetée par la position $i$ de l'état dans la transition. Une autre flèche part du nœud $f$ et pointe sur l'état d'arrivée $q$. Un état final est marqué par un double rond.

**Exemple 2.1.** *Soit l'automate d'arbres suivant.*

$\mathcal{A} = (\{q_a, q_g\}, \{a, g(,)\}, \{q_g\}, \{(a, q_a), (a, q_g), (g, q_a, q_g, q_g)\})$. *La taille de cet automate est :*

$$|\mathcal{A}| = |(a, q_a)| + |(a, q_g)| + |(g, q_a, q_g)| = 2 + 2 + 3 = 7$$

*La Figure 2.1 illustre la représentation graphique de l'automate $\mathcal{A}$.*

**Définition 2.3.** *(*Automate complet*) On dit qu'un AAFA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ est complet si et seulement si :*

$$\forall f \in \Sigma_n, 1 \leq n \leq \hat{r}, q_1, \ldots, q_n \in Q : \exists q \in Q : (f, q_1, \ldots, q_n, q) \in \Delta) \tag{2.1}$$

On peut définir des opérations sur les transitions (ou bien tout élément de $\Sigma \times Q^n$ avec $n \in \mathbb{N}$) :

**Définition 2.4.** *(Arguments, occurrences d'un état)*

- $\Gamma(q) = \{(f, q_1, \ldots, q, \ldots, q_n) \mid (f, q_1, \ldots, q, \ldots, q_n, q') \in \Delta\}$ *dénote l'ensemble des arguments extraits depuis les transitions dans lesquels* $q$ *apparaît mais pas en dernière position.*
- $occ_q((f, q_1, \ldots, q_n)) = \{i \mid q_i = q\}$ *dénote l'ensemble des positions d'un état* $q$ *dans l'argument* $(f, q_1, \ldots, q_n)$.

La première opération sur les AAF est la reconnaissance d'un ou de plusieurs arbres. En effet, un automate d'arbre est conçu pour lire des arbres et donc décider s'ils sont reconnaissables ou pas.

**Définition 2.5.** *La fonction de sortie* $\delta : T_\Sigma \to 2^Q$ *d'un AAF* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *est définie pour un arbre* $t = f(t_1, \ldots, t_n)$ *comme :*

$$\delta(t) = \{q \mid \exists (f, q_1, \ldots, q_n, q) \in \Delta \text{ et } q_i \in \delta(t_i) \text{ pour } i = 1, \ldots, n\} \tag{2.2}$$

La fonction de sortie peut être utilisée sur des arbres de $T_{\Sigma \cup \{Q\}}$ (en supposant que les états sont des constantes). Évidemment, si $t = f(q_1, \ldots, q_n) \in T_{\Sigma \cup \{Q\}}$ alors $\delta(t) = q$ s'il existe une transition $(f, q_1, \ldots, q_n, q) \in \Delta$.

On dit qu'un arbre $t$ est reconnu par un automate $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ si et seulement si

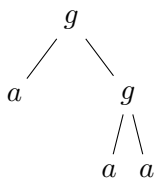
$$\delta(t) \cap Q_f \neq \emptyset$$

Par conséquent, nous définissons le langage accepté (reconnu) par un AAFA :

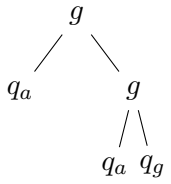
**Définition 2.6.** *Le langage accepté par un AAFA* $\mathcal{A}$ *noté* $L(\mathcal{A})$ *est l'ensemble de tous les arbres acceptés par ce dernier.*

Pour l'Exemple 2.1, l'ensemble d'arguments où $q_g$ figure est $\Gamma(q_g) = \{(g, q_g)\}$.
Pour l'argument $(g, q_g)$, $occ_{q_g}((g, q_g)) = \{1\}$

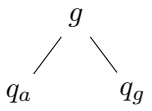
L'arbre $t = g(a, g(a, a))$ est reconnu par l'automate $\mathcal{A}$ (voir les étapes dans Figure 2.2), mais l'arbre $t' = g(g(a, a), g(a, a))$ ne l'est pas (voir les étapes dans Figure 2.3 ).




(1) l'arbre $t$




(2) on utilise $a \to q_a$ et $a \to q_g$

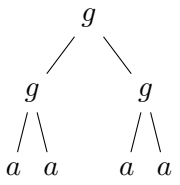



(3) on utilise $g(q_a, q_g) \to q_g$

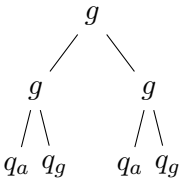
$q_g$

(4) $q_g \in Q_f$ t est reconnu

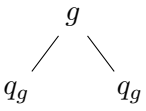
FIGURE 2.2: Reconnaissance d'un arbre par l'AAFA A de l'Exemple 2.1




(1) l'arbre $t$




(2) on utilise $a \to q_a$ et $a \to q_g$




(3) on utilise $g(q_a, q_g) \to q_g$

Blocage

(4) Aucune transition ne peut être appliquée

FIGURE 2.3: Arbre non reconnu par un AAFA

### 2.1.1 Langages reconnaissables

Exactement comme pour les langages sur les mots, les automates d'arbres définissent une nouvelle classe de langages appelés *langages reconnaissables*. Cette classe est d'une extrême importance car elle permet de reconnaître un ensemble fini ou infini d'arbres par une structure compacte et finie qui est l'automate d'arbre.

Maintenant, nous définissons ce que c'est qu'un langage reconnaissable.

**Définition 2.7.** *Un langage $L \in T_\Sigma$ est reconnaissable si et seulement s'il existe un AAFA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ tel que $L = L(\mathcal{A})$.*

L'ensemble des langages reconnaissables à travers un alphabet rangé $\Sigma$ est noté $\mathrm{Rec}(\Sigma)$.

De la même manière, nous définissions les langages liées aux états. En effet, nous distinguons deux types : les langages acceptés et les langages résiduels. Nous utilisons l'automate $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ dans les définitions suivantes.

**Définition 2.8.** *Un langage accepté (bas) $L(q)$ d'un état $q \in Q$ est $L(q) = \{t \in T_\Sigma \mid \delta(t) = q\}$.*

*Un langage résiduel (haut) $L^\uparrow(q)$ d'un état $q \in Q$ est obtenu comme suit :*

$$\begin{aligned} \zeta(q) &= \bigcup_{s \in L^\downarrow(q)} t(s \Leftarrow q) \\ L^\uparrow(q) &= \{t(q \leftarrow \Box) \mid t \in \zeta(q), \delta(t) \in Q_f\} \end{aligned}$$

*où $\Box \notin \Sigma$ est un symbol spécial.*

Prenons l'automate présenté dans la Figure 2.1.

Nous avons :

- $L(\mathcal{A}) = \{g(a,a), g(a,g(a,a)), g(a,g(a,g(a,a))), \ldots\}$
- $L(q_a) = L(q_g) = \{a\}$
- $L^\uparrow(q_a) = \{g(\Box,a), g(a,g(a,\Box)), g(\Box,g(a,a)), \ldots\}$

En se basant sur les langages acceptés et résiduels d'états, nous définissons l'opération de réduction d'automate.

### Automate réduit

Nous allons montrer que dans un automate d'arbres, il est possible que certains états qu'on va appeler *inutiles* ne participent pas dans le processus la reconnaissance et peuvent donc être retirés de l'automate en question [CDG+07].

**Définition 2.9.** *(État accessible) on dit qu'un état $q$ est accessible si et seulement si $L(q) \neq \emptyset$.*

**Définition 2.10.** *(État co-accessible) on dit qu'un état $q$ est co-accessible si et seulement si $t \in T_{\Sigma \cup \{q\}}$ tel que $q \in St(t)$ et $\delta(t) \in Q_f$.*

**Définition 2.11.** *(État inutile) On dit qu'un état est inutile s'il n'est pas accessible ou co-accessible.*

Par conséquent :

**Définition 2.12.** *Un AAFA est dit réduit si aucun de ses états n'est inutile.*

**Proposition 2.1.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA, $\dot{Q} \subseteq Q$ l'ensemble des états inutiles et $\dot{\Delta}$ l'ensemble des transitions où un état inutile figure. Alors l'AAFA réduit (AAFAR) $\mathcal{A}_r = (Q - \dot{Q}, \Sigma, Q_f \cap \dot{Q}, \Delta - \dot{\Delta})$ reconnaît exactement $L(\mathcal{A})$.*

La preuve est évidente. Juste il faut vérifier qu'un arbre qui appartient au langage reconnu par l'automate n'a pas besoin des états $\dot{Q}$ et des transitions $\dot{\Delta}$ pour sa reconnaissance.

**Algorithme de réduction** La fonction *Réduire* permet en un temps $\mathcal{O}(|\mathcal{A}|)$ de calculer l'automate réduit d'un AAFA $\mathcal{A}$.

L'algorithme crée deux ensembles $Acc$ et $Coa$ pour les états accessibles et co-accessibles. $Acc$ contient tous les états qu'on peut atteindre depuis une feuille (une transition de la forme $(a, q_a)$). Quand à $Coa$, elle regroupe tous les états qui ont un chemin vers un état final.

1: **Fonction** REDUIRE(Un AAFA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
2: $Acc \leftarrow \emptyset$ ▷ $Acc$ : les états accessibles
3: $Coa \leftarrow Q_f$ ▷ $Coa$ : les états co-accessibles
4: **répéter**
5: **si** $\exists q \in Q, f \in \Sigma_n, q_1, \ldots, q_n \in Acc, (f, q_1, \ldots, q_n, q) \in \Delta$ **alors**
6: $Acc \leftarrow Acc \cup \{q\}$
7: **Fin si**
8: **jusqu'a** Aucun état ne peut être ajouté à $Acc$
9: **répéter**
10: **si** $\exists q \in Coa, f \in \Sigma_n, q_1, \ldots, q_n \in Q, (f, q_1, \ldots, q_n, q) \in \Delta$ **alors**
11: $Coa \leftarrow Coa \cup \{q_1, \ldots, q_n\}$
12: **Fin si**
13: **jusqu'a** Aucun état ne peut être ajouté à $Coa$
14: $Q^r \leftarrow Coa \cup Acc$
15: $Q^r_f \leftarrow (Q_f \cap (Coa \cup Acc))$
16: $\Delta^r \leftarrow \{(f q_1, \ldots, q_n, q) \mid q, q_1, \ldots, q_n \in Coa \cup Acc\}$
17: **return** AAFAD $\mathcal{A}^r = (Q^r, \Sigma, Q^r_f, \Delta^r)$
18: **Fin Fonction**

**Exemple 2.2.** *Soit l'automate d'arbres* $\mathcal{A} = (\{q_a, q_g, q_1, q_2\}, \{a, b, g(,), f()\}, \{q_g\}, \{(a, q_a), (a, q_g), (g, q_a, q_g), (b, q_2), (f, q_1, q_g)\})$ *(voir Figure 2.4). Nous remarquons que l'état* $q_1$ *n'est pas accessible car* $L(q_1) = \emptyset$*. De plus, l'état* $q_2$ *qui est accessible (*$L(q_2) = \{b\}$*) n'est pas co-accessible car* $L^{\uparrow}(q_2) = \emptyset$*.*

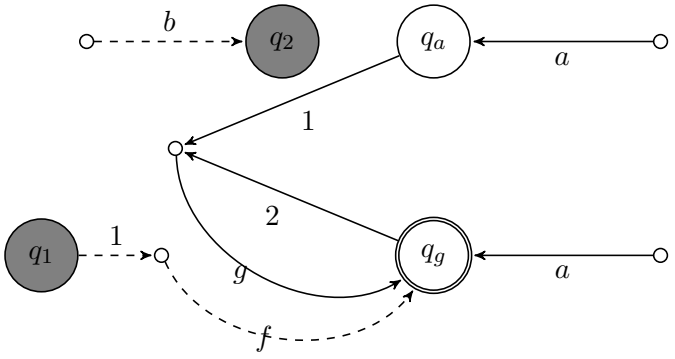



FIGURE 2.4: Automate avec états inutiles $\{q_1, q_2\}$

### 2.1.2 Quelques opérations sur les automates

Dans cette partie, nous étudions la propriété de fermeture de l'ensemble $Rec(\Sigma)$ à travers un alphabet rangé $\Sigma$ pour des opérations usuelles mais d'une grande importance pour la manipulation (représentation, traitement) des grandes masses de données.

#### 2.1.2.1 L'union

L'union a pour but de trouver un AAFA qui permet de reconnaître l'union de deux langages reconnaissables.

**Théorème 3.** $\mathrm{Rec}(\Sigma)$ *est fermé pour l'union.*

*Démonstration.* La preuve est évidente : Soient $L_1, L_2$ deux langages reconnaissables par deux AAFA $\mathcal{A}_1 = (Q_1, \Sigma, Q_f^1, \Delta_1), \mathcal{A}_2 = (Q_2, \Sigma, Q_f^2, \Delta_2)$ respectivement. Alors l'automate $\mathcal{A} = (Q_1 \cup Q_2, \Sigma, Q_f^1 \cup Q_f^2, \Delta_1 \cup \Delta_2)$ reconnaît $L_1 \cup L_2$. □

Cette preuve fournit une construction naturelle de l'automate de l'union.

#### 2.1.2.2 Le complément

L'opération de complétion peut s'appliquer sur les automates d'arbres. Il est à noter que cette notion est différente de la notion de l'automate complet.

L'automate *Complément* de $\mathcal{A}$, noté $\overline{\mathcal{A}}$, est :

$$\overline{\mathcal{A}} = (Q, \Sigma, Q - Q_f, \Delta)$$

Par conséquent, l'automate complément reconnaît le langage complément de $L(\mathcal{A})$ dans $T_\Sigma$.

**Corollaire 2.1.** $\mathrm{Rec}(\Sigma)$ *est fermé pour la complétion.*

#### 2.1.2.3 L'intersection

Avec le même principe, on peut construire l'automate qui reconnaît l'intersection de deux langages reconnaissables. En effet, la classe des langages reconnaissables est aussi fermée pour l'intersection :

$$L_1 \cap L_2 = \overline{\overline{L_1} \cup \overline{L_2}}$$

**Théorème 4.** $\mathrm{Rec}(\Sigma)$ *est fermée pour l'intersection.*

*Démonstration.* On peut construire l'automate d'intersection comme suit : Soit $\mathcal{A}_1 = (Q_1, \Sigma, Q_{f1}, \Delta_1)$ et $\mathcal{A}_2 = (Q_2, \Sigma, Q_{f2}, \Delta_2)$ deux AAFA, alors l'automate de composition est $\mathcal{A} = (Q_1 \times Q_2, \Sigma, Q_{f1} \times Q_{f2}, \Delta_1 \times \Delta_2)$. Il reconnaît le langage $L(\mathcal{A})_1 \cap L(\mathcal{A})_2$. □

#### 2.1.2.4 Équivalence d'automates

On peut dire que deux AAFA sont *équivalents* s'ils reconnaissent le même langage. Mais le problème qui se pose est de savoir si deux AAFA sont équivalents sachant qu'un langage reconnaissable peut être infini. Car l'acceptation de toutes les instances d'un ensemble infini ne peut pas être vérifiée.

**Théorème 5.** *Le problème d'équivalence de deux AAFA est décidable.*

*Démonstration.* En se basant sur les opérations définies sur les langages reconnaissables, on peut décider si deux automates $\mathcal{A}_1 = (Q_1, \Sigma, Q_f^1, \Delta_1), \mathcal{A}_2 = (Q_2, \Sigma, Q_f^2, \Delta_2)$ sont équivalents comme suit.

1. On calcule le complément du premier AAFA $\overline{\mathcal{A}_1}$
2. On calcule l'automate d'intersection $\overline{\mathcal{A}_1} \cap \mathcal{A}_2$
3. Si l'automate d'intersection est vide alors $L(\overline{\mathcal{A}_1}) = T_\Sigma - L_1 = T_\Sigma - L_2$ : les deux automates sont équivalents.

□

#### 2.1.2.5 Lemme de l'étoile

Le lemme de l'étoile, dit aussi de pompage, est la propriété principale pour les langages reconnaissables de mots et d'arbres. Il permet de montrer l'aspect de régularité de ses langages.

Avant d'introduire le lemme, nous définissons quelques notations.

**Définition 2.13.** *Un arbre $t \in T_{\Sigma \cup \{\Box\}}$ lié à la constante $\Box \notin \Sigma_0$ est un arbre dans lequel la feuille $\Box$ figure une seule fois dans $t$ :*

$$\exists d \in D, \forall d' \neq d \in D : Pos^t(d) = \Box \text{ et } Pos^t(d') \neq \Box$$

**Définition 2.14.** *(Concaténation $n$-aire) Soit $t \in T_\Sigma$, $(t._c)^n = t._c \overbrace{\ldots}^{n \text{ fois}} ._c t$.*

Maintenant, nous annonçons le lemme de l'étoile (pompage) [CDG+07] :

**Lemme 2.1.** *Soit $L \in Rec(\Sigma)$ un langage reconnaissable. Alors, il existe une constante $k \in \mathbb{N}$ qui satisfait : pour tout $t \in L$ tel que $\mathcal{P}(t) > k$, il existe un arbre $t' \in T_\Sigma$, deux arbres $u, v \in T_{\Sigma \cup \{\Box\}}$ liées à $\Box$ tel que $t = u._\Box(v._\Box t')$ et $\forall n \in \mathbb{N}, u._\Box((v._\Box)^n._\Box t') \in L$*

**Exemple 2.3.** *En prenant le même exemple illustré dans Figure 2.1. Nous remarquons que le cycle présenté en rouge dans Figure 2.5 permet de "pomper" des arbres de la forme $g(a, g(a, (\ldots)) \ldots)$ où chaque arbre pompé contient une branche $g2g2g \ldots g2a$. Suivant le lemme de l'étoile, nous avons $t = g(a, g(a, a)) = g(a, q_g)._{q_g}(g(a, q_g)._{q_g} a) \in L(\mathcal{A})$ et tout arbre $t' = g(a, q_g)._{q_g}(g(a, q_g).q_g)^n._{q_g} a) \in L(\mathcal{A})$ pour tout $n$.*

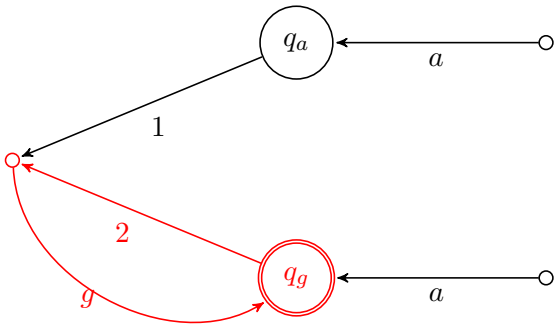



FIGURE 2.5: Pompage d'un arbre à l'aide d'un AADA

### 2.1.3 Déterminisation

Dans un automate déterministe, une seule transition est utilisée dans la progression de la reconnaissance d'un arbre et aucune ambiguïté n'est rencontrée. On doit commencer à partir de l'unique état pour chaque symbole de $\Sigma_0$, et lorsqu'on lit un nouveau symbole avec un ensemble d'états en entrée, il n'y a qu'une seule transition que l'on peut emprunter. En effet, tout langage reconnaissable possède un automate ascendant déterministe qui le reconnaît. Malheureusement, l'algorithme de déterminisation connu possède une complexité polynomiale pire-cas [CDG+07].

On peut grâce à la déterminisation supprimer l'ambiguïté présente dans un AAFA, car dans ce dernier, on peut trouver plusieurs manières de calculer un arbre.

**Automate déterministe** Un automate est dit déterministe si la reconnaissance de tout arbre se fait en utilisant exactement le même nombre d'états que sa taille.

**Définition 2.15.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA. On dit que $\mathcal{A}$ est déterministe (AAFAD) si et seulement s'il existe deux transitions $(f, q_1, \ldots, q_n, q), (f, q_1, \ldots, q_n, q') \in \Delta$ où $f \in \Sigma_n, q_1, \ldots, q_n, q, q' \in Q$ alors $q = q'$*

La fonction de sortie $\delta$ peut être redéfinie sur $Q$ au lieu de $2^Q$ : $\delta : T_\Sigma \to Q$.

**Exemple 2.4.** *Soient les deux AAFA $\mathcal{A} = \{\{q_0, q_1, q\}, \{a, g()\}, \{q_0\}, \{(a, q_0), (g, q_0, q_1), (g, q_1, q_0), (g, q, q_0), (g, q, q_1)\}\}$ et $\mathcal{B} = \{\{q_a, q_g, q_f\}, \{a, g(), f(,)\}, \{q_f\}, \{(a, q_a), (g, q_a, q_g), (g, q_g, q_g), (f, q_g, q_g, q_f)\}\}$.*

*L'automate $\mathcal{A}$ n'est pas déterministe à cause de la présence des deux transitions $(g, q, q_0)$ et $(g, q, q_1)$. Par contre, l'automate $\mathcal{B}$ est déterministe.*

#### Algorithme de déterminisation

Nous présentons l'existence d'un automate déterministe suivant le Théorème 6.

**Théorème 6.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA, alors il existe un AAFAD qui reconnait $L(\mathcal{A})$.*

*Démonstration.* La preuve du théorème se fait en construisant un algorithme de déterminisation d'un automate.

Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA. Nous définissons sa version déterministe $\mathcal{A}_d = (Q_d, \Sigma, Q_{df}, \Delta_d)$ comme suit. $Q_d \subseteq 2^Q$. Les transitions $\Delta_d$ sont construites à partir de $\Delta$ comme suit :

$$(f, r_1, \ldots, r_{\hat{f}}) \in \Delta_d \Leftrightarrow r = \{q \in Q \mid \exists q_1 \in r_1, \ldots, q_{\hat{f}} \in r_{\hat{f}}, (f, q_1, \ldots, q_{\hat{f}}) \in \Delta\}$$

Et $Q_{df} = \{r \mid \exists r \cap Q_f \neq \emptyset\}$ son ensemble d'états finaux.

On peut vérifier que $L(\mathcal{A}_d) = L(\mathcal{A})$ en vérifiant la double inclusion $L(\mathcal{A}_d) \subseteq L(\mathcal{A})$ et $L(\mathcal{A}) = L(\mathcal{A}_d)$.

□

**Corollaire 2.2.** *Le langage accepté de tout état d'un AAFAD est unique.*

Nous pouvons facilement prouver ce corollaire en utilisant directement la définition 2.15.

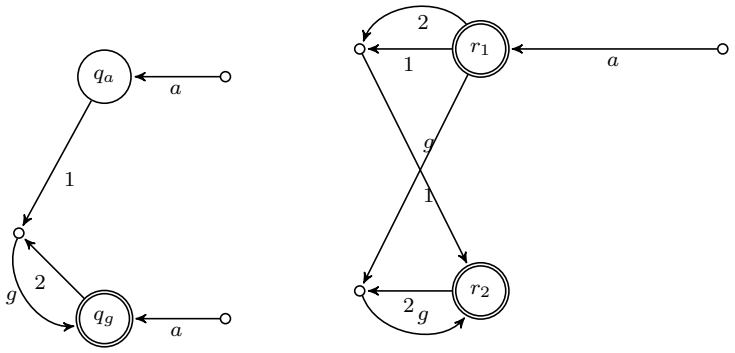
Maintenant, nous présentons un algorithme polynomial pour déterminiser un automate donné [CDG+07].

1: **Fonction** DÉTERMINISER(Un AAFA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
2: $Q_d \leftarrow \emptyset$
3: $\Delta_d \leftarrow \emptyset$
4: **répéter**
5: **pour tout** $(f, q_1, \ldots, q_{\hat{f}+1}) \in \Delta$ **faire**
6: **pour tout** $q \in q_1, \ldots, q_{\hat{f}}$ **faire**
7: $temp_i = \{r \mid q_i \in r\}$
8: **Fin pour**
9: **si** $\forall temp_i, temp_i \neq \emptyset$ **alors**
10: Ajouter $(f, r_1, \ldots, r_{\hat{f}+1})$ à $Q_d$ tel que $r_i \in temp_i$
11: **Fin si**
12: **Fin pour**
13: **jusqu'a** Aucune transition ne peut être ajoutée
14: $Q_d \leftarrow \{(f, r_1, \ldots, r, \ldots, r_{\hat{f}+1}) \in \Delta_d\}$
15: $Q_d f \leftarrow \{r \mid r \cap Q_f \neq \emptyset\}$
16: **retourner** AAFAD $\mathcal{A}_d = (Q_d, \Sigma, Q_{df}, \Delta_d)$
17: **Fin Fonction**

**Exemple 2.5.** *Prenons le même exemple que l'Exemple 2.1. En appliquant la fonction de déterminisation, les étapes sont :*

1. *Ajouter* $(a, r_1), r_1 = \{q_a, q_g\}$ *(a est une feuille et* $\delta(a) = \{q_a, q_g\}$*),*
2. *Ajouter* $(g, r_1, r_1, r_2), r_2 = \{q_g\}$ *à* $\Delta_d$ $(q_a, q_g \in r_1$ *et* $\delta(g, q_a, q_g) = \{q_g\})$*,*
3. *Ajouter* $(g, r_1, r_2, r_2)$ *à* $\Delta_d$*,*
4. *Ajouter* $r_1, r_2$ *a* $Q_{df}$*.*

*Le résultat de la déterminisation est montré dans la Figure 2.6.*




*Automate original non déterministe* *version déterministe*

FIGURE 2.6: Déterminisation d'un AAFA

Nous notons que l'algorithme de déterminisation est polynomial en temps et en mémoire. Un cas de figure est présenté dans l'exemple suivant :

**Exemple 2.6.** *Soit* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *un AAFA où* $Q = \{q, q_1, \ldots, q_{n+1}\}, \Sigma = \{a, f(), g()\}$ *et*

$$\begin{aligned} \Delta &= \{(a, q), (g, q, q), (f, q, q), (f, q, q_1)\} \\ &\cup \{(g, q_i, q_{i+1}) \mid 1 \leq i \leq n\} \\ &\cup \{(f, q_i, q_{i+1}) \mid 1 \leq i \leq n\} \end{aligned}$$

*La determinisation de cet automate mène à un automate déterministe avec* $2^{n+1}$ *états.*

## 2.2 Autres types d'automates d'arbres

Dans cette section, nous présentons d'autres types d'automates d'arbres à savoir les automates descendants, les automates d'arbres non-rangés, les automates à haie et les automates avec multiplicités.

### 2.2.1 Automates d'arbres rangés à états finis descendants

**Définition 2.16.** *Un automate d'arbres rangés à états finis descendant (AAFD) à travers un alphabet rangé* $\Sigma$ *est un quadruplet* $\mathcal{D} = (Q, \Sigma, Q_i, \Delta)$ *tel que :*

1. $Q$ *est un ensemble de symboles appelés états,*
2. $Q_i \subseteq Q$ *est un ensemble d'états initiaux,*
3. $\Delta \subseteq Q \times \cup_{n \geq 0} \Sigma_n \times Q^n$ *est l'ensemble des transitions.*

Nous définissons la fonction de transition $\delta$ comme suit : $\delta : Q \times T_\Sigma \rightarrow T_\Sigma$ définie pour $t = f(t_1, \ldots, t_n)$ comme :

$$\delta(q, t) = \{f(\delta(q_1, t_1), \ldots, \delta(q_n, t_n)) \mid \exists (q, f, q_1, \ldots, q_n) \in \Delta\} \tag{2.3}$$

Un arbre $t$ est dit *reconnu* par un l'AAFD $\mathcal{D}$ si $\exists q \in Q_i \mid \delta(q, t) = t$.

Certaines références [CDG+07] utilisent le formalisme des systèmes de règles de réécriture. $\Delta$ est alors définie comme suit :

$$q \rightarrow f(q_1, \ldots, q_n) \text{ tel que } f \in \Sigma_n, q_1, \ldots, q_n \in Q$$

**Exemple 2.7.** *Soit l'automate suivant :*

$\mathcal{D} = (\{q_f, q_g, q_a\}, \{f(,), g(), a\}, \{q_f\}, \{(q_f, f, q_g, q_g), (q_g, g, q_a), (q_a, a)\})$ *un AAFD et un arbre* $t = f(g(a), g(a))$. *Nous avons :*

$$\begin{aligned} \delta(q_f, t) &= f(\delta(q_g, g(a)), \delta(q_g, g(a))) \textit{ nous avons } (q_f, f, q_g, q_g) \in \Delta \\ &= f(g(\delta(q_a, a)), g(\delta(q_a, a))) \textit{ nous avons } (q_g, g, q_a) \in \Delta \\ &= f(g(a), g(a)) = t \end{aligned}$$

*Alors $t$ est reconnu par $\mathcal{D}$.*

Le problème des automates descendants réside dans leurs versions déterministes. Un AAFD est déterministe si et seulement si :

1. $|Q_i| = 1$
2. $\forall q \in Q, |\{(q, f, q_1, \ldots, q_n) \mid q_1, q_n \in Q, f \in \Sigma_n\}| \leq 1$

En effet, la version déterministe d'un AAFD non déterministe n'existe pas toujours. Nous présentons un exemple de référence (voir [CDG+07]).

**Exemple 2.8.** *Soient $t = f(a, b)$ et $t' = f(b, a)$ deux arbres définis sur $\Sigma = \{f(,), a, b\}$. L'automate non déterministe $\mathcal{D} = (\{q_f, q_a, q_b\}, \Sigma, \{q_f\}, \{(q_f, f, q_a, q_b), (q_f, f, q_b, q_a), (q_a, a), (q_b, b)\})$ reconnaît $t, t'$.*

*Maintenant supposons que $\mathcal{D}'$ est un automate déterministe qui reconnaît $t, t'$. Par définition, nous avons un seul état initial $q_i$ alors $|\delta(q_i, t)| = \delta(q_i, t')| = 1$. Supposons qu'il existe $q', q''$ tel que $\delta(q_i, t) = f(\delta(q', a), \delta(q'', b))$. Pour reconnaître $t'$ il faut que $(q', a) \in \Delta$. Mais comme $\mathcal{D}$ est déterministe alors il faut que $(q', b) \in \Delta$. Ceci est considéré comme une contradiction.*

*Remarque* 2.2. Notons que les automates ascendants sont plus performants que leurs analogues descendants car la version déterministe peut être déduite (en un temps polynomial).

### 2.2.2 Automates d'arbres non rangés

La théorie avancée de XML fait appel à plusieurs domaines de la recherche d'informations (RI), des bases de données et des langages formels [Via01, LNG06]. Les documents XMLs peuvent être représentés par des arbres, donc reconnus par des automates d'arbres.

```
<?xml version="1.0"?>
<annuaire>
    <personne>
        <nom>HAWKING</nom>
        <prenom>Stephen</prenom>
    </personne>
    <personne>
        <nom>REEVES</nom>
        <prenom>Hubert</prenom>
    </personne>
    <personne>
        <nom>EINSTEN</nom>
        <prenom>Albert</prenom>
    </personne>
</annuaire>
```

FIGURE 2.7: Exemple de document XML

Les automates d'arbres rangés, malgré leur puissance, ne cadrent pas une bonne modélisation des documents XMLs. En effet, le rang exige que chaque symbole de l'alphabet ait un nombre fixe de fils, chose qui n'est pas garantie dans tous les cas dans les documents XMLs. L'exemple suivant illustre ce problème.

**Exemple 2.9.** *Soit le document XML suivant (Figure 2.7).*

*Nous remarquons que le nœud comporte trois entrées du type "personne". Néanmoins l'ajout d'une nouvelle personne à l'annuaire exige la prise en compte du nombre variable des nœuds fils que peut avoir le nœud "annuaire".*

La première solution à ce problème est de refaire les définitions des automates rangés en modifiant l'alphabet. Cette fois ci, l'alphabet ne possède pas de rang et tout symbole peut apparaître comme feuille, racine ou arbre intermédiaire.

**Définition 2.17.** *Un AAFA non-rangé (ou bien à arité arbitraire) $\mathcal{N}$ est un tuple $\mathcal{N} = (Q, \Sigma, Q_f, \Delta)$ où $Q$ est un ensemble d'états, $\Sigma$ est un ensemble de symboles (libre du rang), $Q_f \subseteq Q$ est un ensemble d'états finaux et $\Delta \subset \bigcup_{n=1}^{\infty} \Sigma \times Q^n$ est un ensemble de transitions.*

La différence entre cette définition et la définition des automates rangés réside dans les transitions. Une transition dans un automate non rangé n'exige aucune restriction sur les symboles de l'alphabet utilisé. Quant aux transitions des automates rangés, un symbole figure dans une transition si son rang est égale à la longueur de la transition moins un.

Néanmoins, les automates non rangés ont plus de représentativité vis-à-vis des automates rangés.

**Théorème 7.** *Un langage reconnaissable par un AAFA non rangé est reconnaissable par un AAFA rangé.*

*Démonstration.* Soit $\mathcal{N} = (Q, \Sigma, Q_f, \Delta)$ un AAFA non rangé. Soit $\Sigma' = \bigcup_{f \in \Sigma, i \geq 0} f_i$ un alphabet rangé tel que $\forall f_i \in \Sigma', \hat{f}_i = i$. Nous définissons l'automate rangé $\mathcal{A} = (Q, \Sigma', Q_f, \Delta')$ tel que $(f_n, q_1, \ldots, q_{n+1})$ appartient à $\Delta'$ si et seulement si $(f, q_1, \ldots, q_n) \in \Delta$.

Nous définissons sur $T_\Sigma$ une fonction $\Psi : T_\Sigma \to T_{\Sigma'}$ tel que $\Psi(f(t_1, \ldots, t_n)) = f_n(\Psi(t_1), \ldots, \Psi(t_n))$. Alors, il est clair que s'il existe un arbre $t \in T_\Sigma$ où $t \in L(\mathcal{N}) : \Psi(t) \in L(\mathcal{A})$. □

**Exemple 2.10.** *Soit* $\mathcal{N} = (\{q_a, q_b\}, \{a, b\}, \{q_2\}, \{(a, q_a), (b, q_b), (a, q_a, q_a, q_b)\})$ *alors l'automate rangé reconnaissant* $\Psi(L(\mathcal{N}))$ *est :*

$$\mathcal{A} = (\{q_a, q_b\}, \{a_0, a_2, b_0\}, \{q_2\}, \{(a_0, q_a), (b_0, q_b), (a_2, q_a, q_a, q_b)\})$$

En effet, cette transformation consiste à affecter à chaque symbole un coefficient qui correspond à la longueur de la transition où il apparaît. Si le même symbole apparaît deux fois dans deux transitions de longueurs différentes, il est transformé en deux nouveaux symboles dont le rang est unique pour chacun. De ce fait, toute l'algorithmique des automates rangés peut être appliquée aux automates non rangés ainsi définis.

### 2.2.3 Automates à haie

Une autre classe d'automates d'arbres non rangés est définie afin de permettre une représentation plus efficace pour les automates d'arbres. Cette classe s'inspire de celle des DTDs. Les automates de Brüggemann-Klein et al. [BKMW01] qui sont connus sous le nom d'"*automates à haie*"(Hedge Automata en anglais). Ce type d'automates permet de manipuler les transitions comme étant des "langages de mots". Cependant, cette nouvelle classe rencontre des problèmes de modélisation et d'efficacité.

**Définition 2.18.** *Un automate d'arbres à haie (AAH)* $\mathcal{H}$ *est le tuple* $\mathcal{H} = (Q, \Sigma, Q_f, \Delta)$ *où :*

1. $Q$ *est un ensemble d'états,*
2. $\Sigma$ *est un alphabet non rangé,*
3. $Q_f \subseteq Q$ *est un ensemble d'états finaux,*
4. $\Delta \subset \Sigma \times \mathrm{Rat}(Q) \times Q$ *est un ensemble fini de transitions (*$\mathrm{Rat}(Q)$ *est l'ensemble des expressions rationnelles (ER) de mots à travers* $Q$*).*

Nous définissons la fonction de sortie $\delta : T_\Sigma \to 2^Q$ qui reconnaît un arbre $t = f(t_1, \ldots, t_n)$ comme suit :

$$\delta(t) = \{q \mid \exists (f, E, q) \in \Delta \text{ et } \delta(t_i) \in [\![E]\!] \text{ pour } i = 1, \ldots, n\} \tag{2.4}$$

Selon la terminologie des règles de réécriture, on peut écrire $\Delta$ sous la forme d'un système de transitions de la forme $a(E_Q) \to q$ pour dénoter une transition $(a, E_Q, q)$.

**Exemple 2.11.** *Soit* $\mathcal{H} = (Q, \Sigma, Q_f, \Delta)$ *tel que :*

1. $Q = \{q_1, q_2\}$,
2. $\Sigma = \{a, b\}$,
3. $Q_f = \{q_2\}$,
4. $\Delta = \{(a, 1, q_1), (a, q_1^*, q_2), (b, q_1 + q_2, q_2)\}$. *(Nous rappelons que* $1$ *est l'expression rationnelle qui dénote le mot vide)*

*L'arbre* $t = a(b, a(a, a))$ *n'est pas reconnu car* $b$ *considéré en tant que constante ne peut être calculé par aucune transition. Quant à l'arbre* $t' = b(b(a))$*, il peut être reconnu car* $a$ *comme feuille donne* $q_1$ *en utilisant la première transition et* $q_2$ *en utilisant la deuxième,* $b(q_1)$ *donne* $q_2$ *car dans la deuxième transition* $q_2 \in [\![q_1 + q_2]\!]$*. La ré-application de la deuxième transition donne* $q_2$ *qui est un état final.*

Plusieurs problèmes sont à surmonter pour les automates à haie. Le premier réside dans le fait que la définition de la fonction d'acceptation doit opérer sur des ER de mots. Ces dernières ne sont pas faites initialement pour reconnaître mais pour décrire des langages de mots [HU79]. Le deuxième est de définir la notion de déterminisme et l'algorithme de déterminisation. Quant au troisième, c'est la recherche d'un algorithme de minimisation efficace et de prouver son unicité.

Concernant la déterminisation, nous présentons une définition de l'automate à haie déterministe [CDG+07].

**Définition 2.19.** *Un automate à haie* $\mathcal{H} = (Q, \Sigma, Q_f, \Delta)$ *est déterministe si et seulement si pour toute transition* $(f, E, q) \in \Delta$*, aucune transition de la forme* $(f, E', q')$ *tel que* $[\![E]\!] \cap [\![E']\!] \neq \emptyset$ *n'existe dans* $\Delta$*.*

**Exemple 2.12.** *Prenons l'automate défini dans l'exemple 2.11. Nous remarquons qu'il n'est pas déterministe car* $[\![1]\!] \cap [\![q_1^*]\!] = \{\epsilon\} \neq \emptyset$*.*

Maintenant, pour décider qu'une séquence $f, q_1, \ldots, q_n$ soit réduite en $q$, il faut que le mot $q_1 \ldots q_n$ appartienne à une expression $E$ tel que $(f, E, q) \in \Delta$. Comme les expressions rationnelles sont utilisées pour dénoter un langage et non pas pour reconnaître des mots, la technique adéquate à ce problème est de créer pour chaque expression présente dans une transition un automate de mots qui reconnaît sa sémantique.

### 2.2.4 Automate à multiplicités

La modélisation de la multiplicité dans les arbres présente maintenant un domaine d'application ouvert comme dans les travaux de Bailly et al. [BDR09] et de Balle et al. [BCLQ14]. Pour cette raison, les automates d'arbres à multiplicité donnent un outil formel puissant pour manipuler ce type d'objets [RBC16].

Afin de considérer la dimension "multiplicité" aux AAFA, une fonction *poid* doit être ajoutée. De plus, le langage reconnaissable n'est plus un langage d'arbres mais une série d'arbres. Nous définissons l'*automate d'arbres finis ascendant avec multiplicité* selon [BDZ15] :

**Définition 2.20.** *Un* Automate d'Arbres Finis Ascendant avec Multiplicité (AAFAM) *est le tuple* $\mathcal{A} = (\Sigma, Q, \Phi, \eta, \alpha)$ *où :*

1. $\Sigma$ *est un alphabet rangé quelconque.*
2. $Q$ *est un ensemble finis de symboles appelés "états".*
3. $\Phi = (\mathbb{K}, \oplus, \otimes, 0, 1)$ *un semi anneau.*
4. $\eta : \cup_{i=1}^{n} \Sigma_i \times Q^{i+1} \to \mathbb{K}$ *est la fonction* poids*.*
5. $\alpha : Q \to \mathbb{K}$ *est la fonction des* poids finaux*.*

*On peut définir ou ajouter un ensemble de transitions de la même manière que dans les AAFA généraux :* $\Delta = \cup_{i=1}^{\hat{r}} \Sigma_i \times Q^{i+1}$*. Une transition* $\rho \in \Delta$ *est dite* valide *si* $\eta(\rho) \neq 0$*. Un état* $q \in Q$ *est final si* $\alpha(q) \neq 0$*. L'automate* $\mathcal{A}$ *est* déterministe *si pour tout* $m \geq 0, \rho \in \Sigma_m \times Q^{m+1}$ *il existe au plus un état* $q \in Q$ *tel que* $\eta(\rho) \neq 0$*.*

La fonction de poids peut être vue comme une matrice indexée par des tuples de $Q$ ([HMV09]) :

$$\eta_k : \Sigma_k \to \mathbb{K}^{Q^k \times Q}$$

Si l'image d'un élément $f \in \Sigma$ vaut 0 dans la case $q_1 \dots q_k, q$ alors la transition $(f, q_1, \dots, q_k, q)$ n'existe pas. L'implémentation de cette matrice prend en compte la complétude de l'automate en question : s'il n'est pas complet, une matrice creuse peut être engendrée.

Pour définir les séries d'arbres reconnaissables, il faut avoir une fonction de reconnaissance d'un arbre avec multiplicités. La fonction de *sortie* opère sur des arbres de la forme $f(t_1, \dots, t_{\hat{f}})$ en commençant par traiter les sous arbres $t_1, \dots, t_{\hat{f}}$ (forme récursive). Formellement, la fonction de sortie est :

$$\delta_\eta(f(t_1, \dots, t_{\hat{f}}))_q = \bigoplus_{q_1,\dots,q_k \in Q^{\hat{f}}} \eta(f, q_1, \dots, q_{\hat{k}}, q) \otimes \delta_\eta(t_1)_{q_1} \otimes \dots \otimes \delta_\eta(t_{\hat{f}})_{q_k}. \tag{2.5}$$

Le *coefficient* d'un arbre $t$ est :

$$C_t = \bigoplus_{q \in Q} \alpha(q) \otimes \delta_\eta(t)_q$$

Un arbre est dit *reconnaissable* si :

$$C_t \neq 0$$

## 2.3 Conclusion

Nous avons présenté les différents aspects généraux des automates d'arbres. En effet, nous avons donné les définitions d'automates d'arbres rangés, ceux ciblés par notre étude. Nous avons mis le point sur les opérations élémentaires qui leurs sont associées à savoir la reconnaissance et la déterminisation. En dernier, nous avons brièvement revu d'autres types d'automates à savoir les automates non rangés, les automates à haie et ceux augmentés par la multiplicité.

Dans le chapitre suivant, nous nous focalisons sur les automates d'arbres ascendants déterministes. En effet, le déterminisme est une notion très importante dans la conception de nouveaux algorithmes efficaces dans la pratique. Nous étudions en particulier l'opération de minimisation afin de générer une structure la plus compacte possible reconnaissant une infinité d'arbres.

# Chapitre 3

# État de l'art sur la minimisation

La minimisation d'automates consiste à réduire la taille d'un automate sans modification de son langage reconnaissable. Cette opération est largement utilisée dans les automates à mots afin de trouver une structure aussi compacte que possible tout en gardant la reconnaissabilité [HU79].

Dans la littérature, il existe deux variantes de techniques de minimisation. La première, la plus connue, est la minimisation du nombre d'états tandis que la deuxième est la réduction du nombre de transitions.

La minimisation est primordiale car elle permet de réduire en mémoire un automate d'arbres reconnaissant un langage d'arbres quelconque. Dans ce chapitre, nous allons présenter les différentes techniques de minimisation d'automate d'arbres. Nous nous focalisons sur la minimisation d'automates déterministes vu que ce type d'automates est plus utile dans la pratique.

Toutes les techniques de minimisation d'AAFAD de référence [Bra67, AG68, GS80] sont largement inspirées de leurs analogues sur les mots. Ces dernières étaient étudiées pour la première fois par Huffman et Moore [Moo56]. Leur algorithme est basé sur la notion de *paires d'états distincts*. A la fin de l'algorithme, les états qui ne sont pas distincts sont fusionnés afin de construire l'automate minimal. Après, Hopcroft [Hop71] a défini un nouvel algorithme qui procède par la minimisation de tout l'ensemble d'états en utilisant une technique de raffinement. Cette dernière consiste à effectuer une division successive sur l'ensemble des états jusqu'à arriver à la partition d'états équivalents qui assure la minimalité de l'automate.

Par analogie, les techniques de minimisation d'AAFAD ont vu le jour. La première généralisation de la notion de minimisation pour les arbres fût proposée en 1967 par Brainerd [Bra67]. Cet algorithme est resté la base de plusieurs propositions comme Arbib et Giv'on [AG68], Geseg [GS80]et Comon et al. [CDG+07].

Nous avons recensé ces propositions sur la Figure 3.1. En effet, nous avons classé ses techniques hiérarchiquement selon deux critères :

- *Nature d'arbres traités* : deux familles de techniques existent, la minimisation d'AAFAD rangé et celle pour les AAFAD non rangés.

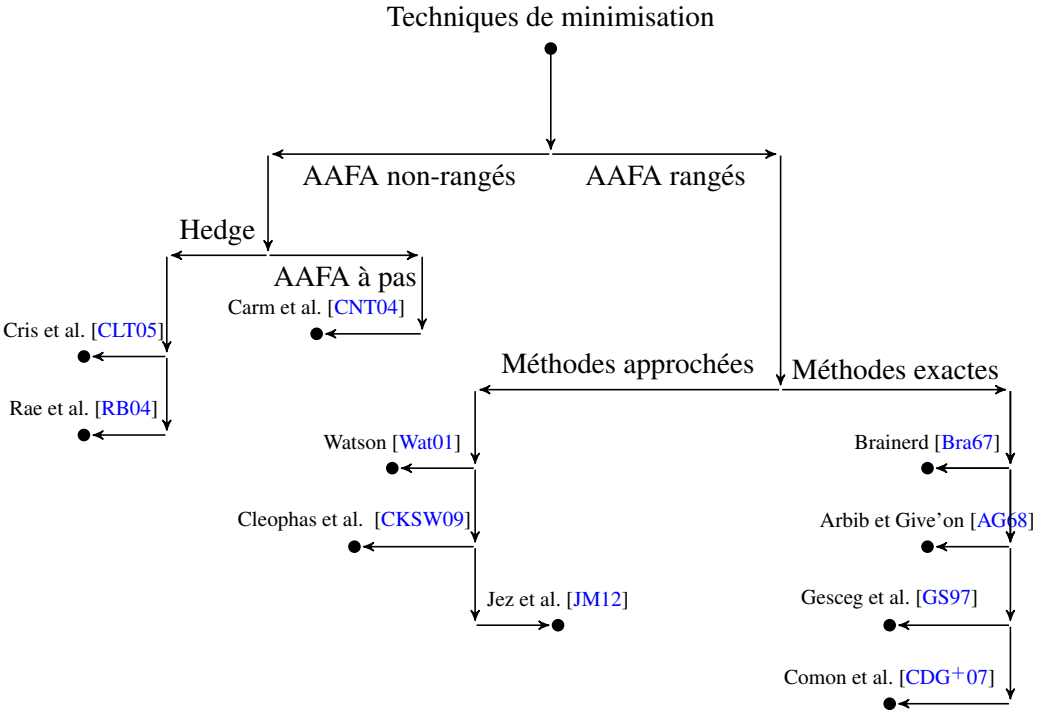



FIGURE 3.1: Classification des techniques de minimisation

- *Nature de l'automate résultant* : encore deux classes sont extraites, la première dite *exacte* regroupe les techniques produisant l'automate minimal unique. Quant à la seconde dite *approchée*, elle assemble les techniques produisant un automate plus compacte mais qui n'est pas nécessairement l'unique minimal. Cette branche tolère la différence de topologie (différence des automates eux-mêmes) et aussi la différence de langages acceptés.

Pour prouver l'existence et l'unicité d'un automate minimal équivalent à un AAFAD, il existe une généralisation du théorème de Myhill-Nerode pour les mots [HU79] aux arbres. Cette généralisation a été introduite pour la première fois par Kozen [Koz92].

## 3.1 Théorème de Myhill-Nerode pour les arbres

Avant d'introduire l'énoncé de ce théorème, nous définissons la relation de *congruence* entre les états d'un AAFAD.

**Définition 3.1.** *Une relation d'équivalence "$\equiv$" définie sur $T_\Sigma$ est une* congruence *si*

$$\forall f \in \Sigma_n, u_i, v_i \in T_\Sigma, i \in \{0, \ldots, n\} \ : u_i \equiv v_i \Rightarrow f(u_1, \ldots, u_n) \equiv f(v_1, \ldots, v_n),$$

La classe d'équivalence d'un arbre $t$ par la congruence $\equiv$ est l'ensemble $[t]_\equiv = \{u \mid t \equiv u\}$.

L'ensemble quotient de $T_\Sigma$ par $\equiv$ qui est un sous ensemble de $2^{T_\Sigma}$ est : $T_\Sigma/_\equiv = \{[t]_\equiv \mid t \in T_\Sigma\}$.

Par conséquent, l'ensemble quotient d'une relation de congruence permet de partitionner l'ensemble $T_\Sigma$ comme tel.

On dit que $\equiv$ est une relation d'indice fini si et seulement si :

$$\exists n \in \mathbb{N}, |T_\Sigma/_\equiv| < n$$

La *congruence des langages*$\equiv_L$ d'un langage $L \in T_\Sigma$ est définie comme suit :

**Définition 3.2.** *Soit $L \subseteq T_\Sigma$ un langage d'arbres, $s, t \in L$. Alors pour tout arbre $u \in T_{\Sigma\cup\{x\}}$ lié au symbole $x \notin \Sigma_0$ nous avons :*

$$(s \equiv_L t) \Leftrightarrow (u[x \leftarrow s] \in L \Leftrightarrow u[x \leftarrow t] \in L) \tag{3.1}$$

La généralisation du théorème de Myhill-Nerode pour les arbres est reportée par le théorème suivant :

**Théorème 8.** *Les déclarations suivantes sont équivalentes :*

**(i)** *$L$ est un langage reconnaissable,*

**(ii)** *$L$ est l'union d'un ensemble donné de classes d'équivalence d'une congruence à indice fini,*

**(iii)** *La relation $\equiv_L$ est une congruence à indice fini.*

*Démonstration.* **(i)→(ii)** $L$ est reconnaissable, alors il existe un AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ qui reconnaît $L$ (voir théorème 6). Nous définissons la relation d'équivalence $\sim$ suivante : $t \sim s$ si $\delta(t) = \delta(s)$. Il est clair que $\sim$ est une relation d'équivalence (égalité), de plus c'est une congruence (le déterminisme de l'automate assure la congruence). Enfin, $\sim$ est d'indice fini car elle utilise l'ensemble fini des états.

**(ii)→(iii)** Soit $\equiv$ une congruence à indice fini, $r, s \in T_\Sigma$ tel que $r \equiv s$. Soit $u \in T_{\Sigma\cup\{x\}}$ un arbre quelconque lié à $x$. Alors, nous avons $u[x \leftarrow s] \equiv u[x \leftarrow r]$ (Tous les sous arbres de $u$ sont équivalents à eux-mêmes et par définition d'une congruence la substitution par $r$ et $s$ reste équivalente).

Maintenant, soit $L$ l'union de congruences $\equiv$ à indice fini. Nous avons $u[x \leftarrow s] \in L$ si $u[x \leftarrow r] \in L$. Cela conduit à la définition d'une relation $\equiv_L$ qui est à indice fini puisque au maximum le nombre de ses classes d'équivalence égale à la somme des classes d'équivalences de l'union des congruences qui est finie.

**(iii)→(i)** Nous construisons un AAFAD qui reconnaît un langage $L \in T_\Sigma$ muni d'une relation $\equiv_L$ à indice fini. L'automate $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ tel que :

- $Q = \{[t]_{\equiv_L} \mid ; t \in T_\Sigma\}$,
- $Q_f = \{[t]_{\equiv_L} \mid ; t \in L\}$,
- $\Delta = \{(f, [t_1]_{\equiv_L}, \ldots, [t_{\hat{f}}]_{\equiv_L}, [f(t_1, \ldots, t_{\hat{f}})]_{\equiv_L}) \mid f \in \Sigma, t_1, \ldots, t_{\hat{f}} \in T_\Sigma\}$

On peut vérifier par induction sur la définition de l'ensemble de tous des arbres que l'automate ainsi défini reconnaît $L$.

□

Ce théorème constitue le fondement théorique de la minimisation des AAFAD. Non seulement il met en évidence l'existence d'un automate qui reconnaît un langage reconnaissable mais il permet de prouver son unicité.

**Définition 3.3.** *(Équivalence de Nerode) Deux états $q$ et $q'$ sont* équivalents *(on note $q \simeq q'$) si leurs langages résiduels sont égaux.*

$$q \simeq q' \Leftrightarrow L^{\uparrow}(q) = L^{\uparrow}(q')$$

*Ainsi, deux états sont* distincts *, s'ils ne sont pas équivalents.*

**Corollaire 3.1.** *Un AAFAD est minimal si tous ses états sont distincts.*

*Démonstration.* **(⇒)** Soient $q, q'$ deux états équivalents. Soit $t \in L(\mathcal{A})$ tel qu'il existe un arbre $s \in St(t), \delta(s) = q$. Nous déduisons alors que $t(s \leftarrow \Box) \in L^{\uparrow}(q)$ et $t(s \leftarrow \#) \in L^{\uparrow}(q')$ pour un symbole $\Box \notin T_{\Sigma}$. Par conséquent, on peut remplacer chaque occurrence de $q'$ par $q$ dans $\Delta$ sans affecter le langage reconnu. Cette transformation met à jour le langage accepté $L(q)$ par $q$ en $L(q) \cup L(q')$ .

En effet, cette opération n'affecte pas le langage reconnu par l'AAFAD : soit $t \in L(\mathcal{A})$ un arbre contenant un sous-arbre $s \in L(q)$. Nous avons donc donc $t(\Box \leftarrow s) \in L^{\uparrow}(q)$. Pour cette raison, nous obtenons un arbre $t(\Box \leftarrow r)$ qui satisfait $r \in L^{\downarrow}(q)$. Sachant que $L^{\uparrow}(q) = L^{\uparrow}(q')$, nous déduisons que $t(\Box \leftarrow r)$ reste dans $L(\mathcal{A})$.

**(⇐)** Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD tel que tous ses états sont distincts et $\mathcal{A}' = (Q', \Sigma, Q'_f, \Delta')$ un deuxième AAFAD tel que $|Q| > |Q'|$. Si $L(A) = L(A')$ alors il existe deux états non égaux $q$ et $q'$ de $Q$ tel que : $\exists\, r, s \in T_{\Sigma} : \delta(r) = q, \delta(s) = q'$ et $\delta'(r) = \delta'(s)$ c.-à-d on peut trouver des arbres menant au même état dans $\mathcal{A}'$ parce que $|Q| > |Q'|$. De là, on peut trouver un arbre $t \in L(\mathcal{A})$ de telle sorte que $r \in St(t)$. D'ici, on peut conclure que $t(s \leftarrow r) \notin L(\mathcal{A})$ ($t(s \leftarrow \Box) \in L_p^{\uparrow}$ mais $t(s \leftarrow \Box) \notin L_q^{\uparrow}$ ). Bien que $t$ et $t(s \leftarrow r)$ soient dans $L(\mathcal{A}')$. Ainsi, $L(\mathcal{A}) \neq L(\mathcal{A}')$.

□

**Corollaire 3.2.** *L'automate minimal en nombre d'état qui reconnaît un langage reconnaissable est unique.*

*Démonstration.* Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD, et $\simeq$ la relation d'équivalence définie sur ses états. Soit l'AAFAD $\mathcal{A}_{min} = (Q_{min}, \Sigma, Q_{f\ min}, \Delta_{min})$ défini comme suit :

1. $Q_{min} = \bigcup_{q \in Q} [q]_{\simeq}$,
2. $Q_{f\ min} = [q_f], q_f \in Q_f$
3. $\Delta_{min} = \{(f, [q_1]_{\simeq}, \ldots, [q_n]_{\simeq}, [q]_{\simeq}) \mid (f, q_1, \ldots, q_n, q) \in \Delta\}$

En se référant au Corollaire 3.1, s'il existe deux états équivalents dans $\mathcal{A}$ on peut les fusionner : Soit $q \in Q$ et $\overline{q} = \{q' \in Q \mid q \simeq q'\}$. Alors, selon la preuve du corollaire précédant, on peut remplacer toutes les occurrences de $q \in \overline{q}$ par $q'$ dans $\Delta$ sans affecter le langage reconnu par l'automate. A partir de là, $q$ est représenté par $[q]_{\simeq}$. La réitération de ce processus mènera à un seul automate où tous ses états sont distincts. Par conséquent, la construction de $\mathcal{A}_{min}$ représente l'AAFAD minimal unique.

□

**Exemple 3.1.** *Soit l'exemple suivant* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *avec :*

1. $Q = \{q_1, q_2, q_3, q_4\}$
2. $\Sigma = \{a, b, g(), f(,)\}$
3. $Q_f = \{q_3, q_4\}$
4. $\Delta = \{(a, q_1), (b, q_2), (g, q_1, q_3), (g, q_2, q_4), (f, q_2, q_4, q_4), (f, q_1, q_3, q_4)\}$

*Nous notons par* $L_1, L_2, L_3$ *et* $L_4$ *les langages acceptés par* $q_1, q_2, q_3$ *et* $q_4$*. Les langages* $L_1^{\uparrow}, L_2^{\uparrow}, L_3^{\uparrow}$ *et* $L_4^{\uparrow}$ *dénotent les langages résiduels de* $q_1, q_2, q_3$ *et* $q_4$ *respectivement.*

*Si nous calculons les langages* $L_3$ *et* $L_4$ *qui forment le langage reconnaissable (*$L(\mathcal{A}) = L_3 \cup L_4$*) de* $\mathcal{A}$*, nous aurons :*

$$\begin{aligned} L_3 &= g(L_1) \\ L_4 &= f(L_1, L_3) \cup f(L_2, L_4) \cup g(L_2) \end{aligned}$$

*On peut donc calculer les langages résiduels à partir de ces deux langages en remplaçant les occurrences de chaque langage par le symbole* $\Box$ *dans les langages finaux* $L_3$ *et* $L_4$*. Nous aurons :*

$$\begin{aligned} L_1^{\uparrow} &= f(\Box, L_3) \cup g(\Box) \\ L_2^{\uparrow} &= f(\Box, L_4) \cup g(\Box) \\ L_3^{\uparrow} &= f(L_1, \Box) \\ L_4^{\uparrow} &= f(L_2, \Box) \end{aligned}$$

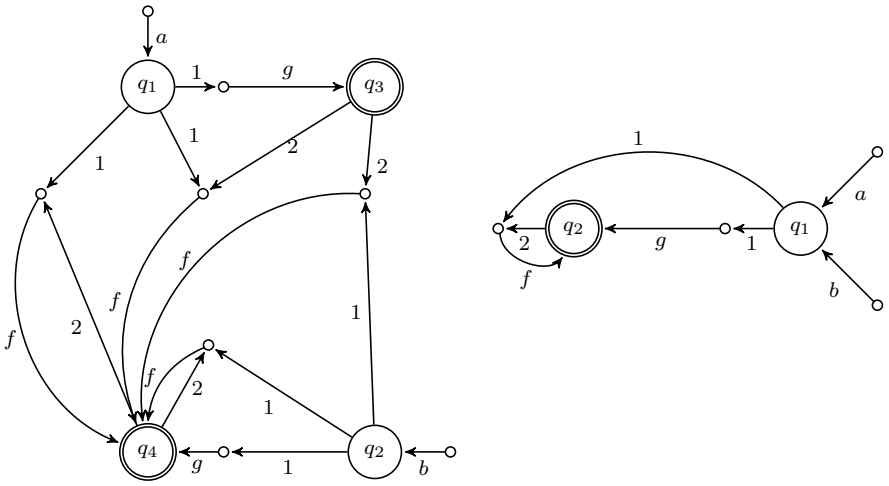
*Sachant que* $\forall q, q' \in Q, L_q^{\uparrow} \neq L_{q'}^{\uparrow} \Leftrightarrow q \not\simeq q'$*, nous déduisons que* $\mathcal{A}$ *est déjà minimal.*

**Exemple 3.2.** *Reprenant le même exemple précédant mais en ajoutant à* $\Delta$ *les transitions suivantes :* $(f, q_1, q_4, q_4)$ *et* $(f, q_2, q_3, q_4)$*. Nous aurons :*

$$\begin{aligned} L_1^{\uparrow} &= f(\Box, L_3) \cup g(\Box) \cup f(\Box, L_4) \\ L_2^{\uparrow} &= f(\Box, L_4) \cup g(\Box) \cup f(\Box, L_3) \\ L_3^{\uparrow} &= f(L_1, \Box) \cup f(L_2, \Box) \\ L_4^{\uparrow} &= f(L_2, \Box) \cup f(L_1, \Box) \end{aligned}$$

*Nous remarquons que* $L_1^{\uparrow} = L_2^{\uparrow}$ *et* $L_3^{\uparrow} = L_4^{\uparrow}$*. Donc* $q_1 \simeq q_2$ *et* $q_3 \simeq q_4$*. Les états équivalents serons fusionnés (voir Figure 3.2). L'automate minimal équivalent est* $\mathcal{A} = (Q_{min}, \Sigma, Q_{fmin}, \Delta_{min})$ *tel que :*

- $Q_{min} = \{q_1, q_2\}$
- $Q_{fmin} = \{q_2\}$
- $\delta_{min} = \{(a, q_1), (b, q_1), (g, q_1, q_2), (f, q_1, q_2, q_2)\}$.




Automate original (exemple 3.1) Version minimale

FIGURE 3.2: Minimisation d'AAFA

### Congruence avec multiplicités

La congruence de Myhill-Nerode peut être généralisée aux séries d'arbres en prenant en compte la dimension multiplicité [Mal09, RBC16, KMW15].

D'abord, nous définissons la congruence de Myhill-Nerode sur les séries d'arbres. Soit $\Sigma$ un alphabet quelconque, $\Phi = (\mathbb{K}, \oplus, \otimes, 0, 1)$ un semi anneau et $\varphi : T_\Sigma \to \mathbb{K}$ une série d'arbres.

**Définition 3.4.** *Une relation d'équivalence* $\equiv_\varphi \subseteq T_\Sigma^2$ *est une congruence si* $\forall f \in \Sigma_n : u_i \equiv_\varphi v_i, 0 \leq i \leq n \Rightarrow \exists k \in \mathbb{K} : \varphi(f(u_1, \ldots, u_n)) = k \otimes \varphi(f(v_1, \ldots, v_n))$.

Rappelons que $i$ est un nombre naturel. Le zéro utilisé dans l'inégalité $0 \leq i$ ne doit pas être confondu avec l'élément absorbant du semi anneau $\Phi$.

Maintenant, le théorème de Myhill-Nerode à multiplicité s'annonce comme suit ([Mal09] :

**Théorème 9.** *Les déclarations suivantes sont équivalentes :*

**(i)** $\varphi : T_\Sigma \to \mathbb{K}$ *est une série d'arbres reconnaissable,*

**(ii)** $\varphi$ *est l'union d'un ensemble donné de classes d'équivalence d'une congruence a indice fini,*

**(iii)** *La relation* $\equiv_\varphi$ *est une congruence à indice fini.*

## 3.2 Minimisation standard

La minimisation d'AAFAD fut proposée pour la première fois par Brainerd et al. [Bra67]. Nous appelons cette technique *la minimisation standard* car elle est la base de pratiquement toutes les méthodes exactes de minimisation.

### 3.2.1 Algorithme standard

L'algorithme standard se base sur une construction d'automate minimal en utilisant une des propriétés des relations d'équivalence : *Le raffinement successif d'ensembles*. En effet, la technique définit une suite de relations d'équivalences partiellement ordonnées dont chacune est plus fine que l'autre. La relation d'équivalence qui forme la borne inférieure de cet ordre est bel et bien l'équivalence de Nerode.

Afin de simplifier l'écriture, nous utilisons $\rho_{q:i}$ , où $\rho = (f, q_1, \ldots, q_n)$ pour abréger $(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)$.

Le raffinement successif des états est réalisé à travers la définition suivante :

**Définition 3.5.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD, nous définissons la relation $\simeq_i, i \in \mathbb{N}$ sur $Q$. Soient $q, q' \in Q$ :*

- $q \simeq_0 q'$ *si et seulement si* $(q \in Q_f \Leftrightarrow q' \in Q_f)$
- $q \simeq_{i+1} q'$ *si et seulement si* $q \simeq_i q'$ *et* $\forall \rho \in \Sigma \times Q^n : \delta(\rho_{q:i}) \simeq_i \delta(\rho_{q':i})$

**Lemme 3.1.** *Il existe un nombre naturel positif $k$ tel que $\simeq_k = \simeq_{|Q|}$.*

*Démonstration.* Premièrement, nous prouvons que le nombre $k$ existe tel que $\forall l > k, \simeq_l = \simeq_k$ :

Soient $k > 0, q, q' \in Q$ tel que $q \simeq_0 q'$. Soit $k > |Q|$, Ici, nous trouvons deux cas : En premier lieu, vérifier que $q \simeq_k q'$ revient à vérifier que $q \simeq_{k-1} q'$. Si $\exists r, s \in P, \rho \in \Sigma \times Q^n : r = \delta(\rho_{q:i}), s = \delta(\rho_{q':i})$ et $r \not\simeq_k s$ Alors $\forall j > k, q \not\simeq_j q'$. En second lieu, si $q$ et $q'$ figurent dans un cycle, alors itérer $k$ fois revient certainement au point de départ car la longueur maximale d'un cycle dans un graphe à $Q$ nœuds est toujours inférieure à $|Q|$. Par ailleurs, la vérification mène à deux états dont l'équivalence est déjà testée. La condition postée est valide. Par conséquent, il existe un nombre $k$ avec lequel $\simeq_k$ est invariant.

Maintenant, il reste à démontrer que $\simeq_k = \simeq$ :

Soient $q, q' \in Q, q \simeq_k q'$, et $t \in L^{\uparrow}(q)$. Alors $\delta(t(q \to \#)) = \delta(t(q' \to \#))$ car $q \simeq_k q'$ implique que si la transition utilisée par $\delta$ est $(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)$ donc $(f, q_1, \ldots, q_{i-1}, q', q_{i+1}, \ldots, q'_n)$ existe et $q_n \simeq_k q'_n$. La réitération de cette vérification mène à deux états $q_f$ et $q'_f$ qui sont tous les deux finaux ($\simeq_k$ est invariant). Par conséquent $t \in L^{\uparrow}(q')$. Finalement, nous déduisons que $L^{\uparrow}(q) = L^{\uparrow}(q')$ et $q \simeq q'$.

□

L'algorithme 3 permet de calculer l'automate minimal d'un AAFAD donné. Le principe est simple : premièrement, commencer par fixer la partition grossière $P = \{Q_f, Q - Q_f\}$ qui coïncide avec $\simeq_1$. Cette partition est ensuite raffinée itérativement par des partitions plus fines jusqu'à arriver enfin à la congruence de Nerode. Néanmoins, Algorithme 3 s'exécute en temps $\mathcal{O}(|\mathcal{A}|^2)$. En effet, cet algorithme n'est pas le code optimal pour réaliser le raffinement successif. Il est à noter qu'une partition intermédiaire qui n'est pas une congruence de Nerode ne peut pas être utilisée pour construire un automate partiellement minimal.

**Algorithme 3** Minimisation d'AAFAD

**Fonction** MINIMISATION(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
  $P_0 \leftarrow Q$ ▷ $P_i$ coïncide avec $\simeq_i$
  $P_1 \leftarrow \{Q_f, Q - Q_f\}$
  **répéter** ▷ raffiner $P_i$ jusqu'à arriver à $\simeq$
    **pour tout** $q, q' \in P_i, \rho \in \Sigma \times Q^n$ **faire**
      **si** $\left(\delta(\rho_{q:i})\ P_i\ \delta(\rho_{q':i})\right)$ **alors**
        $q\ P_{i+1}\ q'$
      **Fin si**
    **Fin pour**
    $i \leftarrow i + 1$
  **jusqu'a** $(P_i = P_{i-1})$
  $Q_{min} \leftarrow \bigcup_{q \in Q}[q]$
  $\Delta_{min} \leftarrow \{(f, [q_1], \ldots, [q_n]) \ / \ (f, q_1, \ldots, q_n) \in \Delta\}$
  $Q_{min\ f} \leftarrow \{[q] \mid q \in Q_f\}$
**retourner** $\mathcal{A}_{min} = (Q_{min}, \Sigma, Q_{min\ f}, \Delta_{min})$
**Fin Fonction**

### 3.2.2 Implémentation de Carrasco et al.

Les automates d'arbres souffrent d'un manque accru d'implémentations comparés à leurs homologues automates de mots, qui possèdent d'énormes boites à outils comme OpenFst [RAJ09]. En effet, mêmes les opérations primitives telles que la composition, la réduction ou bien la minimisation n'ont pas reçu une attention particulière d'implémentations efficaces.

Carrasco et al. [CH09] ont proposé une implémentation de la version standard de la minimisation d'automates d'arbres.

En effet, leur technique repose sur deux idées principales : améliorer l'initialisation de l'algorithme standard et puis définir une technique rapide de recherche d'arguments utiles depuis les transitions de l'automate en question.

#### 3.2.2.1 Améliorer l'initialisation

Cette idée est simple mais efficace. Il s'agit de remplacer la partition la plus grossière $P_0$ qui coïncide avec la relation $\simeq_0$ par une partition plus fine utilisant la notion de *signature*

**Définition 3.6.** *La signature $sig(q)$ d'un état $q$ est définie comme suit :*

$$sig(q) = \{(f, k) \mid \exists (f, q_1, \ldots, q_n) \in \Delta, q_k = q, 1 \leq k < n\} \cup \begin{cases} \{(\#, 1)\} & \textit{si } q \in Q_f \\ \emptyset & \textit{sinon} \end{cases}$$

Nous remarquons que si l'automate à minimiser est complet, tous les états finaux (non finaux) ont les mêmes signatures. Cela est dû à la définition de la complétude qui exige une sortie pour chaque argument qui utilise une lettre de l'alphabet et un ensemble d'états. Pour cette raison, nous redéfinissons

la relation $\simeq_i$ (pour un entier $i$) pour qu'elle soit compatible à la fois avec les automates incomplets, et les automates complets.

**Lemme 3.2.** *La relation $\simeq_n$ peut être définie comme suit :*

1. $p \simeq_0 q$ *si et seulement si* $(p \in Q_f \Leftrightarrow q \in Q_f)$
2. $p \simeq_{j+1} q$ *si et seulement si* $p \simeq_j q$ *et pour tout* $\rho \in \Gamma(p), i \in occ_p(\rho) : \rho_{:_i p} \in \Gamma(q)$ *et* $\delta(\rho) \simeq_j \delta(\rho_{:_i q})$

La preuve du lemme est évidente. Si l'automate n'est pas complet, nous ajoutons *l'état puits* $\perp$ aux arguments qui n'ont pas de sortie : s'il existe des arguments $(f, q_1, \ldots, q_n), f \in \Sigma_n, q_1, \ldots, q_n \in Q$ tel que $\delta((f, q_1, \ldots, q_n))$ n'est pas définie alors nous ajoutons la transition $(f, q_1, \ldots, q_n, \perp)$ à $\Delta$. En effet, cette solution de complétion de transitions est maintenue par beaucoup de références bibliographiques telles que [CDF07, CDG+07].

**Corollaire 3.3.** *Soient deux états* $p, q \in Q$. *Nous avons* $\mathrm{sig(p)} \neq \mathrm{sig}(q) \Rightarrow p \not\simeq q$.

En utilisant le corollaire précédent, on peut initialiser $P_0$ par :

$$P_0 = \{R \mid \forall q, q' \in R, sig(q) = sig(q')\}$$

En effet, la signature d'états à un bon impact sur la rapidité du calcul de l'automate minimal. Un bon nombre d'opérations peut être évité en initialisant la première partition moyennant le concept de $sig$.

**Exemple 3.3.** *Prenons l'automate illustré par la Figure 3.2. Nous avons :*

- $\mathrm{sig}(q_1) = \mathrm{sig}(q_2) = \{(f, 1), (g, 1)\}$
- $\mathrm{sig}(q_3) = \mathrm{sig}(q_4) = \{(f, 2), (\#, 1)\}$

*Nous remarquons que* $\mathrm{sig}(q_1) \neq \mathrm{sig}(q_1)$. *Nous déduisons que* $q_1 \not\simeq q_3$.

#### 3.2.2.2 Accélérer la recherche d'arguments

Dans leur implémentation, Carrasco et al. ont proposé une méthode rapide est efficace pour calculer les raffinements successifs. En effet, l'instruction 6 de l'Algorithme 3 requière l'accès aux différents arguments des transitions. Carrasco et al. maintiennent une structure de file d'attente $K$. Elle garde la trace de chaque partition calculée $P_i$ pendant l'itération $i$. Pour cette raison, l'algorithme 4 utilise une fonction $next_i(q), q \in Q$ qui renvoie en retour l'état suivant $q$ dans $[q]_i$ où $[q]_{\simeq_i}$. Les auteurs supposent que les états sont ordonnés au préalable. $next_i^{[j]}(q)$ applique $next_i(q)$ $j$ fois.

Afin d'éviter les opérations de recherche inutiles, une propriété importante est utilisée :

Soient $p, q \in Q$ tel que $p \simeq_0 q$, alors si $p \not\simeq_{n+1} q$ alors il existe $(f, q_1, \ldots, p, \ldots, q_m, r) \in \Sigma \times Q^{m+1}$ tel que $\delta(f, q_1, \ldots, q, \ldots, q_m) \not\simeq_n r$. Nous déduisons donc que $j > 0$ tel que : $\delta(f, q_1, \ldots, next_n^{[j]}(p), \ldots, q_m) \equiv_n r$ et $\delta(f, q_1, \ldots, next_n^{[j+1]}(p), \ldots, q_m) \not\simeq_n r$ (on suppose que $r \neq \perp$).

Par conséquent, une transition $(f, q_1, \ldots, q_{\hat{f}}, q)$ est comparée seulement avec $\hat{f}$ transitions de la forme $(f, q_1, \ldots, next_i(q_k), \ldots, q_{\hat{f}}, q)$. Nous notons que cette propriété est utilisée pour accélérer le

raffinement des partitions. La complexité pire-cas reste inchangée (quadratique). Néanmoins, les auteurs ont montré sur un échantillon de 20000 AAFAD que le temps empirique est meilleur [CDF07].

Malheureusement, cette étude expérimentale ne prouve pas l'efficacité de l'algorithme car l'absence d'un générateur aléatoire implémenté d'automates d'arbres rend difficile la phase expérimentale. Les auteurs ont utilisé un ensemble d'arbres comme étant des automates acycliques. Mais vu que la complexité de la minimisation vient de la présence des cycles, les résultats ne sont pas très représentatifs.

1: **Fonction** MINIMISATION(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
2: $P_0 \leftarrow \{C_k : \forall p, q \in C_k : sig(p) = sig(q)\}$ ▷ $\forall p \in C_k : [p]_1 = C_k$
3: $i \leftarrow 1$
4: Enfiler les premiers états de $C_k \in P_0$ dans $K$
5: **tanque** $K \neq \emptyset$ **faire**
6: Défiler un élément $q$ depuis $K$
7: **pour tout** $(f, q_1, \ldots, q_n, q') \in \Delta$ tel que $q \equiv_i q'$ et pour tout $k \leq n$ **faire**
8: **si** $\delta(f, q_1, \ldots, next_i(q_k), \ldots, q_n) \not\equiv q'$ **alors**
9: Créer $P_{i+1}$ depuis $P_i$ en fragmentant $[q_k]_i$ en un ensemble de sous-ensembles $[\delta(f, q_1, \ldots, q'_k, \ldots, q_n)]_i$
10: Enfiler le premier état de chaque classe dans $K$
11: **Fin si**
12: **Fin pour**
13: $i \leftarrow i + 1$
14: **Fin tanque**
15: $Q_{min} \leftarrow \bigcup_{q \in Q}[q]$
16: $\Delta_{min} \leftarrow \{(f, [q_1], \ldots, [q_n]) \ / \ (f, q_1, \ldots, q_n) \in \Delta\}$
17: $Q_{min\ f} \leftarrow \{[q] \mid q \in Q_f\}$
18: **retourner** $\mathcal{A}_{min} = (Q_{min}, \Sigma, Q_{min\ f}, \Delta_{min})$
19: **Fin Fonction**

## 3.3 Minimisation incrémentale

Dans le cas des mots, Watson [Wat01] a introduit pour la première fois la notion de minimisation incrémentale d'automates. Après, Watson et al. [WD03] ont amélioré la première version dont la complexité était exponentielle. Enfin, Almeida et al. [AMR10] ont présenté l'algorithme le plus efficace pour la minimisation incrémentale dont la complexité est d'ordre quadratique. Cet algorithme est basé sur la méthode UNION-FIND [GF64].

Au lieu de démarrer l'algorithme de minimisation avec deux partitions $Q_f$ et $Q - Q_f$, Watson et al. [WD03] ont défini une nouvelle approche de minimisation pour les automates de mots. En effet, le calcul commence à partir d'une partition de singletons dans laquelle chaque état représente une classe d'équivalence. Le processus est alors inversé : au lieu d'utiliser le raffinement des partitions, une étape de fusion de partitions jugée équivalentes est appliquée à chaque itération. L'avantage de cette approche réside dans la possibilité d'interrompre le processus à tout moment engendrant ainsi un automate valide (qui reconnaît le même langage et avec un nombre réduit d'états mais pas forcément minimal). Néanmoins, l'algorithme assure l'atteinte de l'automate minimal au terme du calcul.

Plus tard, Cleophas et al. [Cle08, CH09] ont généralisé cette approche pour les automates d'arbres en adaptant la fonction de fusion d'états équivalents aux arbres.

Cette technique peut être d'une extrême importance si les automates à réduire sont volumineux et que le temps alloué à la minimisation est critique. Ainsi l'application de l'algorithme incrémental permet d'atteindre un automate avec un nombre d'état réduit en un temps métrisable selon les conditions de l'utilisateur.

Les auteurs ont proposé la fonction récursive suivante :

**Définition 3.7.** *Soient $p, q \in Q$. Alors*

$$equiv(p,q) = \bigwedge_{\rho \in \Sigma \times Q^n} (equiv(\delta(\rho_{p:i}), \delta(\rho_{q:i}))) \wedge ((p \in Q_f) \Leftrightarrow (q \in Q_f))$$

Nous aurons :

**Lemme 3.3.** *Soient $p, q$ deux états, alors :*

$$equiv(p,q) \Leftrightarrow (p \simeq q)$$

*Démonstration.* En remplaçant $\simeq$ par $\simeq_{|Q|}$ et en utilisant la définition 3.5 le lemme est vrai. □

L'algorithme 5 implémente la fonction récursive $equiv$. Il utilise une variable globale $S$ pour sauvegarder les états susceptible d'être équivalents. A la fin de chaque itération, les états vérifiés sont retirés de $S$.

Dans les automates où des cycles sont présents, les appels récursifs peuvent engendrer des boucles infinies. Pour cette raison, la fonction utilise un paramètre $k$ comme étant un *compteur* de cycles. Comme la longueur d'un cycle ne dépasse pas $|Q| - 2$, le compteur $k$ est initialisé par cette valeur lors du premier appel de la fonction.

**Algorithme 5** Fonction d'équivalence d'états

```
1:  Fonction equiv(p, q, k)
2:      si k = 0 alors
3:          eq ← ((p ∈ Q_f) ⇔ (q ∈ Q_f))
4:      sinon si {p, q} ∈ S alors
5:          eq ← true
6:      sinon
7:          eq ← ((p ∈ Q_f) ⇔ (q ∈ Q_f))
8:          S ← S ∪ {{p, q}}
9:          pour tout ρ ∈ Σ × Q^n faire
10:             eq ← eq ∧ equiv(δ(ρ_{p:i}), δ(ρ_{q:i}), k − 1)
11:         Fin pour
12:         S ← S − {{p, q}}
13:
14:     Fin si
15: retourner (eq)
16: Fin Fonction
```

Néanmoins, la technique incrémentale pour les mots souffre de sa complexité élevée qui est de l'ordre de $\mathcal{O}(|\mathcal{A}|^{|Q|-2}|Q|^2)$.

## 3.4 Hyper-minimisation

L'hyper-minimisation est une nouvelle technique présentée par Holzer et al. [HM09] pour les automates de mots et généralisée après par Jez et al. [JM12] pour les AAFAD. Cette technique trouve un automate plus petit que la version minimal d'un AAFAD qui accepte un langage similaire avec des différences finies en termes d'arbres reconnus. L'avantage de cette technique réside dans la taille réduite de l'automate résultant comparée avec celle de l'automate minimal exacte.

**Définition 3.8** (États presque équivalents, états noyaux, états préliminaires)**.**

- *Deux états $q, q' \in Q$ sont presque équivalents ($q \doteq q'$) si et seulement si $(L^{\uparrow}(q) - L^{\uparrow}(q')) \cup (L^{\uparrow}(q') - L^{\uparrow}(q))$ est fini.*
- *Deux AAFAD $\mathcal{A}$ et $\mathcal{B}$ sont presque équivalents si et seulement si $(L(\mathcal{A}) - L(\mathcal{B})) \cup (L(\mathcal{B}) - L(\mathcal{A}))$ est fini.*
- *Un état $q$ est un état noyau ssi $(L_q^{\uparrow})$ est fini. Sinon, $q$ est un état préliminaire.*

Dans cette section, nous nous limitons à quelques résultats utilisés dans l'hyper-minimisation, les preuves sont disponibles dans [JM12].

Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD, $p, q \in Q$ :

1. Si $p \doteq q$, alors $\mathcal{A}$ est l'AAFAD obtenu en fusionnant $p$ et $q$ qui sont presque équivalents.
2. Un AAFAD est hyper-minimal si et seulement si chaque paire des états presque équivalents est une paire d'états noyaux.

L'algorithme 6 donne une description générale du processus de l'hyper-minimisation.

Premièrement, Algorithme 6 exécute la fonction $minimize(\mathcal{A})$ afin de retrouver l'automate minimal de $\mathcal{A}$. Ensuite, les deux ensembles $P$ et $K$ qui représentent respectivement les états préliminaires et les états noyaux sont définis. Après, l'ensemble des états presque équivalents $E$ est calculé et mémorisé sous la forme d'un ensemble partitionné. Évidemment, chaque partition contient les états d'une même classe d'équivalence. Néanmoins, La définition de $E$ n'est pas une tâche facile. Pour cette raison, au lieu de calculer la différence symétrique entre deux états, le calcul d'un produit spécifique appelé l'auto-produit $\mathcal{A}_{\otimes}$ de l'automate $\mathcal{A}$ peut être utile pour la détermination des paires d'états presque équivalents. L'auto-produit peut être calculé comme suit :

**Définition 3.9.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$. Alors l'auto-produit de $\mathcal{A}$ est $\mathcal{A}_{\otimes} = (P, \Sigma, P_f, \Delta')$ où*

$$\begin{aligned} P &= Q \cup \{\bot\} \cup (Q \cup \{\bot\})^2 \\ P_f &= \{\langle p, q\rangle \mid p \in Q_f \text{ ou bien } q \in Q_f\} \\ \Delta' &= \Delta \cup \Delta_c \end{aligned}$$

**Algorithme 6** Hyper minimisation

**Fonction** MAIN(Un AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
  $\mathcal{A}_{min} \leftarrow minimize(\mathcal{A})$
  $K \leftarrow compute\_kernel(\mathcal{A})$
  $E \leftarrow \Phi\{Q\}$ tel que $q$ et $q'$ son dans la même partition si et seulement si $(q, q') \in \mathcal{A}_{\otimes}$ : $L^{\uparrow}_{(q,q')}$ est fini. ▷ $\Phi(Q)$ est un partitionnement de $Q$.
  **pour tout** $e \in E$ **faire**
    $P_e \leftarrow e \cap P$
    $K_e \leftarrow e \cap K$
    **si** $K_e \neq \emptyset$ **alors**
      **pour tout** $q \in P_e$ **faire**
        Fusionner $q$ dans $K_e$
      **Fin pour**
    **sinon**
      **pour tout** $q \in P_e$ **faire**
        Fusioner $q$ dans $P_e$
      **Fin pour**
    **Fin si**
  **Fin pour**
**Fin Fonction**

*et pour tout* $f \in \Sigma_n, p, q, q_1, \ldots, q_n \in P$ *:*

$$(f, q_1, \ldots, q_{i-1}, \langle p, q\rangle, q_{i+1}, \ldots, q_n, \langle \delta(f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n, ), \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n, )\rangle) \in \Delta_c$$

Enfin, une fusion des états satisfaisant les propriétés de l'hyper minimisation est effectué.

## 3.5 Construction incrémentale de l'AAFAD minimal

La construction incrémentale d'un AAFAD est une approche intéressante qui consiste à construire un automate à partir d'exemples (arbres) positifs. Un exemple concret est la construction d'une structure compacte et finie pour représenter un grand (voir infini) fichier d'index sous forme arborescente. En effet, cette technique permet d'ajouter ou de supprimer un mot (arbre) depuis un automate déjà minimal. En d'autres termes, si $\mathcal{A}$ est un automate de mots (resp. un AAFAD) minimal et $w$ un mot (resp. $t$ un arbre) alors la construction incrémentale consiste a créer un nouvel automate minimal reconnaissant ainsi $L(\mathcal{A}) \cup \{w\}$ (resp. $L(\mathcal{A}) \cup \{t\}$).

La première version de cette approche pour les automates de mots acycliques a été présentée par Daciuk et al. [DMWW00] . Ensuite, Carrasco et al. [CF02] ont généralisé l'algorithme pour les automates e mots cycliques. Enfin, les mêmes auteurs ont proposé une version pour les AAFAD dans [CDF09].

L'idée naturelle d'une telle construction est de créer premièrement un AAFAD acyclique qui reconnaît seulement l'arbre $t$ à ajouter. Ensuite, un nouvel automate est créé via l'opération d'union de l'automate de départ avec l'automate acyclique construit. La dernière étape est donc de minimiser le

nouvel automate. Néanmoins, à chaque ajout, des opérations sont répétées et d'autres sont inutiles. En effet, il n'est pas nécessaire de retraiter des paires d'états qui ne sont pas concernées par la reconnaissance de l'arbre à ajouter. Une autre lacune de cette méthode est la suppression d'un arbre : la méthode constructive permet l'ajout et la suppression en même temps, chose qui ne peut pas être faite via cette méthode.

Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA minimal, $t \notin L(\mathcal{A})$ est l'arbre à ajouter. Pour y remédier, une construction incrémentale suppose que l'automate à augmenter soit déjà minimal. Nous définissons $C_q$ pour un état $q$ comme étant le nombre de transitions produisant $q$. La première étape consiste à essayer de reconnaître $t$ par $\mathcal{A}$. Si $m_{\mathcal{A}}(t) \in Q_f$ alors $\mathcal{A}$ est inchangé. Sinon, les états concernés par la reconnaissance sont identifiés afin d'effectuer les changements nécessaires. La construction incrémentale proposée par Carrasco et al. se résume en ce qui suit.

1. pour chaque arbre $s \in St(t)$ :
   - Si la sortie de $\delta(s)$ n'est pas définie alors on peut ajouter sans risque un état $n$ tel que $L^{\downarrow}(n) = s$.
   - s'il existe un état $q \in Q$ tel que $\delta(s) = q$ on rencontre deux cas :
     (a) si $C_q$=1 alors aucun changement est effectué.
     (b) sinon, ($C_q > 1$) un état *clone* $n$ est ajouté à $Q$ et $\Delta$ est modifié en sorte que $n \simeq q$. Les algorithmes 7 et 8 expliquent le fonctionnement du processus d'ajout.
2. Exécuter une minimisation *locale* touchant juste les états participant dans les étapes précédentes.

**Algorithme 7** Fonction split

**Fonction** split(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, arbre $f(s_1, \ldots, s_m)$, ensemble d'états $\Theta$ )
  **pour tout** $k = 1 \ldots$ *jusqu'a* $m$ **faire**
    $\mathcal{A} \leftarrow \text{split}(\mathcal{A}, \Theta, s_k)$
    $r_k \leftarrow \delta(s_k)$
  **Fin pour**
  $q \leftarrow \delta(f, r_1, \ldots, r_m)$
  **si** $C_q \neq 1$ **alors**
    $\text{create}(n)$
    $Q \leftarrow Q \cup \{n\}$
    **si** $C_q = 0$ **alors**
      $\Delta \leftarrow \Delta \cup \{(f, r_1, \ldots, r_m, n)\}$
    **sinon**
      $\mathcal{A} \leftarrow \text{Clone}(\mathcal{A}, q, n)$
      **pour tout** $q, (f, r_1, \ldots, r_m, q) \in \Delta$ **faire**
        $\text{Replace}(\Delta, (f, r_1, \ldots, r_m, q), (f, r_1, \ldots, r_m, n))$
      **Fin pour**
    **Fin si**
    $\Theta \leftarrow \Theta \cup \{r_1, \ldots, r_n\}$
  **Fin si**
**Fin Fonction**

**Algorithme 8** Fonction clone

1: **Fonction** clone(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, état $q$, état $n$)
2: **si** $q \in F$ **alors**
3: $Add(Q_f, n)$
4: **Fin si**
5: **tanque** il existe $m > 0, k \leq m, q_1, \ldots, q_m \in Q$ tel que $(f, q_1, \ldots, q_{k-1}, q, q_{k+1}, \ldots, q_{m-1}, q_m) \in \Delta$ **faire**
6: $\Delta \leftarrow \Delta \cup \{(f, q_1, \ldots, q_{k-1}, n, q_{k+1}, \ldots, q_{m-1}, q_m)\}$
7: **Fin tanque**
8: **Fin Fonction**

## 3.6 Conclusion

Dans ce chapitre, nous avons étudié la notion de minimisation des différents types d'automates d'arbres. Pour cette raison, nous avons donné une classification des techniques existantes selon le type de l'automate ou la nature du résultat. Pour ce dernier, la classe exacte regroupe les techniques qui aboutissent à l'automate minimal unique. La classe approchée quant à elle contient les méthodes qui produisent un automate avec moins d'états que l'original mais la minimalité ou le respect du langage reconnu n'est pas garantie.

Dans le prochain chapitre, nous décrivons notre contribution et nous montrons les améliorations que nous pouvons atteindre vis à vis des techniques existantes.

# Chapitre 4

# Nouvelles méthodes de minimisation

DAns ce chapitre, nous décrivons notre première contribution. En effet, nous proposons deux nouvelles méthodes de minimisation d'automates d'arbres. Ces propositions reposent sur deux projections de l'automate en question. Elles peuvent être vues comme *méta-méthodes* car elles s'appliquent aux différentes techniques de minimisation exactes et approchées.

La première vision est une vision graphique. L'automate est vu comme un graphe biparti. Des propriétés sont tirées directement de la théorie des graphes comme le chemin entre deux nœuds, les niveaux hiérarchiques d'un graphe, les voisins d'un nœud ou bien la notion des graphes fortement connexes. Ces propriétés sont ensuite appliquées sur différentes techniques de minimisation afin d'améliorer la complexité de calcul et/ou de stockage.

Notre deuxième méthode consiste premièrement à transformer l'automate à minimiser en un automate de mots. Ensuite, nous montrons que la version minimale de l'automate de mots associé coïncide avec l'automate d'arbre minimal.

Cette transformation est bénéfique sur deux plans. Le premier apport consiste à améliorer la complexité de plusieurs techniques comme la minimisation acyclique et incrémentale. Le deuxième intérêt permet d'éviter la ré-implémentation de toutes les techniques de minimisation et d'utiliser juste la large gamme d'outils et algorithmes déjà présents et qui ont fait leurs preuves d'efficacité pour les mots.

Nous avons appelé ces deux méthodes de minimisation d'automates d'arbres *méthode directe* et *méthode indirecte* vu que la deuxième passe par les automates de mots afin de minimiser ceux des arbres.

Nos deux méthodes sont illustrées dans la classification des techniques de minimisation proposée dans l'état de l'art (Figure 4.1).

## 4.1 Méthode directe

Dans cette section, nous proposons une construction d'un graphe spécial et nous introduisons quelques propriétés importantes afin de retrouver l'automate minimal en un temps linéarithmique.

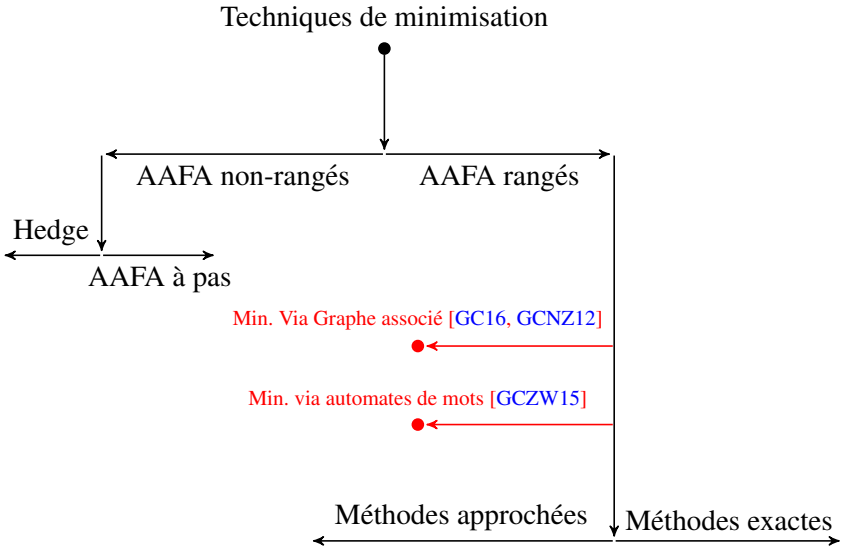



FIGURE 4.1: Classification des techniques de minimisation avec les deux méta-méthodes proposées

Premièrement, nous présentons la structure graphique qui permet de garantir l'accès rapide aux arguments et ainsi vérifier les équivalences en un temps optimisé.

En effet, un AAFA peut être vu comme un graphe "biparti" selon la définition suivante.

**Définition 4.1.** *Le graphe $G_{\mathcal{A}} = (X, lab, A, \gamma)$ associé à l'AAFA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ est défini comme suit :*

- *L'ensemble des $\Sigma$-nœuds $X_\Sigma$ est un ensemble assurant une bijection $X_\Sigma \to \Delta$.*
- *$X = X_\Sigma \cup Q$ est l'ensemble des nœuds. $Q$ est l'ensemble des États-nœud (chaque état est un nœud dans le graphe).*
- *$lab : X \to \Sigma \cup Q$ est la fonction d'étiquetage des nœuds.*
- *$A \subseteq (X_\Sigma \times Q) \cup (Q \times X_\Sigma)$ est l'ensemble des arcs.*
- *$\gamma : A \to \mathbb{N}$ est la fonction des poids.*

*$G_{\mathcal{A}}$ est construit comme suit : avec chaque transition $(\sigma, q_1, \ldots, q_n, q)$, nous associons l'ensemble d'arcs $\{(q, x_\sigma)\} \cup \{(x_\sigma, q_i) \mid 0 \leq i \leq n\}$, tel que :*

1. $lab(x_\sigma) = \sigma$.
2. $\forall q_i, lab(q_i) = q_i, lab(q) = q$.
3. $\gamma(q, x_\sigma) = 0$.
4. $\gamma(x_\sigma, q_i) = i, 1 \leq i \leq n$.

**Exemple 4.1.** *Soit l'automate $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ où*

- $Q = \{q_1, q_2, q_3, q_4\}$
- $\Sigma = \{a, b, g(), f(,)\}$
- $Q_f = \{q_3, q_4\}$
- $\Delta = \{(a, q_1), (b, q_2), (g, q_1, q_3), (f, q_1, q_3, q_4), (f, q_1, q_4, q_4), (g, q_2, q_4), (f, q_2, q_3, q_4), (f, q_2, q_4, q_4)\}$

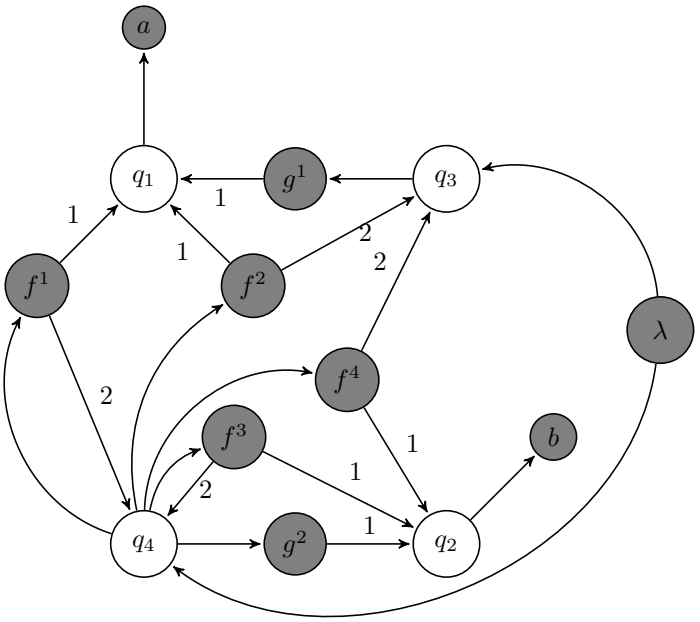



FIGURE 4.2: Graphe associé à l'automate de l'Exemple 4.1

*L'ensemble des* $\Sigma$ *nœuds est* $X_\Sigma = \{a, b, g^1, g^2, f^1, f^2, f^3, f^4\}$. *En effet, le numéro utilisé en exposant nous permet de distinguer deux transitions ayant le même symbole de* $\Sigma$.

*L'ensemble des arcs* $A = \{(a, q_1), (b, q_2), (q_3, g^1), (g^1, q_1), (q_4, f^1), (f^1, q_1), (f^1, q_3),$
$(q_4, f^2), (f^2, q_1), (f^2, q_4), (q_4, f^3), (f^3, q_2), (f^3, q_3), (q_4, f^4), (f^4, q_2), (f^4, q_4), (q_4, g^2), (g^2, q_2)\}$.

*Comme exemple de la fonction d'étiquetage et de poids, nous avons :*

- $lab(f^1) = lab(f^2) = lab(f^3) = lab(f^4) = f, lab(g^1) = lab(g^2) = g, lab(a) = a, lab(b) = b$.
- $\gamma(q_4, f^2) = 0$ *mais* $\gamma(f^2, q_4) = 2$.

Rappelons qu'à chaque transition, un nouveau $\Sigma$-nœud est utilisé. Nous avons donc $|X_\Sigma| = |\Delta|$. En effet, les $\Sigma$-nœuds sont étiquetés avec le premier élément de la transition avec laquelle l'ensemble d'arcs associé est construit. Par conséquent, deux $\Sigma$-nœuds peuvent avoir la même étiquette.

En effet, ce graphe est composé de sommets et d'arcs orientés. L'ensemble des sommets inclue un sommet spécial étiqueté $\lambda$ qui sert de point d'entrée au graphe. Les sommets étiquetés $\Sigma$ ($\Sigma$-nœuds) représentent les premiers symboles des transitions. Quant aux sommets étiquetés $Q$, ils représentent les occurrences des états dans les transitions.

L'algorithme 9 donne le processus de construction du graphe associé à un AAFAD $\mathcal{A}$. La fonction $CREATE$(node $u$, élément $e \in Q \cup \Sigma$, Intégrer $i$) crée un sommet $v_e$ étiqueté $e$ et un arc $(u, v_e)$ tel que $\gamma((u, v_e)) = i$. Il est à noter que cet algorithme parcourt chaque transition une seule fois. Par conséquent, la complexité est linéaire.

La Figure 4.2 illustre le graphe associé à l'automate présenté dans l'Exemple 4.1.

**Algorithme 9** Construction du graphe associé

Créer un sommet source $\lambda$
**pour tout** $q_f \in Q_f$ **faire**
    $create(\lambda, q_f, 0)$
    $Treat \leftarrow Treat \cup v_{q_f}; Marked \leftarrow \emptyset$
**Fin pour**
**répéter**
    **pour tout** $v \in Treat$ **faire**
        **pour tout** $\tau = f(q_1, \ldots, q_n) \in F_{lab(v)}$ **faire**
            $CREATE(v, f, 0)$
            **pour** $r = q_1$ to $q_n$ **faire**
                **si** $q_i \notin Marked$ **alors**
                    $create(v, q_i, i)$
                    $Marked \leftarrow Marked \cup \{q_i\}$
                **sinon**
                    $create(v, q_i, 0)$
                **Fin si**
                $Temp \leftarrow Temp \cup q_i$
            **Fin pour**
        **Fin pour**
    **Fin pour**
    $Treat \leftarrow Temp$
    $Temp \leftarrow \emptyset$
**jusqu'a** Tous les états sont traités

### 4.1.1 Initialisation efficace

Deux cas triviaux peuvent se manifester lors de la minimisation. En effet, on peut rencontrer seulement une classe d'équivalence, si tous les états finaux partagent le même langage résiduel. Par ailleurs, le nombre des classes d'équivalence peut atteindre au maximum le nombre d'états, si l'automate en question est déjà minimal.

Néanmoins, ce nombre de classes peut être majoré au début du processus de minimisation afin de permettre une amélioration du temps de calcul. Nous introduisons quelques propriétés nécessaires mais pas suffisantes afin de décider si deux états sont susceptibles d'être équivalents.

Rappelons que la signature, définie précédemment, détecte la similarité des occurrences d'états dans l'ensemble des transitions. Nous proposons une vérification de similarité "verticale" basé sur ce qu'on va appeler *niveaux des états*.

**Définition 4.2.** *(Chemin d'états) L'ensemble des* chemins *pour un état $q \in Q$, dénoté $\hat{q}$, est l'ensemble de toutes les séquences finies $q_1, \ldots, q_n, q$ tel que $q_1 \in Q_f, q_2, \ldots, q_n \in Q$, $n \leq |Q|$ et pour chaque $(q_{i-1}, q_i)$ il existe un $\Sigma$-nœud $x$ tel que $(q_{i-1}, x), (x, q_i) \in A$.*

En d'autres termes, seuls les arcs avec un poids non nul sont considérés dans le calcul des chemins entre l'état en question et un état final.

**Définition 4.3.** *Soit $q \in Q$ un état. Alors le* niveau *de $q$ est $\mathcal{L}(q) = \min_{s \in \hat{q}} \|s\|$. $\|s\|$ est la longueur du chemin $s$.*

Nous avons :

**Lemme 4.1.** *L'ensemble des niveaux d'états pour un AAFAD peut être calculé en un temps linéaire*

Évidemment, l'ensemble des niveaux d'états peut être calculé "à la volée" en exécutant un parcours en largeur à partir du nœud $\lambda$ du graphe associé.

Par conséquent, la propriété suivante est vraie.

**Lemme 4.2.** *Soient* $p, q \in Q$. *Alors,* $p \simeq q \Rightarrow (sig(p) = sig(q) \wedge \mathcal{L}(p) = \mathcal{L}(q))$.

*Démonstration.* Pour démontrer ce lemme, il suffit de prouver que $\mathcal{L}(p) \neq \mathcal{L}(q) \Rightarrow p \not\simeq q$ (voir [CDF07] pour les signatures).

Soient $p, q \in Q$.

Supposons que $\mathcal{L}(p) \neq \mathcal{L}(q)$. Alors, le chemin minimal entre $p$ et un état final diffère du chemin minimal entre $q$ et un autre état final. Nous trouvons $t_1 \in L_p^{\uparrow}, t_2 \in L_q^{\uparrow}$ tel qu'il existe deux branches d'arbres pris respectivement de $t_1$ et de $t_2$ avec différentes longueur. Nous concluons que $L_p^{\uparrow} \neq L_q^{\uparrow}$ et donc $p \not\simeq q$. □

**Exemple 4.2.** *Prenons l'automate illustré par Figure 4.2. Nous avons :*

- $\hat{q_1} = \{(q_3, q_1), (q_4, q_3, q_1), (q_4, q_4, q_1), (q_4; q_4, q_3, q_1), (q_4, q_2, q_3, q_1)\}$
- $\mathcal{L}(q_1) = 2$
- *De la même manière, nous avons* $\mathcal{L}(q_3) = \mathcal{L}(q_4) = 1$ *et* $\mathcal{L}(q_2) = 2$
- *En tenant en compte que* $q_1 \simeq q_2$ *et* $q_3 \simeq q_4$ *nous remarquons que les niveaux des états équivalents sont égaux.*

### 4.1.2 Application à la minimisation standard

Moyennant le graphe ainsi défini, on peut voir que les propriétés de la relation de Myhill-Nerode peuvent être facilement extraites et vérifiés [GC16].

Nous définissons quelques propriétés afin de simplifier la prédiction de l'équivalence entre états.

**Définition 4.4.** *Soit* $q \in Q$ *un état tel que* $(x, q) \in A$. *Alors, l'ensemble des* voisins *de* $q$ *par rapport à* $x \in X_\Sigma$ *est* $N_x(q) = \{(p, \gamma(x, p)) \mid (x, p) \in A, p \neq q\}$.

Nous avons :

**Proposition 4.1.** *Soient* $p, q \in Q$. *Alors,* $p \simeq q \Rightarrow \big(\forall x \in X_\Sigma, \exists y \in X_\Sigma : lab(x) = lab(y) \wedge N_x(p) = N_y(q)\big)$

*Démonstration.* Soient $p, q \in Q$ tel que $\exists x, y \in X_\Sigma : lab(x) = lab(y) \wedge N_x(p) \neq N_y(q)$. Alors, en utilisant Définition 4.4 : il existe $(p', \gamma(x, p'))$ dans $N_x(p)$ qui ne figure pas dans $N_y(q)$. Par conséquent, la transition $(lab(x), q_1, \ldots, q_n) \in \Delta$ tel que $p, q_{\gamma(x,p')} \in q_1 \ldots, q_n$ ne satisfait pas l'équivalence de Myhill-Nerode. □

En raison de simplification, Nous mettons $p \doteq q \Leftrightarrow \forall x \in X_\Sigma, \exists y \in X_\Sigma : lab(x) = lab(y) \wedge N_x(p) = N_y(q)$.

**Exemple 4.3.** *En se référant à l'automate présenté par Figure 4.2.*

$$\begin{aligned} N_{f^1}(q_1) &= \{(q_3, 2)\},\ N_{f^2}(q_1) = \{(q_4, 2)\}, N_{g^1}(q_1) = \emptyset \\ N_{f^3}(q_2) &= \{(q_3, 2)\},\ N_{f^3}(q_4) = \{(q_4, 2)\}, N_{g^2}(q_2) = \emptyset \end{aligned}$$

*Nous concluons que* $q_1 \doteq q_3$.

#### 4.1.2.1 Un algorithme linéarithmique

Dans cette partie, nous adaptons la stratégie de "l'analyse de la plus petite partition" invoquée par Hopcroft dans le cas des mots ([Hop71]) aux arbres. Effectivement, cette stratégie résout efficacement le problème de raffinement des partitions. Elle consiste à utiliser un *diviseur* de partitions suivant un critère bien défini. Ce diviseur assure que deux états présents dans deux nouvelles partitions sont nécessairement non équivalents. L'étape suivante est de choisir la plus petite partition pour la prochaine itération. Le processus s'arrête quand un point fixe est atteint.

Nous définissons la notion de *prédécesseur* et de *successeur* d'un nœud :

**Définition 4.5.** *Soit* $x \in X_\Sigma$*, alors le prédécesseur* $pred(x)$ *de* $x$ *est* $pred(x) = q$ *tel que* $(q, x) \in A$*. L'ensemble des successeurs* $succ(x)$ *de* $x$ *est* $succ(x) = \{(q, \gamma(x, q)) \mid (x, q) \in A\}$.

Nous rappelons que le prédécesseur d'un nœud représente la sortie d'une seule transition. En d'autres termes, ce prédécesseur est unique (déterminisme).

Nous décrivons maintenant l'algorithme standard (Algorithme 10).

A chaque symbole $\sigma \in \Sigma$ est associé un ensemble de $\Sigma$-nœud dénoté $F_\sigma = \{x \mid lab(x) = \sigma\}$.

La fonction *initialize()* partitionne l'ensemble des états en se référant au Lemme 4.2. La fonction *smallest()* retourne la partition la plus petite dans l'ensemble donné comme paramètre. L'algorithme utilise des couples de $P : 2^Q \times X_\Sigma$ utilisées par la fonction *reset* .

Nous annonçons la complexité linéarithmique par le théorème suivant :

**Théorème 10.** *Algorithme 10 minimise un AAFA* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *en* $\mathcal{O}(|\mathcal{A}| \log(|Q|))$

*Démonstration.* Ce résultat peut être vérifié suivant les même étapes cités dans [Hop71]. Chaque paire $(p, y)$ appartient à au plus $\log n$ paires $P$ qui entrent dans $T$. En effet, chaque bloc $P$ qui entre dans $T$ provient d'une coupure d'une partie en choisissant la plus petite des deux. Pour chaque

**Algorithme 10** Minimisation linéarithmique

1: $P \leftarrow initialize(Q)$
2: $S \leftarrow smallest(P)$
3: $Treat \leftarrow \emptyset$
4: **pour tout** $\sigma \in \Sigma$ **faire**
5: $\quad Treat \leftarrow Treat \cup \{(S, \sigma)\}$
6: **Fin pour**
7: **tanque** $Treat \neq \emptyset$ **faire**
8: $\quad$ let $T = (S, \sigma) \in Treat$
9: $\quad Treat = Treat - T$
10: $\quad$ **pour tout** $p \in P$ and $\sigma \in \Sigma$ **faire**
11: $\quad\quad$ let $h \in p$
12: $\quad\quad$ **pour** $i = 1..Arity(\sigma)$ **faire**
13: $\quad\quad\quad p_1 \leftarrow \{q \mid \exists x \in F_\sigma : N_x(q) = succ(x) - (q, i) \wedge pred(x) \in S\}$
14: $\quad\quad\quad p_2 \leftarrow P - p_1$
15: $\quad\quad$ **Fin pour**
16: $\quad\quad$ **si** $\exists y \in X_\Sigma : (p, y) \in T$ **alors**
17: $\quad\quad\quad$ replace $(p, y)$ by $(p_1, y), (p_2, y)$ in $T$
18: $\quad\quad$ **sinon**
19: $\quad\quad\quad T \leftarrow T \cup smallest(p_1, p_2)$
20: $\quad\quad$ **Fin si**
21: $\quad\quad reset(P)$
22: $\quad$ **Fin pour**
23: **Fin tanque**

paire $(p, y)$ de $P$, lorsque $P$ sort de $T$, on examine (et traite en temps constant) l'origine des flèches d'étiquette $y$ arrivant sur $p$. La complexité globale est donc $logn$ fois la somme pour tous les états $p$ du nombre de flèches arrivant sur $p$. La complexité à atteindre est donc $\mathcal{O}(|Q| \log(|Q|))$. Néanmoins, la propriété $\exists x \in F_\sigma : N_x(q) = succ(x) - (q, i) \wedge pred(x) \in S$ influe sur la complexité générale. En effet, vu le rang maximum de l'alphabet et la taille de $X_\Sigma = |\Delta|$, la complexité générale peut atteindre $\mathcal{O}((\hat{r}|\Delta| + |Q|) \log(|Q|)) = \mathcal{O}(|A| \log(|Q|))$. □

### 4.1.3 Application à la minimisation incrémentale

Afin de donner une bonne implémentation à la fonction récursive ainsi qu'à l'algorithme global de minimisation incrémentale, nous allons présenter dans cette section une implémentation basée sur la traçabilité des équivalences calculées. Chaque vérification est donc enregistrée afin d'éviter la répétition et donc améliorer le temps de calcul [GCNZ12].

#### 4.1.3.1 États équivalents et états dissimilaires

La réduction des calcul lors du processus de minimisation est un problème qui à été massivement étudié pour les automates de mots (voir [Wat93, GVVL15]). En effet, une des techniques proposées est l'utilisation d'un ensemble particulier $D$ présentant la notion de'*dissemblance*. Cette notion n'est

rien que le regroupement des états jugés non équivalents. Néanmoins, cet ensemble peut être gourmand en mémoire et en temps d'accès et de calcul. Il est clair que la relation $\not\simeq$ est n'est ni une relation d'équivalence ni une relation d'ordre (absence de transitivité).

Pour cette raison, nous adaptons cette notion et nous ajoutons une autre dimension afin de permettre quelques réductions en temps et espace.

Nous définissons deux ensembles particuliers $\mathcal{I}$ pour les états équivalents et $\mathcal{D}$ pour les états dissemblables. Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFA et $G_{\mathcal{A}} = (X, lab, A, \gamma)$ son graphe associé.

$$\begin{aligned} \mathcal{I}(p) &= \{q \in Q \mid p \simeq q\} \cup \{p\} \\ \mathcal{D}(p) &= \{q \in Q \mid p \not\simeq q\} \end{aligned}$$

Donc, on peut intuitivement déduire les propriétés suivantes :

**Lemme 4.3.** *Soient $p, q \in Q$ :*

$$\begin{aligned} (\mathcal{I}(p) \cap \mathcal{I}(q) \neq \emptyset) &\Rightarrow p \simeq q \\ (\mathcal{D}(p) \cap \mathcal{I}(q) \neq \emptyset) &\Rightarrow p \not\simeq q \end{aligned}$$

En implémentant les deux ensembles $\mathcal{I}$ et $\mathcal{D}$ comme des vecteurs de bits, la vérification des deux propriétés peut se faire en $log(|Q|)$ étapes.

Notons que le lemme 4.2 peut être utilisé pour initialiser $\mathcal{D}$ avec les états ayant différentes signatures.

#### 4.1.3.2 Détection des cycles

Par analogie aux travaux sur les automates de mots [Wat01], la méthode incrémentale peut être d'une extrême utilité pour les AAFA cycliques et acycliques. Une technique de détection de cycles parait évidente.

Dans ce contexte, nous proposons le lemme suivant qui peut être considéré comme un résultat direct du lemme de l'étoile.

**Lemme 4.4.** *Soient $q \in Q$ et $t \in L^{\downarrow}(q)$. Alors si $\exists q' \in Q, r \in St(t) : t(r \Leftarrow t) \in L^{\downarrow}(q')$ alors $q$ apparaît au moins dans un cycle et $L^{\downarrow}(q)$ à une taille infinie.*

Sachant que les automates traités sont déterministes, si un arbre appartenant au langage accepté par un état quelconque est un sous arbre d'un autre arbre accepté par le même état, alors cet arbre peut être rencontré plus d'une fois (cycle) dans un même chemin.

Par conséquent, si un AAFA est cyclique, alors il existe des états pour lesquels le calcul de $\delta$ est infini.

Comme un résultat direct de ce lemme :

**Corollaire 4.1.** *Le graphe $G_{\mathcal{A}}$ es cyclique si et seulement s'il existe $q \in Q$ tel qu'on peut trouver une sequence $q, q_1, \ldots, q_n, q \in \hat{q}, q_1, \ldots, q_n \in Q$.*

Cependant, le nombre de cycles dans un graphe peut augmenter d'une manière exponentielle vis à vis du nombre des arcs [Joh75]. Heureusement, des algorithmes de référence peuvent naturellement détecter les nœuds participant dans des cycles. Un algorithme excellent pour cette situation est l'algorithme de détection des *composantes fortement connexes* pour un graphe. Plusieurs implémentations existent dans la littérature. Nous utilisons l'algorithme de Tarjan [Tar72] qui peut détecter les composantes fortement connexes en un temps linéaire. Dans notre cas, la détection de ces composants se fait en $\mathcal{O}(|Q| + |\mathcal{A}|)$, le nombre d'arcs et de nœuds dans le graphe associé.

Maintenant, en tenant en compte que le calcul des composantes fortement connexes partitionne l'ensemble des nœuds du graphe, nous mettons $\Pi_Q(G_{\mathcal{A}})$ le partitionnement des états de $G_{\mathcal{A}}$ en blocs d'états figurant dans la même composante. $\pi(q)$ est la composante contenant un état $q \in Q$. Nous avons :

**Lemme 4.5.** *Soit $\rho = (\sigma, q_1, \ldots, q_n), \sigma \in \Sigma_n, q_1, \ldots, q_n \in Q$. Soient $p, q \in Q$ tel que $\delta(\rho_{p:i}) = p'$ et $\delta(\rho_{q:i}) = q'$. Alors, si $p' \in \pi(p) \wedge q' \notin \pi(q)$ donc $p \not\approx q$.*

**Corollaire 4.2.** *$\mathcal{A}$ est acyclique si et seulement si $\forall \pi \in \Pi_Q(G_{\mathcal{A}}), |\pi| = 1$.*

Évidemment, si aucune composante fortement connexe avec plus d'un état n'existe, chaque composante contiendra un seul état.

#### 4.1.3.3 Amélioration de l'algorithme incrémental

Maintenant, nous étendons la notion de prédécesseurs aux états :

**Définition 4.6.** *Soit $q \in Q$. L'ensemble des prédécesseurs $pred(q)$ de $q$ est :*

$$pred(q) = \{x \in X_\Sigma \mid (p, x) \in A\}$$

Toutefois, il est important de noter que l'utilisation des deux ensembles $\mathcal{I}$ et $\mathcal{D}$ ne permet pas d'éviter la complexité exponentielle de la minimisation mais seulement de réduire quelques paramètres. Une autre idée consiste à utiliser des techniques de stockage et de vérifications efficaces afin d'éviter de refaire plusieurs calculs. Pour cette raison, nous définissons une structure spéciale $Trace$ mémorisant pour chaque paire d'états inspectée le chemin parcouru ainsi que la profondeur de la récursivité. Dans l'Algorithme 11, la fonction $addNode(P, Q)$ ajoute la paire d'états $Q$ à la paire $P$ comme sous arbre dans $Trace$. Nous définissons aussi la fonction $subTree(node)$ afin de regrouper toutes les paires d'états appartenant au sous arbre $node$ dans $Trace$.

$Trace$ est utilisée ensuite pour décider si deux états sont équivalents ou bien un nouvel appel à $equiv()$ doit être invoqué. La fonction $update(p, q, \mathbf{B})$ met à jours les deux ensembles $\mathcal{I}$ et $\mathcal{D}$ pour la paire d'états $(p, q)$ suivant le booléen $\mathbf{B}$. Ce booléen est mis à *vrai* quand $p$ et $q$ sont équivalent et *faux* sinon.

**Algorithme 11** Nouvelle fonction d'équivalence

1: **Fonction** EQUIV$(p, q, k)$
2: **si** $k = 0$ **alors**
3: $eq \leftarrow ((p \in Q_f) \wedge (q \in Q_f))$
4: **sinon si** $(\mathcal{I}(p) \cap \mathcal{I}(q) \neq \emptyset)$or$\{p, q\} \in S$ **alors**
5: $eq \leftarrow$`true`
6: **sinon si** $(\mathcal{D}(p) \cap \mathcal{I}(q) \neq \emptyset)$ **alors**
7: $eq \leftarrow$`false`
8: **sinon**
9: $S \leftarrow S \cup \{\{p, q\}\}$
10: $eq \leftarrow ((p \in Q_f) \wedge (q \in Q_f))$
11: **pour** $x \in pred(p)$ **faire**
12: **si** $\exists y \in pred(q), lab(x) = lab(y) \wedge N_x(p) = N_y(q) \wedge |pred(p)| = |pred(q)|$ **alors**
13: $eq \leftarrow eq \wedge equiv(pred(x), pred(y), k - 1)$
14: **Fin si**
15: **si** $\overline{eq}$ **alors**
16: $update(p, q,$ `false`$)$
17: **break ;** ▷ la boucle s'arrête ici, $\{pred(x), pred(y)\}$doit être ajouté à $\{p, q\}$ comme sous arbre dans $Trace$.
18: **Fin si**
19: $addNode(\{p, q\}, \{pred(x), pred(y)\})$
20: **Fin pour**
21: $S \leftarrow S - \{\{p, q\}\}$
22: **Fin si**
23: **return** $(eq)$
24: **Fin Fonction**

Quand la fonction $equiv$ est appelée, $Trace$ est créée, contenant toutes les paires d'états testées durant les appels récursifs ultérieurs. Afin d'éviter la réutilisation de ces états, nous introduisons l'Algorithme 12 opérant sur $Trace$. Ici, nous rencontrons trois cas :

1. $equiv$ renvoie `vrai`, alors toutes les paires localisées dans $Trace$ sont équivalentes,
2. $equiv$ renvoie `faux` ($equiv$ rencontre un break) et la taille de $Trace$ est inférieure à $|Q| - 2$, alors tous les sous arbres concernés par le break reçoivent `faux`. Les paires restantes reçoivent `vrai`,
3. $equiv$ renvoie `faux` et la taille de $Trace$ est exactement $|Q| - 2$, alors tous les sous arbres participant dans un cycle et ceux concernés par un break sont mémorisés dans une matrice spéciale $Stack$. Les paires restantes reçoivent `vrai`.

On peut restaurer un chemin parcouru à l'aide de $Stack$. L'algorithme 13 effectue cette action.

De plus, nous proposons un préordre afin d'optimiser les appels $equiv$ dans l'algorithme principal.

Soient $p, q \in Q$, nous mettons $p \preceq q \Leftrightarrow \mathcal{L}(p) \leq \mathcal{L}(q)$. Il est évident que ce préordre est total.

**Corollaire 4.3.** *Il existe un état* $p \in Q$ *tel que* $\forall q \in Q : q \preceq p \wedge \mathcal{L}(p) \leq |Q|$.

Ici, on peut voir que le niveau d'un état est majoré par $|Q|$. Cette valeur est la longueur du chemin le plus long sans répétition qui peut être rencontré dans un graphe.

**Algorithme 12** Traitement de $Trace$

**Fonction** TREAT($Trace$)
  **si** $\nexists node \in Trace$ marked $|Q|-2$ **alors**
    **pour tout** $p, q \in subTree(Trace)$ **faire**
      $update(p, q, \texttt{true})$
    **Fin pour**
  **sinon**
    **pour tout** $p, q \in Subtree(Trace)$ **faire**
      **si** $\exists subTree(p, q)$ marked $|Q|-2$ **alors**
        $Stack(p, q) \leftarrow \{(N, depth(N)) \mid N \in subTree(p, q)\}$
      **sinon**
        $update(p, q, \texttt{true})$
      **Fin si**
    **Fin pour**
  **Fin si**
**Fin Fonction**

**Algorithme 13** Verify stack

**Fonction** CALLSTACK($p, q$)
  **pour tout** $((q_1, q_2), k)) \in Stack(p, q)$ **faire**
    $eq \leftarrow eq \wedge equiv(q_1, q_2, k)$
  **Fin pour**
  **return** $eq$
**Fin Fonction**

Suivant ce préordre, nous donnons le principe général pour effectuer le processus incrémental :

1. Commencer par tester les états finaux ($\forall p, q \in Q : \mathcal{L}(p) = \mathcal{L}(q) = 0$).
2. A chaque itération, traiter les états qui participent dans des transitions produisant les états traités dans l'étape précédente. En d'autres termes, quand une itération $i$ est en cours, traiter tous les états dans l'ensemble $\{p, \mathcal{L}(p) = i\}$.

L'Algorithme 14 représente la fonction principale. $CSCC(G_{\mathcal{A}})$ est la fonction qui calcule les composants fortement connexes. Concernant la fonction $InOut(p, q)$, elle renvoie un booléen indiquant si deux états $p$ et $q$ sont équivalents ou non suivant le lemme 4.5.

La fonction principale utilise deux ensembles : $V$ pour les paires d'états vérifiées et $R$ pour les paires concernées par l'étape suivante. Premièrement, $R$ est initialisé par des couples d'états finaux $Q_f$ ($\mathcal{L}(q \in Q_f) = 0$). Ensuite, à chaque itération, des couples de $R^2$ sont vérifiés et stockés dans $V$. $R$ est alors mis à jour pour le niveau suivant. Enfin, l'algorithme se termine quand toutes les possibilités sont traitées (stabilité de $V$). Afin d'accélérer l'algorithme et éviter des calculs répétés, $Stack$ est vérifié à chaque appel de $equiv$.

**Théorème 11.** *Un AAFAD peut être minimisé utilisant Algorithme 14 en* $\mathcal{O}(\hat{r}(|\mathcal{A}| + |Q|^3 log(|Q|)))$.

La fonction principale exécute exactement $|Q| \times |Q-1|/2$ appels. A chaque appel, une profondeur maximale de $|Q-2|$ est assurée. En effet, le calcul des prédécesseurs et des voisins (calcul de$\doteq$)

**Algorithme 14** Nouvelle fonction main

```
1:  Fonction MAIN
2:      CSCC(G_A)
3:      S ← ∅
4:      pour tout q ∈ Q faire
5:          I(q) ← q
6:          D(q) ← ∅
7:      Fin pour
8:      R ← Q_f                                   ▷ Le niveau initial est 0
9:      V ← ∅
10:     V' ← ∅
11:     répéter
12:         T = {(p, p) | p ∈ Q}
13:         pour tout (p, q) ∈ R^2 − (V ∪ T) faire
14:             Trace ← ∅
15:             si InOut(p, q) alors
16:                 update(p, q, false)
17:             sinon
18:                 update(p, q, equiv(p, q, |Q| − 2))
19:                 Treat(Trace)
20:             Fin si
21:         Fin pour
22:         H ← R
23:         V' ← V
24:         V ← V ∪ R^2
25:         R ← {q | ∃q' ∈ H, n ∈ N, x ∈ X_Σ : (x, q'), (q, x) ∈ A}   ▷ On a L(q) = L(q') + 1
26:     jusqu'a V = V'
27: Fin Fonction
```

peut être fait en amant. Nous rappelons que le calcul de $\doteq$ peut être fait en $\mathcal{O}(\hat{r}|\mathcal{A}|)$ car pour chaque $\Sigma$-nœud, $\hat{r}$ états au plus peuvent y être liés. De quoi, le test de ces états (Algorithme 11,ligne 12) peut être fait en un temps constant. En outre, aucune branche n'est recalculée.

Un résultat important est l'application de cette méthode dans le cas acyclique :

**Corollaire 4.4.** *un AAFAD acyclique peut être minimisé en* $\mathcal{O}(|\mathcal{A}|log(|Q|))$.

En effet, il n'est pas nécessaire d'utiliser $Trace, Stack$ car aucun cycle n'est présent. Seulement, une version primitive de la fonction $equiv$ est suffisante.

## 4.2 Méthode indirecte

Notre deuxième approche repose sur le constat démontré que la minimisation d'un AAFAD peut se faire moyennant la minimisation d'automates de mots. Cette approche est très importante car elle permet d'utiliser le grand nombre d'outils et de Framework implémentés pour la minimisation d'automates sur les mots. En effet, les boites à outils pour les arbres souffrent du nombre très réduit d'algorithmes réellement implémentés comparés à ceux pour les automates de mots [CHM03, CH09].

En effet, notre approche se base principalement sur trois étapes [GCZW15].

- la construction d'un automate de mots associé à un AAFAD.
- la minimisation de l'automate de mots.
- l'extraction de l'AAFAD minimal depuis son homologue de mots.

Afin de construire l'automate de mots associé à l'AAFAD, nous utilisons quelques propriétés basiques.

Rappelons qu'une idée similaire a été introduite par Abdulla et al. [ABH+09]. Dans le but de minimiser un AAFA *non déterministe*, les auteurs ont calculé des relations de bissimulation à travers des systèmes de transitions (qui sont une sorte d'automate finis pour les mots).

Ici, nous rappelons la partie qui nous intéresse dans notre approche.

La première astuce est de définir un *environnement* pour chaque occurrence d'un état quelconque dans une transition :

**Définition 4.7.** *Un environnement pour un état $q_i$ est le tuple$((q_1, \ldots, q_{i-1}, \bullet, q_{i+1}, \ldots, q_n), f, q)$ obtenu en remplaçant l'état $q_i$ par un symbole spécial $\bullet \notin Q$. L'ensemble de tous les environnements est dénoté $Env(\mathcal{A})$.*

*Les auteurs transforment ainsi l'automate voulu en un système de transitions : $TS = (Q^\bullet, \Delta^\bullet)$ pour calculer une relation d'équivalence comme suit :*

- *$Q^\bullet$ contient un état $q^\bullet$ pour chaque $q \in Q$ et un état $e^\bullet$ pour chaque environnement $e \in Env(\mathcal{A})$.*
- *$\Delta^\bullet$ est le plus petit ensemble satisfaisant : si $(f, q_1 \ldots, q_n, q) \in \Delta$ alors $\Delta^\bullet$ contient $q_i^\bullet \to e_i^\bullet$ et $e_i^\bullet \to q^\bullet$ où $e_i^\bullet$ a la même forme que $((f, q_1, \ldots, q_{i-1}, \bullet, q_{i+1}, \ldots, q_n), f, q)$.*

Il est à noter que ce système de transitions peut être vu comme un *système de transitions étiqueté* si chaque environnement est vu comme une "étiquette" entre deux états.

Les auteurs ont prouvé qu'à travers cette transformation, le calcul de la relation de bissimulation peut être fait directement sur le nouveau système de transitions de mots. Par conséquent, ils ont employé directement l'algorithme de Tarjan-Paige [PT87].

Pour le cas déterministe, cette approche peut s'adapter en employant les résultat de Högberg et al. [HMM09].

Maintenant, nous procédons à la présentation de notre approche. En effet, la construction de l'automate de mots associé est faite via trois étapes :

- Calculer un langage de mots dit *horizontal* à partir des transitions de l'AAFAD à minimiser.
- Construire et puis minimiser un automate acyclique reconnaissant ce langage.
- Extraire les étiquettes qui vont servir d'alphabet pour l'automate de mots associé depuis l'automate acyclique minimal.
- Construire l'automate de mots associé à l'AAFAD en question. La minimisation de ce dernier coïncide avec l'automate d'arbre minimal recherché.

### 4.2.1 Calcul de l'alphabet pour l'automate de mots

Dans les définitions suivantes, nous associons un langage de mots $L_\Delta$ qu'on va appeler *langage horizontal* à l'ensemble des transitions $\Delta$. Ensuite, nous définissons une relation d'équivalence sur les états utilisant ce langage.

**Définition 4.8** (Langage horizontal)**.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD. Le langage horizontal de $\Delta$ noté $L_\Delta$ est défini comme suit :*

$$L_\Delta = \bigcup_{\rho \in \Delta} L_\rho \tag{4.1}$$

$$L_\rho = \bigcup_{i=1}^{n} q_i f q_1 \dots q_{i-1} \bullet q_{i+1} \dots q_n \tag{4.2}$$

*où $\rho = (f, q_1, \dots, q_n, q)$ et $\bullet \notin \Sigma_0 \cup Q$ est un symbol spécial.*

Maintenant, nous définissons la relation d'équivalence $\sim$ sur $L_\Delta$.

**Définition 4.9.** *Soient $p, q \in Q$. On dit que $p$ et $q$ sont susceptibles d'être équivalents (on note $p \sim q$), si et seulement si $(p \in Q_f) \Leftrightarrow (q \in Q_f)$ et pour tout $f \in \Sigma, u, v \in Q^* : pfu \bullet v \in L_\Delta \Leftrightarrow qfu \bullet v \in L_\Delta$.*

**Proposition 4.2.** *Pour $p, q \in Q$, on a $p \simeq q \Rightarrow p \sim q$*

*Démonstration.* Soient $p, q \in Q$ où $p \not\sim q$. Donc, il existe $pfq_1 \dots q_{i-1} \bullet q_{i+1} \dots q_n \in L_\Delta$ mais $pfq_1 \dots q_{i-1} \bullet q_{i+1} \dots q_n \notin L_\Delta$. En utilisant Définition 4.8, nous avons $(f, q_1, \dots, q_{i-1}, p, q_{i+1}, \dots, q_n, p') \in \Delta$ mais aucune transition sous la forme $(f, q_1, \dots, q_{i-1}, q, q_{i+1}, \dots, q_n, q')$ ne figure parmi les transitions. Ici nous trouvons que $(f, q_1, \dots, q_{i-1}, p, q_{i+1}, \dots, q_n) \in \Gamma(p)$ mais $(f, q_1, \dots, q_{i-1}, p, q_{i+1}, \dots, q_n)(p :_i q) \notin \Gamma(q)$ bien que $i \in occ(p)$. Par conséquent $p \not\simeq q$. □

En outre :

**Lemme 4.6.** *La relation d'équivalence $\sim$ peut être calculée en $O(\hat{r}|\mathcal{A}|)$.*

*Démonstration.* Évidemment, la taille de $L_\Delta$ est $\hat{r}|\mathcal{A}|$. Nous pouvons procéder en définissant un automate acyclique de mots. Cet automate reconnaît les mots depuis $L_\Delta$ commençant par l'état en question. L'assemblage de tous les automates acycliques construits nécessite $\hat{r}|\mathcal{A}|$ en temps et en espace. De plus, l'automate résultant (qu'on appelle $A_{L_\Delta}$) peut être minimisé en temps linéaire (cas pour les automates acycliques de mots [Rev92]) :

$$L_\Delta \xrightarrow{\hat{r}|\mathcal{A}|} A_{L_\Delta} \xrightarrow{\hat{r}|\mathcal{A}|} \min(A_{L_\Delta})$$

□

L'Algorithme 15 est une implémentation du lemme 4.6. Premièrement, un automate de mots vide $A_{L_\Delta}$ est créé. Un automate de mots acyclique $A_q$ est créé pour chaque $q \in Q$ lisant des mots de la forme $qfq_1 \ldots, \bullet, \ldots q_n \in L_\Delta$. Ces automates élémentaires peuvent être vus comme des "arbres des préfixes" représentant $L_\Delta$. La fonction $Add$ ajoute un automate élémentaire $A_q$ à $A_{L_\Delta}$ et crée une transition depuis l'état initial 0 vers l'état initial de $A_q$. Finalement, $Acyc\_Min$ minimise d'une manière acyclique l'automate $A_{L_\Delta}$. La complexité globale de cet algorithme est $\mathcal{O}(|Q| + \hat{r}|\mathcal{A}| + \hat{r}|\mathcal{A}|) = \mathcal{O}(\hat{r}|\mathcal{A}|)$.

**Algorithme 15** Calcul de $\sim$

**Fonction** $SIM\_COMPUTATION$(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
  $A_{L_\Delta} = (\{0\}, Q \cup \Sigma, \{0\}, \emptyset, \emptyset)$
  **pour tout** $q \in Q$ **faire**
    soit $A_q = (\emptyset, Q \cup \Sigma, \{0\}, \emptyset, \emptyset)$
  **Fin pour**
  **pour tout** $(f, q_1, \ldots, q_n, q) \in \Delta$ **faire**
    **pour tout** $q_i \in q_1, \ldots, q_n$ **faire**
      $Add(A_{q_i}, fq_1 \ldots \bullet \ldots q_n)$
    **Fin pour**
  **Fin pour**
  **pour tout** $q \in Q$ **faire**
    $Assembly(A_{L_\Delta}, A_q)$
  **Fin pour**
  $Acyc\_Min(A_{L_\Delta})$
  $return(A_{L_\Delta})$
**Fin Fonction**

**Exemple 4.4.** *Nous considérons l'automate $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ présenté dans la Figure 4.3.*

1. $Q = \{q_1, q_2, q_3, q_4, q_5\}$
2. $\Sigma = \{a, b, g(), f(,)\}$
3. $Q_f = \{q_3, q_4, q_5\}$
4. $\Delta = \{(a, q_1), (b, q_2), (g, q_1, q_3), (g, q_2, q_4), (f, q_2, q_4, q_4), (f, q_1, q_3, q_4), (f, q_1, q_4, q_4), (f, q_2, q_3, q_4), (g, q_4, q_5\}$

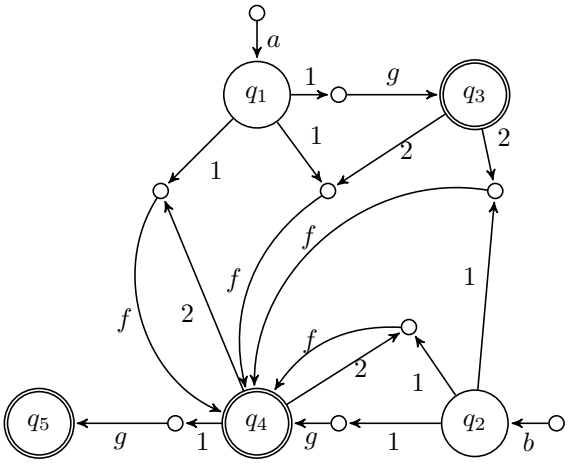



FIGURE 4.3: Exemple d'un AAFAD

*Le calcul du langage horizontal produit :*

$$L_\Delta = \{a, b, q_1 g\bullet, q_1 f \bullet q_3, q_1 f \bullet q_4, q_2 g\bullet, q_2 f \bullet q_4,$$
$$q_2 f \bullet q_3, q_3 f q_2\bullet, q_3 f q_1\bullet, q_4 f q_2\bullet, q_4 f q_1\bullet, q_4 g\bullet, q_5\}$$

*Un exemple des automates élémentaires créés pour tous les états est illustré par Figure 4.4. Ici, l'automate associé à l'état $q_1$ est présenté. Cet automate reconnaît $g\bullet$, $f \bullet q_3$ et $f \bullet q_4$.*

*Aprés, l'automate minimal du langage $L_\Delta$ est présenté dans Figure 4.5.*

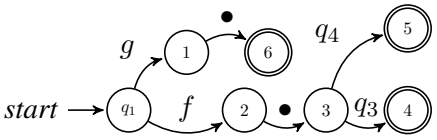



FIGURE 4.4: L'automate acyclique associé à $q_1$

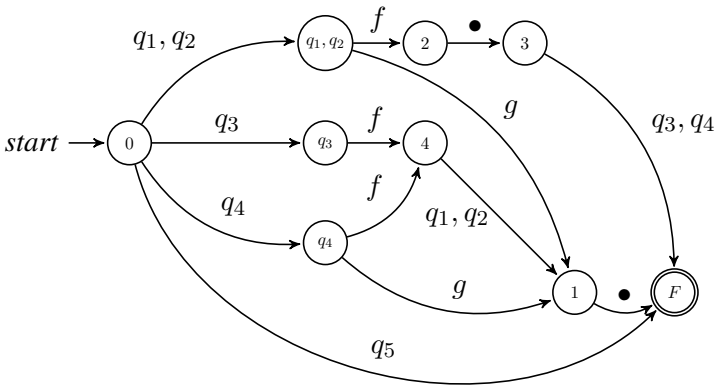



FIGURE 4.5: L'automate minimal reconnaissant $L_\Delta$

Nous associons à chaque état $q$ qui figure dans une transition $(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots q_n)$ la *lettre* $f_{q_1 \cdots q_{i-1}, q_{i+1} \cdots q_n}$ afin de produire un alphabet de mots qui servira pour la construction de l'automate de mots.

Maintenant, après avoir calculé le langage horizontal $L_\Delta$ et identifier les lettres affectées à chaque état. Nous procédons par la définition de l'alphabet nécessaire pour la construction de l'automate de mots ciblé. Néanmoins, certaines lettres ne sont pas utiles pour la minimisation. Pour cette raison, nous filtrons cet ensemble grâce à la définition de l'ensemble des états $\overline{Q}$ susceptible d'être équivalents à d'autres états.

**Définition 4.10.** *Soit $\overline{Q} = \{q \mid \exists p \neq q$ tel que $q \sim p\}$. Pour chaque état, nous associons l'alphabet $\sigma_q$ défini comme suit :*

$$\sigma_q = \{f_{u,v} \mid qfu \bullet v \in L_\Delta\}. \tag{4.3}$$

**Proposition 4.3.** *Soit* $\sigma = \bigcup_{q \in \overline{Q}} \sigma_q$ *l'alphabet associé à* $\overline{Q}$ *alors* $|\sigma| \leq |\mathcal{A}|$

L'automate de mots associé sera défini sur l'alphabet $\sigma$. Néanmoins, la représentation des éléments de $\sigma$ est toujours sous la forme de chaînes de caractères. Le traitement de ces symboles, qui nécessite d'éventuelles comparaisons, peut accroître la complexité globale. Par conséquent, un système d'étiquetage des éléments de $\sigma$ permettra d'éviter ce problème. Nous proposons d'étiqueter chaque symbole de $\sigma$ par le numéro de l'état figurant comme sa feuille dans l'arbre des préfixes : Nous définissons le système d'étiquetage comme une application : $\varphi : \sigma \to \mathbb{N}$. Il est clair que cet étiquetage est unique. La construction et l'exploration d'un arbre de préfixes nécessite $\mathcal{O}(|\mathcal{A}|)$ (voir [WD03]).

**Corollaire 4.5.** *$\sigma$ peut être calculé et étiqueté en $\mathcal{O}(|\mathcal{A}|)$.*

**Exemple 4.5.** *En prenant le même exemple (Figure 4.3), seuls les chemins sortants de $q_1, q_2$ sont considérés vu que $\overline{Q} = \{q_1, q_2\}$. Donc, nous aurons $\sigma = \{g_{\epsilon,\epsilon}, f_{\epsilon,q_3}, f_{\epsilon,q_4}\}$. Comme $q_1, q_2$ sont les seuls états susceptibles d'être équivalents, leurs automates $A_{q_1}$ et $A_{q_2}$ sont vus comme des arbres de préfixes. Depuis Figure 4.4, nous aurons $\varphi(g_{\epsilon,\epsilon}) = 6, \varphi(f_{\epsilon,q_3}) = 4, \varphi(f_{\epsilon,q_4}) = 5$.*

### 4.2.2 Construction de l'automate de mots associé

Maintenant, nous définissons l'automate de mots associé à un AAFAD :

**Définition 4.11.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD. L'automate de mots associé $M_\mathcal{A}$ est le tuple $M_\mathcal{A} = (Q', \sigma, \{i_s\}, F, \widehat{\delta})$ où $Q' = Q \cup \{i_s\}$ est l'ensemble des états, $\sigma = \bigcup_{q \in \overline{Q}} \sigma_q \cup Q$ est l'alphabet, $\{i_s\}$ est l'état initial, $F = Q_f$ est l'ensemble des états finaux et $\widehat{\delta} : Q' \times \sigma \to Q'$ la fonction de transition définie comme suit.*

- *pour chaque $q$ dans $\overline{Q}$ : $\widehat{\delta}(q, a) = q'$ où $a = f_{q_1 \cdots q_{i-1}, q_{i+1} \cdots q_n}, \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) = q'$.*
- *pour chaque $q$ dans $Q$ : $\widehat{\delta}(i_s, q) = q$.*
- *pour chaque $q$ dans $Q - \overline{Q}$ : $\widehat{\delta}(q, q) = q$.*

**Exemple 4.6.** *A travers le même exemple présenté dans la Figure 4.3, l'automate de mots associé est présenté dans Figure 4.6. Nous utilisons le système d'étiquetage.*

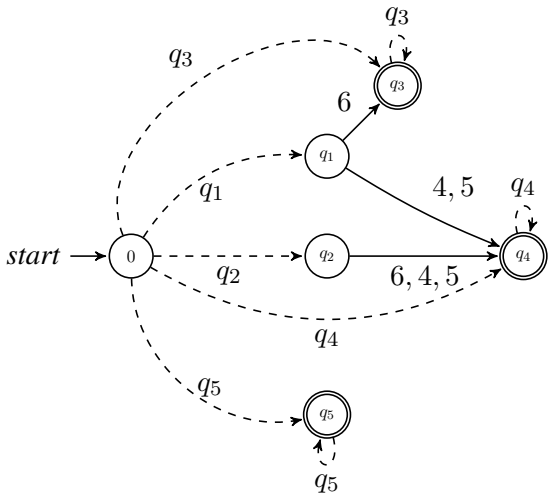



FIGURE 4.6: L'automate $M_{\mathcal{A}}$ associé l'AAFAD de l'Exemple 4.3.

Notons que $i_s$, $Q$ et les transitions sortantes de $i_s$ n'ont aucune importance dans le processus de minimisation vu que l'automate associé n'est pas utilisé pour reconnaître des arbres. Leurs utilisation est dans le but de garder la forme usuelle d'un automate de mots. De plus il est facile de démontrer que les états appartenant à $Q - \overline{Q}$ ne sont équivalent à aucun état :$\forall p \in Q, q \in Q - \overline{Q} : p \not\sim q$ alors $p \not\equiv q$.

Nous présentons dans Algorithme 16 un "filtre" qui calcule l'automate associé à un AAFAD. La fonction $Compute\_Sigma(ADFA)$ extrait $\sigma$ depuis $ADFA$ et mémorise le résultat dans des arbres de préfixes. La fonction $\varphi$ est la fonction d'étiquetage de symboles de $\sigma$. La fonction $\varphi^{-1}$ représente la fonction inverse de $\varphi$. Enfin, $export(DFA)$ renvoie l'automate associé.

**Algorithme 16** Filtre AAFAD vers automate de mots

**Fonction** $TTS\_Filter$(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$)
  $A_{L_\Delta} = SIM\_COMPUTATION(\mathcal{A})$
  $\sigma \leftarrow Compute\_Sigma(A_{L_\Delta})$
  $Q' \leftarrow Q \cup \{i_s\}$
  $\widehat{\delta} \leftarrow \emptyset$
  **pour** $q \in Q$ **faire**
    $\widehat{\delta} \leftarrow \widehat{\delta} \cup \{(i_s, q) \rightarrow q\}$
  **Fin pour**
  **pour tout** $a \in \sigma : \varphi^{-1}(a) = f_{q_1 \cdots q_{i-1}, q_{i+1} \cdots q_n}, \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n) = q'$ **faire**
    $\widehat{\delta} \leftarrow \widehat{\delta} \cup \{(q, a) \leftarrow q'\}$
  **Fin pour**
  $M_{\mathcal{A}} = (Q', \sigma, \{i_s\}, Q_f, \widehat{\delta})$
  $export(M_{\mathcal{A}})$
**Fin Fonction**

**Lemme 4.7.** *Algorithme 16 calcule l'automate associé à $\mathcal{A}$ en $\mathcal{O}(\hat{r}|\mathcal{A}|)$.*

### 4.2.3 Application à la minimisation standard

Maintenant, nous montrons que l'automate de mots résultant est déterministe.

**Proposition 4.4.** *L'automate $M_{\mathcal{A}}$ associé à l'AAFAD $\mathcal{A}$ est déterministe.*

*Démonstration.* Par définition, nous avons pour $q \in Q' - \overline{Q}, a \in \sigma : |\widehat{\delta}(q,a)| \leq 1$. Soit $q \in \overline{Q}$ et il existe un symbole $f_{u,v} \in \sigma$ tel que $\widehat{\delta}(q,a) = \{q', q''\}$ avec $q' \neq q''$. Let $u = q_1 \dots q_{i-1}$ et $v = q_{i+1} \dots q_n$. En se référant à la Définition 4.8, nous obtenons $(f, q_1, \dots, q_{i-1}, q, q_{i+1}, \dots, q_n, q') \in \Delta$ et $(f, q_1, \dots, q_{i-1}, q, q_{i+1}, \dots, q_n, q'') \in \Delta$. Ceci est considéré comme contradiction car $\mathcal{A}$ est déterministe. □

Dans ce qui suit, l'état $i_s$ et les transitions sortantes ne sont pas considérés. $\sigma$ est alors réduit à $\sigma - Q$.

Notons par $\approx$ la congruence de Nerode sur les états de $M_{\mathcal{A}}$ :

$$p \approx q \Leftrightarrow \begin{cases} p \in F \Leftrightarrow q \in F \\ \text{pour tout } a \in \sigma, \widehat{\delta}(p,a) \approx \widehat{\delta}(q,a) \end{cases} \tag{4.4}$$

**Proposition 4.5.** *Soient $p, q \in Q$, alors nous avons $(p \approx q) \Leftrightarrow (p \simeq q)$*

*Démonstration.* Il est clair que l'équivalence de Nerode $\approx$ peut être calculée en utilisant les approximations successives $\approx_j$ suivantes :

$$p \approx_0 q \quad \text{si et seulement si} \quad (p \in F \Leftrightarrow q \in F) \tag{4.5}$$

$$p \approx_{j+1} q \quad \text{si et seulement si} \quad p \approx_j q \text{ et pour tout } a \in \sigma, p, q \in Q' : \widehat{\delta}(p,a) \approx_j \widehat{\delta}(q,a). \tag{4.6}$$

Comme précédemment mentionné, les états inutiles qui figurent dans l'automate de mots sont équivalents (leurs langages droits sont vides et donc égaux). Ce qui peut fausser la minimisation. Pour éviter ce scénario, nous initialisons $\approx_0$ comme suit :

$$\approx_0 = \overline{Q}^2 \cup \{(q,q) \mid q \in Q - \overline{Q}\}$$

Il suffit de montrer que pour tout $p, q \in Q, j \in \mathbb{N} : p \approx_j q \Leftrightarrow p \simeq_j q$. Ceci est fait par induction sur les définitions de $\simeq_j$ et de $\approx_j$. Pour le cas de base, nous avons $(j = 0)$, $Q_f = F$. Pour $p, q \in Q$, nous avons $p \approx_0 q \Leftrightarrow p \simeq_0 q$. Supposons maintenant que pour tout $p, q \in Q, (p \approx_k q) \Leftrightarrow (p \simeq_k q)$ pour un nombre entier $k \geq 0$ :

Premièrement, nous montrons le premier sens de l'implication : $(p \approx_{k+1} q) \Rightarrow (p \simeq_{k+1} q)$ :

Supposons que $(p \approx_{k+1} q)$. Par approximations successives de $\approx$ nous avons :

$$(p \approx_{k+1} q) \quad \Leftrightarrow \quad p \approx_k q \text{ et pour tout } f_{u,v} \in \sigma, \widehat{\delta}(p, f_{u,v}) \approx_k \widehat{\delta}(q, f_{u,v}) \tag{4.7}$$

En appliquant l'hypothèse d'induction nous aurons :

$$(p \approx_{k+1} q) \quad \Leftrightarrow \quad p \simeq_k q \text{ et pour tout } f_{u,v} \in \sigma, \widehat{\delta}(p, f_{u,v}) \simeq_k \widehat{\delta}(q, f_{u,v}) \tag{4.8}$$

Soit $u = q_1 \cdots q_{i-1}$ et $v = q_{i+1} \cdots q_n$. La définition du langage horizontal donne $pfu \bullet v, qfu \bullet v \in L_\Delta$. Donc, il existe deux états $p'$ et $q' \in Q$ tel que $(f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n, p')$ et $(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n, q')$ sont dans $\Delta$. Utilisant Définition 4.11 nous avons : $p' = \widehat{\delta}(p, f_{u,v}) = \delta(f, q_1, \ldots, q_{i-1}, p, q_{i+1}, \ldots, q_n)$ et $q' = \widehat{\delta}(q, f_{u,v}) = \delta(f, q_1, \ldots, q_{i-1}, q, q_{i+1}, \ldots, q_n)$. Finalement, nous avons $(p \simeq_{k+1} q)$.

La démonstration du deuxième sens $(p \simeq_{k+1} q) \Rightarrow (p \approx_{k+1} q)$ se fait avec les mêmes étapes.

□

On peut vérifier le corollaire suivant :

**Corollaire 4.6.** *Soit $p \in Q' - \overline{Q}, q \in Q'$, (avec $q \neq p$) alors $p \not\approx q$.*

Par conséquent, seuls les états de $\overline{Q}$ sont considérés par le processus de minimisation (On peut facilement montrer que $i_s$ n'est équivalent à aucun autre état). Utilisant Corollaire 4.5 et [HU79], nous avons :

**Lemme 4.8.** *Soit $\mathcal{A}(Q, \Sigma, Q_f, \Delta)$ un AAFAD. Soit $M_\mathcal{A}$ son automate de mots associé. Alors, $M_\mathcal{A}$ peut être minimisé en $\mathcal{O}(|Q| + |\mathcal{A}| \log |Q|) = \mathcal{O}(|\mathcal{A}| \log |Q|)$.*

Par conséquent, en utilisant le lemme 4.6, Corollaire 4.5 et le lemme 4.8 nous aurons :

**Théorème 12.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD, $\mathcal{A}$ peut être minimisé en $\mathcal{O}(\hat{r}|\mathcal{A}| + |\mathcal{A}| \log |Q|)$.*

En terme de complexité, chaque état se connecte avec au plus $\min(\hat{r}, |\Delta|)$ autres états où $\hat{r}$ est le rang maximal de $\Sigma$. Donc la taille maximale de $\sigma$ sera $\frac{\min(\hat{r}, |\Delta|) \times |\Delta|)}{2} < |\mathcal{A}|$. Le gain est donc apprécié.

### 4.2.4 Application aux autres variantes de minimisation

Dans cette partie, nous montrons que cette approche peut être adaptée aux différentes techniques de minimisation à savoir la minimisation acyclique, incrémentale ou bien la construction à base d'exemples.

#### 4.2.4.1 Application à la minimisation acyclique

Les automates acycliques présentent une structure de données très bénéfique et peut stocker un ensemble fini d'objets arborescents (même très grand en taille). Un exemple concret est les documents XML (voir [BLMN15]). La minimisation acyclique a été discutée pour les automates de mots comme [Wat03, Rev92], il a été prouvé qu'elle peut être réalisée en un temps linéaire. Mais dans le cas des arbres, aucun algorithme n'a été discuté à notre connaissance. Heureusement, l'automate associé peut être utilisé pour minimiser un AAFAD acyclique en un temps linéaire aussi.

Le lemme suivant montre la nature de l'automate associé à un AAFAD acyclique.

**Lemme 4.9.** *L'automate de mots associé à un AAFA acyclique l'est aussi.*

Par conséquent, la minimisation peut être faite en temps linéaire utilisant des techniques sur les mots comme : [Rev92, Bub11].

**Théorème 13.** *Un AAFAD acyclique* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *peut être minimisé utilisant* $M_{\mathcal{A}} = (Q', \sigma, \{i_s\}, F, \widehat{\delta})$ *en* $\mathcal{O}(\hat{r}|\mathcal{A}|)$.

En effet, la construction de l'automate associé se fait en $\mathcal{O}(\hat{r}|\mathcal{A}|)$ et la minimisation acyclique pour les mots est linéaire et coûte $\mathcal{O}(|\mathcal{A}|)$. De ce fait, nous avons $\mathcal{O}(\hat{r}|\mathcal{A}| + |\mathcal{A}|) = \mathcal{O}(\hat{r}|\mathcal{A}|)$.

#### 4.2.4.2 Minimisation incrémentale

Nous avons vu que l'implémentation de la minimisation incrémentale pour les AAFAD présente une complexité asymptotique d'ordre exponentiel. Néanmoins, le passage à l'automate associé de mots permet de bénéficier des algorithmes présents. En effet, rappelons que l'algorithme d'Almeida et al. [AMR10] et celui de Garcia et al. [GVVL15] ont la meilleur complexité asymptotique qui est de l'ordre quadratique.

**Définition 4.12.** *Soit* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *un AAFAD et* $M_{\mathcal{A}} = (Q', \sigma, \{i_s\}, F, \widehat{\delta})$ *son automate associé. Nous étendons la fonction de transition* $\widehat{\delta}$ *par* $\widehat{\delta}'$ *comme suit :* $\widehat{\delta}'(q, a) = \widehat{\delta}(q, a)$ *où* $a \in \sigma$ *et* $\widehat{\delta}'(q, ax) = \widehat{\delta}'(\widehat{\delta}(q, a), x)$ *avec* $ax \in \sigma^+$. *Nous définissons le* langage à droite *pour un état* $q \in Q$ *comme suit :* $\overrightarrow{L}(q) = \{x \in \sigma^+, \widehat{\delta}'(q, x) \in F\}$.

Par conséquent :

**Lemme 4.10.** *Soient* $p, q \in Q$. *Alors,* $L^{\uparrow}(p) = L^{\uparrow}(q) \Leftrightarrow \overrightarrow{L}(p) = \overrightarrow{L}(q)$.

En utilisant ce résultat, on peut calculer $equiv(p, q)$ dans le nouvel automate $M_{\mathcal{A}}$ au lieu de calculer cette fonction dans l'automate d'arbres.

Le calcul récursif de $equiv$ dans l'automate associé est présenté dans l'équation 4.9.

$$equiv(p, q) = \bigwedge_{a \in \sigma} (equiv(\widehat{\delta}(p, a), \widehat{\delta}(q, a)) \wedge ((p \in F) \Leftrightarrow (q \in F)) \tag{4.9}$$

Maintenant, en utilisant l'algorithme de Garcia et al. [GVVL15], la complexité globale de la minimisation incrémentale est :

**Théorème 14.** *Un AAFAD* $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ *peut être minimisé moyennant son automate associé* $M_{\mathcal{A}} = (Q', \sigma, \{i_s\}, F, \widehat{\delta})$ *en* $\mathcal{O}(\hat{r}|\mathcal{A}| + |\mathcal{A}||Q|^2) = \mathcal{O}(|\mathcal{A}||Q|^2)$ .

#### 4.2.4.3 Construction incrémentale d'un AAFAD

Maintenant, nous discutons la possibilité d'utiliser l'automate associé pour effectuer la construction incrémentale.

**Algorithme 17** Nouvelle fonction split

**Fonction** split(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, $M_{\mathcal{A}}$, arbre $f(s_1, \ldots, s_m)$, ensemble d'états $\Theta$ )
  **pour tout** $k = 1 \ldots$ *jusqu'a* $m$ **faire**
    $\mathcal{A} \leftarrow \text{split}(\mathcal{A}, \Theta, s_k)$
    $r_k \leftarrow \delta(s_k)$
  **Fin pour**
  $q \leftarrow \delta(f, r_1, \ldots, r_m)$
  **si** $C_q \neq 1$ **alors**
    $\text{create}(n)$
    $Q \leftarrow Q \cup \{n\}$
    **si** $C_q = 0$ **alors**
      $L_\Delta \leftarrow L_\Delta \cup L_{(f,r_1,\ldots,r_m)}$ ▷
    **sinon**
      $\mathcal{A} \leftarrow \text{Clone}(\mathcal{A}, q, n)$
      **pour tout** $(p, a) \rightarrow q \in \widehat{\delta}$ **faire**
        $\text{Replace}(\widehat{\delta}, (p, a) \rightarrow q, (p, a) \rightarrow n)$
      **Fin pour**
    **Fin si**
    $\Theta \leftarrow \Theta \cup \{r_1, \ldots, r_n\}$
  **Fin si**
**Fin Fonction**

**Algorithme 18** Nouvelle fonction clone

**Fonction** clone(AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$, $M_{\mathcal{A}}$, état $q$, état $n$)
  **si** $q \in F$ **alors**
    $Add(F, n)$
  **Fin si**
  **tanque** il existe $m > 0, k \leq m, q_1, \ldots, q_m \in Q$ tel que $(q, f_{q_1 \ldots q_{k-1}, q_{k+1} \ldots, q_{m-1}}) \rightarrow q_m \in \widehat{\delta}$ **faire**
    $\widehat{\delta} \leftarrow \widehat{\delta} \cup \{(n, f_{q_1 \ldots q_{k-1}, q_{k+1} \ldots, q_{m-1}}) \rightarrow q_m))$
    $L_\Delta \leftarrow L_\Delta \cup (f, q_1, \ldots, q_{k-1}, n, q_{k+1} \ldots, q_m\}$
  **Fin tanque**
**Fin Fonction**

Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD minimal et $M_{\mathcal{A}} = (Q', \sigma, \{i_s\}, F, \widehat{\delta})$ son automate associé. Soit $t \notin L(\mathcal{A})$ l'arbre à ajouter. En prenant en compte que $M_{\mathcal{A}}$ n'est pas conçu pour reconnaître des séquences, il ne peut pas être utilisé pour déterminer les états concernés par les changements. Ceci est le résultat de la perte de certaines transitions lors de sa construction. Pour cette raison, il est inévitable de faire l'identification moyennant $\mathcal{A}$. Nous rappelons que la relation $\sim$ doit être maintenue lors des changements pour des éventuels ajouts ou suppressions.

Nous présentons la version adaptée dans Algorithme 17. En effet, les étapes se résument dans ce qui suit :

Pour chaque sous arbre $s \in St(t)$ :

- Si la sortie de $\delta(s)$ n'est pas définie alors on peut ajouter sans risque un état $n$ tel que $L^{\downarrow}(n) = s$.
- S'il existe un état $q \in Q$ tel que $\delta(s) = q$ alors nous rencontrons deux cas :
  1. si $C_q$=1 alors, l'automate est inchangé.
  2. sinon, ($C_q > 1$) nous ajoutons un état *clone* $n$ (voir Algorithme 18) dans $M_{\mathcal{A}}$ comme suit : ajouter $nq_1 \dots q_{i-1} \bullet q_{i+1} \dots q_n$ à $L_{\Delta}$ tel que $qq_1 \dots q_{i-1} \bullet q_{i+1} \dots q_n \in L_{\Delta}$. Ici, $\sigma_n = \sigma_q$.

Ensuite, $\sim$ est mis à jours et $\overline{Q}$ est recalculé. Finalement, $M_{\mathcal{A}}$ est résolu comme suit :

1. enlever de $L_{\Delta}$ toutes les séquences de la forme $pq_1 \dots q_i \bullet q_{i+1} \dots q_n$ où $\delta((f, q_1, \dots, q_i, p, q_{i+1}, \dots, q_n)) = q$ pour tout état $q$ qui procède des clones.
2. recalculer $\overline{Q}$ et $M_{\mathcal{A}}$.

Enfin, la même minimisation locale définie pour les automates de mots [CF02] est appliquée à l'automate associé.

Notons que la fonction $\mathrm{Replace}$ remplace une instance donnée par une autre dans l'ensemble correspondant.

Concernant la complexité, le calcul des états clones ne peut pas être évité. Cette étape coûte $\mathcal{O}(|\Delta|2^{\hat{r}})$. Le recalcule de $\overline{Q}$ nécessite $\mathcal{O}(|\mathcal{A}|)$. Quant à la minimisation locale, elle peut être effectuée en $\mathcal{O}(|\mathcal{A}|)$ ($t$ est un arbre fini).

**Théorème 15.** *Soit $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$ un AAFAD minimal et $M_{\mathcal{A}} = (Q', \sigma, \{i_s\}, F, \widehat{\delta})$ son automate associé. Donc, l'automate minimal reconnaissant $L(\mathcal{A} \cup \{t\})$ où $t \in T_{\Sigma}$ est construit en $\mathcal{O}(|\Delta|2^{\hat{r}} + \hat{r}|\mathcal{A}|)$.*

En effet, le calcul des clone coûte $\mathcal{O}(|\Delta|2^{\hat{r}})$ et la minimisation locale se fait en $\mathcal{O}(\hat{r}|\mathcal{A}|)$. Tenant en compte la construction de l'automate associé, la complexité globale est $\mathcal{O}(|\Delta|2^{\hat{r}} + \hat{r}|\mathcal{A}| + |\mathcal{A}|) = \mathcal{O}(|\Delta|2^{\hat{r}} + \hat{r}|\mathcal{A}|)$.

## 4.3 Conclusion

Dans ce chapitre, nous avons présenté nos contributions au profit de la minimisation d'AAFAD. En effet, nous avons défini deux méta-méthodes pour cette fin : la première est basée sur une vue graphique de l'automate à minimiser et la seconde est basée sur la transformation de l'automate en question en un automate de mots dont sa minimisation coïncide avec celle des arbres. Nous avons montré que ces deux techniques améliorent considérablement la complexité asymptotique globale de différents types de minimisation.

Les améliorations apportées par les méthodes proposées sont résumées dans le tableau suivant pour un AAFAD $\mathcal{A} = (Q, \Sigma, Q_f, \Delta)$.

| **Techniques de minimisation** | **Complexité existante** | **Méthode directe** | **Méthode indirecte** |
|---|---|---|---|
| Standard | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert \log(\lvert Q\rvert))$ | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert + \lvert\mathcal{A}\rvert \log \lvert Q\rvert)$ | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert + \lvert\mathcal{A}\rvert \log \lvert Q\rvert)$ |
| Acyclique | - | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert \log(\lvert Q\rvert))$ | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert)$ |
| Incrémentale | $\mathcal{O}(\lvert\mathcal{A}\rvert^{\lvert Q\rvert-2}\lvert Q\rvert^2)$ | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert + \lvert Q\rvert^3 \log(\lvert Q\rvert))$ | $\mathcal{O}(\hat{r} \times \lvert\mathcal{A}\rvert + \lvert\mathcal{A}\rvert \times \lvert Q\rvert^2)$ |
| Construction incrémentale | $\mathcal{O}(\lvert\mathcal{A}\rvert \times 2^{\hat{r}} + \lvert\mathcal{A}\rvert)$ | - | $\mathcal{O}(\lvert\mathcal{A}\rvert \times 2^{\hat{r}} + \lvert\mathcal{A}\rvert)$ |

TABLE 4.1: Comparatif de complexités entre les techniques existantes et proposées

Un autre créneau très important pour dénoter un motif d'arbre est l'étude des expressions rationnelles. L'objectif du prochain chapitre est d'étudier la relation entre les ERA et les AAFA ainsi que de proposer des constructions de passage entre les deux structures.

# Chapitre 5

# Automates d'arbres et expressions rationnelles d'arbres

Dans cette partie, nous étudions la relation entre les automates d'arbres et les expressions rationnelles. Sachant que les expressions rationnelles sont généralement utilisées comme un motif (pattern) qui sert à dénoter un langage d'arbre dit *rationnel*, une relation d'équivalence entre la classe des langages rationnelles et des langages reconnaissables peut être bien définie.

Nous pouvons citer comme exemple illustratif d'utilisation des expressions rationnelles pour les arbres (ERA) la recherche de motifs d'arbres qui est un domaine assez ancien [Kos89, AG85]. L'usage des ERA pour dénoter un motif est une nouvelle tendance qui a été discutée dans [Cle08, Bel14]. L'importance de cet usage est cruciale car elle permet de révolutionner la recherche dans les différentes structure arborescente tel que les documents XMLs [ZL03], la vérification formelle et la cryptographie [DH04] etc.

Le but de ce chapitre est de discuter l'efficacité de l'usage des expressions rationnelle d'arbres pour dénoter des grandes masses de données (dans langages réguliers infinis par exemple). Nous allons discuter la complexité des algorithmes de traitement des ERA et le passage entre un automate d'arbres vers une ERA.

Comme mentionné dans les chapitres précédents, les automates d'arbres sont bénéfiques pour reconnaître un langage d'arbres, quant aux ERA, elles sont orientées plus vers un point de vue descriptif. En effet, Kleene a prouvé que les ER et les automates de mots sont équivalents [Kle64]. Ce résultat fut généralisé et prouvé pour les arbres (voir Comon et al. [CDG+07]). En effet, le résultat de Kleene permet deux constructions : le passage d'une ERA vers un AAF et l'inverse (voir la Figure 5.1).

Quant au passage d'une ERA vers un AAF, plusieurs techniques existent. Premièrement, Kuske et Meinecke [KM11] ont généralisé la notion des dérivations des langages [Ant96] aux arbres et ont proposé un automate d'arbres d'équations qui est construit à partir d'une version linéarisée de l'ERA à transformer. Il ont utilisé pour cette raison la structure ZPC proposée par Champarnaud et al. [CZ01] pour atteindre une meilleure complexité. Après, Mignot et al. [MSZ14a] ont proposé un algorithme efficace qui améliore le temps de calcul de cet automate. Quant aux automates de positions, après avoir défini les notions de First, Follow et Last pour les ERA, Laugerotte et al. [LSZ13]

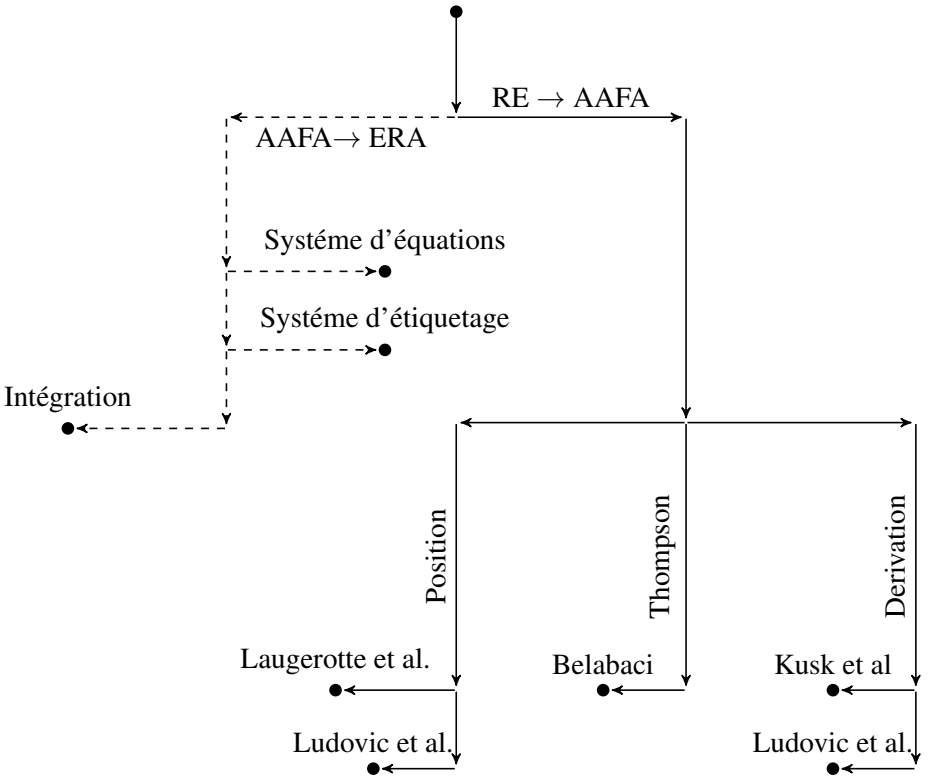



FIGURE 5.1: Passage ERA ↔ AAFA

ont généralisé cette notion aux arbre . Récemment, Bellabaci et al. [Bel14] ont proposé un passage des ERA vers les AAFA qui est une généralisation de l'automate de Thompson pour les mots.

Contrairement à ces propositions, très peu de travaux se sont intéressé au passage depuis un AAFA vers une ERA. Dans ce chapitre, nous généralisons trois techniques de passage d'un automate vers une ERA (partie pointillée dans la Figure 5.1). La première utilise une reformulation du lemme d'Arden [Con61] pour les arbres afin de solutionner un système d'équations rationnelle d'arbres extrait à partir de l'automate en question. La deuxième technique est une technique simple d'élimination d'états (généralisation d'une techniques similaire sur les mots connus sous le nom de McNAughton-Yamada [MY60]) visant à remplacer chaque état par une ERA équivalente. Quant à la dernière, elle généralise la notion d'intégrales définie par Smith et Yau [SY72], qui est l'opération inverse de la notion des dérivées partielles.

Les trois premières sections sont dédiées aux contributions proposées de passage d'un AAFA vers une ERA. Une dernière section présente un algorithme de transformation générique basé sur une heuristique qui permet d'atteindre une meilleure ERA en termes de longueur lexicographique.

## 5.1 Méthode de résolution de système d'équations rationnelles

La méthode consiste à extraire depuis un AAFA un système d'équations rationnelles équivalent. Ce système est obtenu en définissant pour chaque état de l'automate une équation rationnelle qui dénote sont langage accepté. Ce système est ensuite résolu en définissant un système de substitutions

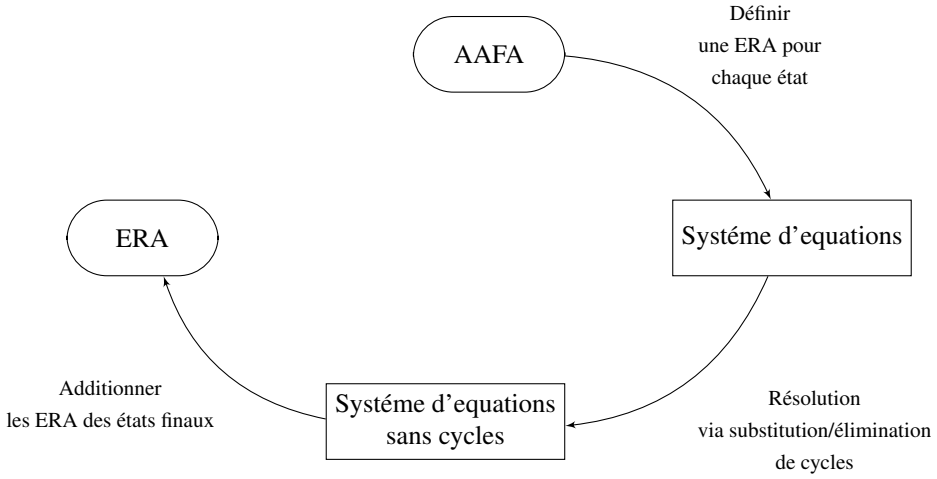



FIGURE 5.2: Transformation d'un AAFA en une ERA via systéme d'équations rationnelles

efficace. Enfin, l'ERA cherchée est obtenue en sommant les ERA des états finaux de l'AAFA. La Figure 5.2 illustre les étapes de cette approche.

Mais avant de montrer comment extraire un système d'équations rationnelles depuis un AAFA, nous présentons d'une manière plus générale la notion des systèmes d'équations rationnelles et nous fixons une condition sous laquelle ces systèmes peuvent avoir une solution dans $\mathrm{Rat}(\Sigma)$. Pour cette raison, nous augmentons la définition des ERA par des *variables*.

### 5.1.1 Expression rationnelle d'arbres avec variables

Dans ce qui suit, nous considérons des expressions rationnelles avec des *variables*. Soit $X = \{x_1, \ldots, x_k\}$ un ensemble de $k$ variables. Une expression rationnelle $E$ à travers $(\Sigma, X)$ est définie par induction comme suit :

$$E = 0, \quad E = x_j, \quad E = f(E_1, \ldots, E_n),$$
$$E = E_1 + E_2, \quad E = E_1 \cdot_c E_2, \quad E = E_1^{*_c}$$

où $f \in \Sigma_n$, $c \in \Sigma_0$, $1 \leq j \leq k$ est un entier naturel et$E_1, \ldots, E_n$ sont $n$ ERA à travers $(\Sigma, X)$.

La sémantique d'une ERA avec variables a besoin d'un "contexte" pour être calculée. Donc, chaque variable doit être évaluée par rapport à un langage d'arbres.

Soit $\mathcal{L} = (L_1, \ldots, L_k)$ un $k$-tuple de langages d'arbres à travers $\Sigma$. La sémantique de $E$ est le langage $L_{\mathcal{L}}(E)$ défini par induction comme suit :

$$L_{\mathcal{L}}(0) = \emptyset, \quad L_{\mathcal{L}}(x_j) = L_j,$$
$$L_{\mathcal{L}}(f(E_1, \ldots, E_n)) = f(L_{\mathcal{L}}(E_1), \ldots, L_{\mathcal{L}}(E_n)),$$
$$L_{\mathcal{L}}(E_1 + E_2) = L_{\mathcal{L}}(E_1) \cup L_{\mathcal{L}}(E_2)$$
$$L_{\mathcal{L}}(E_1 \cdot_c E_2) = L_{\mathcal{L}}(E_1) \cdot_c L_{\mathcal{L}}(E_2), \quad L_{\mathcal{L}}(E_1^{*_c}) = (L_{\mathcal{L}}(E_1))^{*_c}$$

où $f \in \Sigma_n$, $c \in \Sigma_0$, $1 \leq j \leq k$ est un entier naturel et $E_1, \ldots, E_n$ sont $n$ ERA à travers $(\Sigma, X)$.

Deux ERA $E$ et $F$ avec variables sont *équivalentes* si pour chaque tuple $\mathcal{L}$ de langages à travers $\Sigma$ nous avons $L_{\mathcal{L}}(E) = L_{\mathcal{L}}(F)$. L'équivalence est notée $E \sim F$. Soit $\Gamma \subset \Sigma$. Deux ERA $E$ et $F$ avec variables sont $\Gamma$-*équivalentes*, (notée $E \sim_\Gamma F$), si pour chaque tuple $\mathcal{L}$ de langages à travers $\Gamma$, $L_{\mathcal{L}}(E) = L_{\mathcal{L}}(F)$.

Par définition, nous avons

$$E \sim F \Rightarrow E \sim_\Gamma F \tag{5.1}$$

Notons qu'une ERA à travers $(\Sigma, X)$ est aussi une ERA à travers $\Sigma \cup X$. Mais l'inverse n'est pas toujours vrai.

**Exemple 5.1.** *L'expression $x \cdot_a b$ est équivalent à $x$ comme ERA à travers $\{a, b, x\}$, mais non pas à travers $(\{a, b\}, \{x\})$ :*

$$L(x \cdot_a b) = \{x\} = L(x)$$
$$L_{\{a\}}(x \cdot_a b) = \{b\} \neq L_{\{a\}}(x) = \{a\}$$

Nous définissons maintenant l'opération de *substitution* de variables. Cette opération est très importante pour la résolution d'un système d'équations rationnelles.

$E_{x \leftarrow E'}$ est l'ERA obtenue en substituant une variable $x$ par une expression $E'$ dans $E$. Cette opération est définie par induction comme suit :

**Définition 5.1.** *(Substitution)*

$$a_{x \leftarrow E'} = a \quad 0_{x \leftarrow E'} = 0$$
$$y_{x \leftarrow E'} = y \quad x_{x \leftarrow E'} = E'$$
$$(f(E_1, \ldots, E_n))_{x \leftarrow E'} = f((E_1)_{x \leftarrow E'}, \ldots, (E_n)_{x \leftarrow E'})$$
$$(E_1 + E_2)_{x \leftarrow E'} = (E_1)_{x \leftarrow E'} + (E_2)_{x \leftarrow E'} \quad (E_1 \cdot_c E_2)_{x \leftarrow E'} = (E_1)_{x \leftarrow E'} \cdot_c (E_2)_{x \leftarrow E'}$$
$$(E_1^{*_c})_{x \leftarrow E'} = ((E_1)_{x \leftarrow E'})^{*_c}$$

*ou $a \in \Sigma_0$, $x \neq y$ sont deux variables dans $X$, $f \in \Sigma_n$, $c \in \Sigma_0$ et $E_1, \ldots, E_n$ sont $n$ ERA à travers $(\Sigma, X)$.*

La substitution préserve le langage sous la condition suivante :

**Lemme 5.1.** *Soient $E, F$ une ERA à travers $\Sigma$ et sous l'ensemble de variables $X = \{x_1, \ldots, x_n\}$. Soit $x_j \in X$. Soit $\mathcal{L} = (L_1, \ldots, L_n)$ un $n$-uple de langages tel que $L_j = L_{\mathcal{L}}(F)$. Alors :*

$$L_{\mathcal{L}}((E)_{x_j \leftarrow F}) = L_{\mathcal{L}}(E)$$

*Démonstration.* Par induction sur la structure de $E$.

1. Si $E \in \{a, y, 0\}$ avec $a \in \Sigma_0$ et $y \neq x_j$, $(E)_{x_j \leftarrow F} = E$.

2. Si $E = x_j$, alors $(E)_{x_j \leftarrow F} = F$. Donc

$$\begin{aligned} L_{\mathcal{L}}((E)_{x_j \leftarrow F}) &= L_{\mathcal{L}}(F) \ = L_j \\ &= L_{\mathcal{L}}(x_j) = L_{\mathcal{L}}(E) \end{aligned}$$

3. Si $E = f(E_1, \ldots, E_n)$, avec $f \in \Sigma_k$, $k > 0$ alors :

$$\begin{aligned} L_{\mathcal{L}}((E)_{x_j \leftarrow F}) &= L_{\mathcal{L}}(f((E_1)_{x_j \leftarrow F}, \ldots, (E_n)_{x_j \leftarrow F})) \\ &= f(L_{\mathcal{L}}((E_1)_{x_j \leftarrow F}), \ldots, L_{\mathcal{L}}((E_n)_{x_j \leftarrow F})) \\ &= f(L_{\mathcal{L}}(E_1), \ldots, L_{\mathcal{L}}(E_n)) && \text{(Hypothèse d'induction)} \\ &= L_{\mathcal{L}}(f(E_1, \ldots, E_n)) \end{aligned}$$

4. Si $E = E_1 + E_2$, alors

$$\begin{aligned} L_{\mathcal{L}}((E_1 + E_2)_{x_j \leftarrow F}) &= L_{\mathcal{L}}((E_1)_{x_j \leftarrow F} + (E_2)_{x_j \leftarrow F}) \\ &= L_{\mathcal{L}}((E_1)_{x_j \leftarrow F}) \cup L_{\mathcal{L}}((E_2)_{x_j \leftarrow F})) \\ &= L_{\mathcal{L}}(E_1) \cup L_{\mathcal{L}}(E_2) && \text{(Hypothèse d'induction)} \\ &= L_{\mathcal{L}}(E_1 + E_2) \end{aligned}$$

5. Si $E = E_1 \cdot_c E_2$, alors

$$\begin{aligned} L_{\mathcal{L}}((E_1 \cdot_c E_2)_{x_j \leftarrow F}) &= L_{\mathcal{L}}((E_1)_{x_j \leftarrow F} \cdot_c (E_2)_{x_j \leftarrow F}) \\ &= L_{\mathcal{L}}((E_1)_{x_j \leftarrow F}) \cdot_c L_{\mathcal{L}}((E_2)_{x_j \leftarrow F})) \\ &= L_{\mathcal{L}}(E_1) \cdot_c L_{\mathcal{L}}(E_2) && \text{(Hypothèse d'induction)} \\ &= L_{\mathcal{L}}(E_1 \cdot_c E_2) \end{aligned}$$

6. Si $E = E_1^{*_c}$, alors

$$\begin{aligned} L_{\mathcal{L}}((E_1^{*_c})_{x_j \leftarrow F}) &= (L_{\mathcal{L}}((E_1)_{x_j \leftarrow F}))^{*_c} \\ &= (L_{\mathcal{L}}(E_1))^{*_c} && \text{(Hypothèse d'induction)} \\ &= L_{\mathcal{L}}(E_1^{*_c}) \end{aligned}$$

□

Dans ce qui suit, $\mathrm{op}(E)$ est l'ensemble des opérations rationnelles qui apparaissent dans $E$.

Néanmoins, l'opération de substitution préserve le langage s'il est basé sur un alphabet restreint :

**Proposition 5.1.** *Soit $E$ une ERA à travers $\Sigma$ et un ensemble $X$ de variables. Soit $x \in X$. Soit $\Gamma \subset \Sigma$ tel que $\Gamma = \{b \in \Sigma_0 \mid \{\cdot_b, {}^{*_b}\} \cap \mathrm{op}(E) \neq \emptyset\}$. Soit $a \notin \Sigma$. Alors :*

$$E \sim_{\Sigma \setminus \Gamma} (E)_{x \leftarrow a \cdot_a x}$$

### 5.1.2 Système d'équations rationnelles

Dans cette section, nous discutons les systèmes d'équations rationnelles d'arbres. Dans la littérature, plusieurs références ont défini et utilisé cette notion pour la résolutions de plusieurs problèmes comme [CDG+07, ES77].

Ici, nous redéfinissons les systèmes d'équations et nous proposons une méthode qu'on va appeler *Gauss elimination-like* qui donne une solution rationnelle à un système d'équations.

Soit $\Sigma$ un alphabet et $\mathbb{E} = \{\mathbb{E}_1, \ldots, \mathbb{E}_n\}$ un ensemble de $n$ variables. Une *equation* à travers $(\Sigma, \mathbb{E})$ est l'expression $\mathbb{E}_j = F_j$, où $1 \leq j \leq n$ et $F_j$ est une ERA à travers $(\Sigma, \mathbb{E})$. Un *système d'équations* à travers $(\Sigma, \mathbb{E})$ est l'ensemble $\mathcal{X} = \{\mathbb{E}_j = F_j \mid 1 \leq j \leq n\}$ de $n$ équations. Soit $\mathcal{L} = (L_1, \ldots, L_n)$ un $n$-tuple de langages d'arbres. Le tuple $\mathcal{L}$ est une *solution* pour une équation donnée $(\mathbb{E}_j = F_j)$ si $L_j = L_{\mathcal{L}}(F_j)$. Le tuple $\mathcal{L}$ est une *solution* pour $\mathcal{X}$ si pour chaque équation $(\mathbb{E}_j = F_j)$ dans $\mathcal{X}$, $\mathcal{L}$ est une solution de $(\mathbb{E}_j = F_j)$.

**Exemple 5.2.** *Soit $\mathcal{X}$ le système d'équations suivant :*

$$\mathcal{X} = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, \mathbb{E}_4) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, \mathbb{E}_4) \\ \mathbb{E}_3 & = a + h(\mathbb{E}_4) \\ \mathbb{E}_4 & = a + h(\mathbb{E}_3) \end{cases}$$

*Le tuple $(\emptyset, \emptyset, \emptyset, \emptyset)$ est une solution de l'équation $\mathbb{E}_1 = F_1$, mais pas pour le système $\mathcal{X}$.*

Deux systèmes à travers le même ensemble de variables sont *équivalents* s'ils admettent les mêmes solutions.

Évidement,

**Proposition 5.2.** *Si $\mathcal{X}$ contient seulement des équations $\mathbb{E}_k = F_k$ avec $F_k$ une ERA sans variables, alors $(L(F_1), \ldots, L(F_n))$ est l'unique solution de $\mathcal{X}$.*

Nous définissons l'opération de substitution sur un système d'équations :

**Définition 5.2.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations. La* substitution *de $(\mathbb{E}_k = F_k)$ dans $\mathcal{X}$ est le système $\mathcal{X}^k = \{\mathbb{E}_k = F_k\} \cup \{\mathbb{E}_j = (F_j)_{\mathbb{E}_k \leftarrow F_k} \mid j \neq k \wedge 1 \leq j \leq n\}$.*

Utilisant le lemme 5.1,

**Proposition 5.3.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations. Soit $\mathbb{E}_k = F_k$ une équation dans $\mathcal{X}$. Soit $\mathcal{L}$ une solution de $\mathcal{X}$. Alors, pour chaque $1 \leq j, k \leq n$ avec $j \neq k$,*

$$\mathcal{L} \text{ est une solution de } \mathbb{E}_j = (F_j)_{\mathbb{E}_k \leftarrow F_k}.$$

Ainsi,

**Proposition 5.4.** *Soit $\mathcal{X}$ un système d'équations avec $n$ variables. Soit $k \leq n$. Alors :*

$$\mathcal{X} \text{ et } \mathcal{X}^k \text{ sont équivalents.}$$

**Exemple 5.3.** *Soit le système $\mathcal{X}$ utilisé dans l'exemple 5.2. Alors :*

$$\mathcal{X}^4 = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_3 & = a + h(a + h(\mathbb{E}_3)) \\ \mathbb{E}_4 & = a + h(\mathbb{E}_3) \end{cases}$$

Nous déterminons un cas particulier où un système peut être directement résolu via une succession de substitutions :

**Définition 5.3.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations. La relation $<_\mathcal{X}$ est définie pour deux variables $\mathbb{E}_j$ et $\mathbb{E}_k$ par*

$$(\mathbb{E}_j <_\mathcal{X} \mathbb{E}_k) \Leftrightarrow (\mathbb{E}_j \textit{ figure dans } F_k)$$

*La relation $\preceq_\mathcal{X}$ est la fermeture transitive de $<_\mathcal{X}$. Dans le cas où $\mathbb{E}_k <_\mathcal{X} \mathbb{E}_k$ , l'équation $\mathbb{E}_k = F_k$ est* récursive*.*

*On dit qu'un système est* récursif *s'il existe $\mathbb{E}_j$ et $\mathbb{E}_k$ tel que $\mathbb{E}_j \preceq_\mathcal{X} \mathbb{E}_k$ et $\mathbb{E}_k \preceq_\mathcal{X} \mathbb{E}_j$. Si un système n'est pas récursif, il peut être résolu via une succession de substitutions.*

*Si $\mathbb{E}_k$ ne figurent pas dans la partie droite de n'importe quelle équation $\mathcal{X}$, alors le système $\mathcal{X} \setminus (\mathbb{E}_k = F_k)$ est obtenu en supprimant $\mathbb{E}_k = F_k$ de $\mathcal{X}$, et en ré-indexant tout symbole dans $\mathbb{E}_j$ par $j > k$ dans $\mathbb{E}_{j-1}$.*

Par conséquent :

**Lemme 5.2.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations à travers $\Sigma$ et $n$ variables $\{\mathbb{E}_1, \ldots, \mathbb{E}_n\}$. Soit $\mathbb{E}_k = F_k$ une équation dans $\mathcal{X}$ tel que $\mathbb{E}_k = F_k$ n'est pas récursive. Alors pour chaque $n-1$-tuple $Z = (L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n)$, les deux déclarations suivantes sont équivalentes :*

1. *$(L_1, \ldots, L_{k-1}, L_Z(F_k), L_{k+1}, \ldots, L_n)$ est une solution de $\mathcal{X}$*
2. *$(L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n)$ est une solution de $\mathcal{X}^k \setminus \{\mathbb{E}_k = F_k\}$*

Donc, un système non-récursif peut être résolu en se référant à un système plus petit obtenu par substitution :

**Corollaire 5.1.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations à travers $\Sigma$ et $n$ variables $\{\mathbb{E}_1, \ldots, \mathbb{E}_n\}$. Soit $\mathbb{E}_k = F_k$ une équation dans $\mathcal{X}$ tel que $F_k$ est une ERA à travers $\Sigma$. Alors, pour chaque $n-1$-tuple $(L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n)$, les deux déclarations suivantes sont équivalentes :*

1. *$(L_1, \ldots, L_{k-1}, L(F_k), L_{k+1}, \ldots, L_n)$ est une solution pour $\mathcal{X}$*
2. *$(L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n)$ est une solution pour $\mathcal{X}^k \setminus \{\mathbb{E}_k = F_k\}$*

De plus, un tel système admet une solution unique :

**Proposition 5.5.** *Soit* $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ *un système non récursif à travers* $\Sigma$ *et* $\{\mathbb{E}_1, \ldots, \mathbb{E}_n\}$. *Alors*

$$\mathcal{X} \text{ admet une solution unique}.$$

*Démonstration.* Par récurrence sur le cardinal de $\mathcal{X}$.

1. $\mathcal{X} = \{\mathbb{E}_1 = F_1\}$, alors $F_1$ est une ERA à travers $\Sigma$ (sans variables) donc $L(F_1)$ est la solution unique de $\mathcal{X}$
2. Comme $\mathcal{X}$ est non récursif, il existe une équation $\mathbb{E}_k = F_k$ avec $F_k$ une ERA à travers $\Sigma$ (sans variables). Par conséquent, suivant le corollaire 5.1, un tuple $(L_1, \ldots, L_{k-1}, L(F_k), L_{k+1}, \ldots, L_n)$ est une solution pour $\mathcal{X}$ si et seulement si $(L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n)$ est une solution pour $\mathcal{X}^k \setminus \{\mathbb{E}_k = F_k\}$. Suivant l'hypothèse de récurrence, $\mathcal{X}^k \setminus \{\mathbb{E}_k = F_k\}$ est non récursif et admet donc une solution unique $(L_1, \ldots, L_{k-1}, L_{k+1}, \ldots, L_n)$. Donc, $(L_1, \ldots, L_{k-1}, L(F_k), L_{k+1}, \ldots, L_n)$ est une solution pour $\mathcal{X}$. Enfin, $L_k \neq L(F_k)$,le tuple $(L_1, \ldots, L_{k-1}, L_k, L_{k+1}, \ldots, L_n)$ n'est pas une solution pour $\mathbb{E}_k = F_k$ et $(L_1, \ldots, L_{k-1}, L(F_k), L_{k+1}, \ldots, L_n)$ représente l'unique solution pour $\mathcal{X}$.

□

**Exemple 5.4.** *Soit* $\mathcal{Y}$ *le système d'équations suivant :*

$$\mathcal{Y} = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_2, \mathbb{E}_3) + f(\mathbb{E}_2, \mathbb{E}_3) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_4, \mathbb{E}_4) \\ \mathbb{E}_3 & = a + h(\mathbb{E}_4) \\ \mathbb{E}_4 & = a + (f(a,b))^{*_b} \cdot_b a \end{cases}$$

*Alors*

$$\mathcal{Y}^4 = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_2, \mathbb{E}_3) + f(\mathbb{E}_2, \mathbb{E}_3) \\ \mathbb{E}_2 & = b + f(a + (f(a,b))^{*_b} \cdot_b a, a + (f(a,b))^{*_b} \cdot_b a) \\ \mathbb{E}_3 & = a + h(a + (f(a,b))^{*_b} \cdot_b a) \\ \mathbb{E}_4 & = a + (f(a,b))^{*_b} \cdot_b a \end{cases}$$

$$(\mathcal{Y}^4)^3 = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_2, a + h(a + (f(a,b))^{*_b} \cdot_b a)) + f(\mathbb{E}_2, a + \\ & \quad h(a + (f(a,b))^{*_b} \cdot_b a)) \\ \mathbb{E}_2 & = b + f(a + (f(a,b))^{*_b} \cdot_b a, a + (f(a,b))^{*_b} \cdot_b a) \\ \mathbb{E}_3 & = a + h(a + (f(a,b))^{*_b} \cdot_b a) \\ \mathbb{E}_4 & = a + (f(a,b))^{*_b} \cdot_b a \end{cases}$$

$$((\mathcal{Y}^4)^3)^2 = \begin{cases} \mathbb{E}_1 & = f(b + f(a + (f(a,b))^{*_b} \cdot_b a, a + (f(a,b))^{*_b} \cdot_b a), a + \\ & \quad h(a + (f(a,b))^{*_b} \cdot_b a)) + f(b + f(a + (f(a,b))^{*_b} \cdot_b a, a + \\ & \quad (f(a,b))^{*_b} \cdot_b a), a + h(a + (f(a,b))^{*_b} \cdot_b a)) \\ \mathbb{E}_2 & = b + f(a + (f(a,b))^{*_b} \cdot_b a, a + (f(a,b))^{*_b} \cdot_b a) \\ \mathbb{E}_3 & = a + h(a + (f(a,b))^{*_b} \cdot_b a) \\ \mathbb{E}_4 & = a + (f(a,b))^{*_b} \cdot_b a \end{cases}$$

### 5.1.3 Système d'équations récursif

Par analogie au cas de mots, l'élimination du cas récursif est inévitable. En effet, la récurrence doit être enlevée afin d'éviter des substitutions infinies.

Le lemme d'Arden [Con61] est un résultat fondamental en théorie des automates de mots. Ce lemme permet de trouver une solution pour une équation récursive de la forme $X = A \cdot X \cup B$ où $X$ est le langage inconnu. Ce résultat permet d'identifier une expression rationnelle de mots. En effet, des résultats similaires sont connus pour les arbres. Par exemple, Bozapalidis [Boz99] a proposé une solution récursive dans une algèbre qui contient la classe des langages d'arbres.

Dans la proposition suivante, nous généralisons ce lemme pour les langages d'arbres reconnaissables :

**Proposition 5.6.** *Soient $A$ et $B$ deux langages à travers $\Sigma$. Alors $A^{*_c} \cdot_c B$ est la plus petite solution dans la famille $\mathcal{F}$ des langages $L$ à travers $\Sigma$ satisfaisant $L = A \cdot_c L \cup B$. En outre, si $c \notin A$, alors $\mathcal{F} = \{A^{*_c} \cdot_c B\}$.*

*Démonstration.* Nous mettons $Z = A^{*_c} \cdot_c B$.

1. Clairement, $Z \in \mathcal{F}$ :

$$\begin{aligned} A \cdot_c (A^{*_c} \cdot_c B) \cup B &= (A \cdot_c A^{*_c}) \cdot_c B \cup B && \text{utilisant le Corollaire 1.1} \\ &= (A \cdot_c A^{*_c}) \cdot_c B \cup \{c\} \cdot_c B \\ &= ((A \cdot_c A^{*_c}) \cup \{c\}) \cdot_c B \\ &= A^{*_c} \cdot_c B \end{aligned}$$

2. Montrons que si on pose $C \in \mathcal{F}$, alors $Z \subset C$. Pour cette raison, nous montrons que pour $n \geq 0$, $A^{n,c} \cdot_c B \subset C$. Comme $C \in \mathcal{F}$, alors $C = A \cdot_c C \cup B$. Par conséquent, $A^{0,c} \cdot_c B = B \subset C$ et $A \cdot_c C \subset C$. Supposons que $A^{n,c} \cdot_c B \subset C$ pour $n \geq 0$. Alors, suivant le corollaire 1.2, $A \cdot_c (A^{n,c} \cdot_c B) \subset A \cdot_c (C)$ et suivant le Corollaire 1.1, $A^{n+1,c} \cdot_c B \subset A \cdot_c C \subset C$. Alors, sachant que $n$, $A^{n,c} \cdot_c B \subset C$, nous avons $Z = A^{*_c} \cdot_c B \subset C$.
3. Finalement, nous montrons que si $c \notin A$, alors n'importe quel langage $Y \in \mathcal{F}$ satisfaisant $Y \subset Z$, implique que $\mathcal{F} = \{Z\}$. Soit $Y \neq Z$ satisfaisant $Y = A \cdot_c Y \cup B$. Supposons que $Y \not\subset Z$. Soit $t \in Y \setminus Z$ tel que $\mathrm{Height}(t)$ est minimal. Évidemment, $B \subset Z$, $t \notin B$. Par conséquent, $t$ appartient à $A \cdot_c Y$ et $t = t_1 \cdot_c t_2$ avec $t_1 \in A$ et $t_2 \in Y$. ($c \notin A$, $t_1 \neq c$). De plus, si $c$ n'apparaît pas dans $t_1$, alors $t = t_1 \in A$ et $t \in A^{*_c} \cdot_c B = Z$ est en contradiction avec $t \notin Z$. Alors, $c$ figure dans $t_1$, et $\mathrm{Height}(t_2) < \mathrm{Height}(t)$. Ce résultat est en contradiction avec la minimalité de la taille de $t$.

   Par conséquent, $Y = Z$.

□

En appliquant des substitutions successives, tout système récursif peut être transformé en un autre système équivalent tel qu'il existe un symbole $\mathbb{E}_j$ qui satisfait $\mathbb{E}_j <_{\mathcal{X}} \mathbb{E}_j$.

Nous présentons un cas spécial où un système récursif peut être résolu :

Premièrement, nous définissons la *décomposition* et la fonction split d'une expression quelconque :

**Définition 5.4.** *La décomposition de $E$ notée $dec(E)$ est définie comme suit :*

$$dec(E) = \begin{cases} \{E\} & \text{si } E = c, E = \mathbb{X}, E = f(E_1, \ldots, E_n), \\ & E = E_1 \cdot_c E_2 \text{ ou bien } E = E_1^{*c} \\ dec(E_1) \cup dec(E_2) & \text{si } E = E_1 + E_2 \end{cases}$$

*Où $E_1, \ldots, E_n$ sont des ERA à travers $(\Sigma, X)$, $\mathbb{X} \in X$ et $c \in \Sigma_0$.*

*Pour une variable $\mathbb{X}$, $dec_{\mathbb{X}}(E) = \{F \in dec(E) \mid \mathbb{X} \text{ figure dans } F\}$. $dec_{\overline{\mathbb{X}}}(E) = dec(E)/dec_{\mathbb{X}}(E)$.*

*Pour un entier $j$, la $j-$split d'une expression $E$ à travers $(\Sigma, \{\mathbb{X}_1, \ldots, \mathbb{X}_n\})$ est le couple $j-\mathrm{split}(E)$ défini par :*

$$j-\mathrm{split}(E) = (\sum_{F \in dec_{\mathbb{X}_j}(E)} F, \sum_{F \in dec_{\overline{\mathbb{X}}_j} E} F)$$

Ensuite, ce couple peut être utilisé pour *factoriser* une équation récursive dans le but d'appliquer la proposition 5.6.

En effet :

**Proposition 5.7.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations. Soit $a \notin \Sigma$. Soit $1 \leq k \leq n$. Soit $\Gamma \subset \Sigma$ tel que $\Gamma = \{c \in \Sigma_0 \mid \{\cdot_c, {}^{*c}\} \cap \mathrm{op}(F_k) \neq \emptyset\}$. Soit $\mathcal{L}$ un $n$-tuple de langages à travers $\Sigma \setminus \Gamma$. Soit $k-\mathrm{split}(F) = (F'_k, F''_k)$. Les déclarations suivantes sont équivalentes :*

1. *$\mathcal{L}$ est une solution pour $\mathbb{E}_k = F_k$,*
2. *$\mathcal{L}$ est une solution pour $\mathbb{E}_k = (F'_k)_{\mathbb{E}_k \leftarrow a} \cdot_a \mathbb{E}_k + F''_k$.*

Du moment où une équation est factorisée, Proposition 5.6 peut être appliquée par *contraction* :

**Définition 5.5.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations. Soit $1 \leq k \leq n$ tel que $\mathbb{E}_k = F'_k \cdot_c \mathbb{E}_k + F''_k$. La* contraction *de $(\mathbb{E}_k = F_k)$ dans $\mathcal{X}$ est le système $\mathcal{X}^k = \{\mathbb{E}_k = (F'_k)^{*_c} \cdot_c F''_k)\} \cup \{\mathbb{E}_j = F_j \mid j \neq k \wedge 1 \leq j \leq n\}$.*

Suivant Proposition 5.6, cette contraction préserve le langage :

**Proposition 5.8.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système d'équations. Soit $1 \leq k \leq n$ tel que $\mathbb{E}_k = F'_k \cdot_c \mathbb{E}_k + F''_k$. Soit $\mathcal{L} = (L_1, \ldots, L_n)$ un $n$-tuple de langages d'arbres. Les déclarations suivantes sont équivalentes :*

1. *$\mathcal{L}$ est une solution pour $\mathcal{X}$,*
2. *$\mathcal{L}$ est une solution pour $\mathcal{X}_k$.*

*De plus, si $c \notin L_{\mathcal{L}}(F'_k)$ alors pour chaque langage $L'_k \neq L_k$,*

$$(L_1, \ldots, L_{k-1}, L'_k, L_{k+1}, \ldots, L_n) \text{ n'est pas une solution pour } \mathcal{X}.$$

**Exemple 5.5.** *Soit $\mathcal{X}_4$ le système illustré dans Exemple 5.3 :*

$$\mathcal{X}^4 = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_3 & = a + h(a + h(\mathbb{E}_3)) \\ \mathbb{E}_4 & = a + h(\mathbb{E}_3) \end{cases}$$

$2 - \text{split}$ *de* $b + f(\mathbb{E}_2, a + h(\mathbb{E}_3))$ *est* $f(x_2, a + h(\mathbb{E}_3)) \cdot_{x_2} \mathbb{E}_2 + b$*, contractée dans* $f(x_2, a + h(\mathbb{E}_3))^{*_{x_2}} \cdot_{x_2} b$*.*

Néanmoins, la factorisation qui précède une contraction peut produire une ERA qui n'est équivalente avec celle de départ. Nous présentons une condition nécessaire mais pas suffisante pour détecter les systèmes pouvant avoir une solution.

La *portée* d'un opérateur $({}^{*_c}, \cdot_c)$ est l'ensemble des opérandes qui figurent à sa gauche. Une occurrence d'un symbole $c \in \Sigma$ est *délimitée* si elle figure dans la portée d'un symbole $c$ ou bien si elle est un symbole de concaténation ($\cdot_c$ ou ${}^{*_c}$).

Une ERA est *fermée* si toutes les occurrences des symboles délimités sont délimitées. Plus généralement, un système est *fermé* si toutes ses équations le sont.

L'ensemble $\text{free}(\mathcal{X})$ contient les symboles de $\Sigma_0$ qui ne sont pas délimités.

Nous montrons maintenant que la *fermeture* est préservée pour la substitution, la factorisation et la contraction.

**Lemme 5.3.** *Soient $F$ et $F'$ deux ERA fermées à travers $\Sigma, \mathbb{E}$ tel que les symboles délimités dans $F$ le sont aussi pour $F'$. Soit $\mathbb{E}_k \in \mathbb{E}$. Alors :*

$$F_{\mathbb{E}_k \leftarrow F'} \text{ est fermée.}$$

*Démonstration.* Inductivement sur la structure de $F$. Soit $H$ une ERA, l'expression $G(H) = H_{\mathbb{E}_k \leftarrow F'}$. Nous mettons $G = G(F)$.

1. Si $F \in \Sigma_0 \cup \{0\} \cup \mathbb{E} \setminus \{\mathbb{E}_k\}$, alors $G = F$. Alors $G$ est fermée.
2. Si $F = \mathbb{E}_k$, alors $G = F'$. Par conséquent, $G$ est fermée.
3. Si $F = f(E_1, \ldots, E_n)$, alors $G = f(G(E_1), \ldots, G(E_n))$. Suivant l'hypothèse d'induction, $G(E_1)$,. . ., et $G(E_n)$ sont fermées, donc $G$ l'est aussi.
4. Si $F = E_1 + E_2$, alors $G = G(E_1) + G(E_2)$. Suivant l'hypothèse d'induction, $G(E_1)$ et $G(E_2)$ sont fermées, même cas pour $G$.
5. Si $F = E_1 \cdot_c E_2$, alors $G = G(E_1) \cdot_c G(E_2)$. Suivant l'hypothèse d'induction, $G(E_1)$ et $G(E_2)$ sont fermées. Sachant que les symboles délimités de $F$ sont aussi délimités dans $F'$, $c$ est délimité dans $G(E_2)$ ($G(E_1)$ est déjà délimité). Par conséquent, $G$ est fermée.
6. Si $F = E_1^{*_c}$, alors $G = (G(E_1))^{*_c}$. Suivant l'hypothèse d'induction, $G(E_1)$ est fermée. Enfin, $G$ est fermée.

□

Par la suite, nous avons :

**Corollaire 5.2.** *Soit $\mathcal{X}$ un système fermé à travers $n$ variables. Soit $1 \leq k \leq n$. Alors :*

$$\mathcal{X}^k \textit{ Est fermé}$$

**Corollaire 5.3.** *Soit $F$ une ERA fermée à travers $\Sigma, \mathbb{E}$. Soit $\mathbb{E}_k \in \mathbb{E}$. Soit $k - \text{split}(F_n) = (F', F'')$. Soit $a \notin \Sigma$. Alors :*

$$(F')_{\mathbb{E}_k \leftarrow a} \cdot_a \mathbb{E}_k + F'' \textit{ est fermée.}$$

Et :

**Lemme 5.4.** *Soit $E = F \cdot_c F' + F''$ une ERA fermée. Alors :*

$$F^{*_c} \cdot_c F'' \textit{ est une ERA fermée.}$$

*Démonstration.* Soit $E' = F^{*_c} \cdot_c F''$. Supposant que $E'$ n'est pas fermée. Soit $c$ est délimité dans $F''$, ou bien il existe un opérateur $\{\cdot_a, ^{*_a}\}$ figurant dans $F$ (resp. $F''$) tel qu'une occurrence de $a$ n'est pas délimitée dans $F''$ (resp. in $F$). Ceci est une contradiction avec la fermeture de $E$.

□

**Corollaire 5.4.** *Soit $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ un système fermé. Soit $1 \leq k \leq n$ tel que $\mathbb{E}_k = F'_k \cdot_c \mathbb{E}_k + F''_k$. Alors :*

$$\mathcal{X}_k \textit{ est fermé.}$$

Dans ce qui suit, nous disons qu'un $n$-tuple d'ERA $(E_1, \ldots, E_n)$ *dénote* une solution rationnelle $(L_1, \ldots, L_n)$ si $L_i = L(E_i)$ pour tout $1 \leq i \leq n$.

**Exemple 5.6.** *Nous utilisons le système $\mathcal{X}$ illustré dans Exemple 5.2. En substituant par $\mathbb{E}_3$, nous obtenons :*

$$\mathcal{X}^3 = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, \mathbb{E}_4) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_4)) \\ \mathbb{E}_3 & = a + h(\mathbb{E}_4) \\ \mathbb{E}_4 & = a + h(a + h(\mathbb{E}_4)) \end{cases}$$

$4-\text{split}$ *de $a + h(a + h(\mathbb{E}_4))$ mène à la factorisation $(h(a + h(x_4))) \cdot_{x_4} \mathbb{E}_{\not\geq} + a$, contractée dans $(h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a$. Alors,*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, \mathbb{E}_4) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_4)) \\ \mathbb{E}_3 & = a + h(\mathbb{E}_4) \\ \mathbb{E}_4 & = (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a \end{cases}$$

*En substituant,*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)) \\ \mathbb{E}_3 & = a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a) \\ \mathbb{E}_4 & = (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a \end{cases}$$

$2-$split *de* $b + f(\mathbb{E}_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a))$ *mène à la factorisation de* $(f(x_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a))) \cdot_{x_2} \mathbb{E}_2 + b$*, contracté dans* $(f(x_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)))^{*_{x_2}} \cdot_{x_2} b$*. Nous aurons le nouveau système*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f((f(x_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)))^{*_{x_2}} \cdot_{x_2} b, \\ & \quad (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a) \\ \mathbb{E}_2 & = (f(x_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)))^{*_{x_2}} \cdot_{x_2} b \\ \mathbb{E}_3 & = a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a) \\ \mathbb{E}_4 & = (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a \end{cases}$$

*Finalement, factoriser/contracter la première équation donne :*

$$\begin{cases} \mathbb{E}_1 & = (f(x_1, x_1))^{*_{x_1}} \cdot_{x_1} (f((f(x_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)))^{*_{x_2}} \cdot_{x_2} b, \\ & \quad (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)) \\ \mathbb{E}_2 & = (f(x_2, a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a)))^{*_{x_2}} \cdot_{x_2} b \\ \mathbb{E}_3 & = a + h((h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a) \\ \mathbb{E}_4 & = (h(a + h(x_4)))^{*_{x_4}} \cdot_{x_4} a \end{cases}$$

N'importe quel système fermé admet une solution *canonique* :

**Théorème 16.** *Soit* $\mathcal{X} = \{(\mathbb{E}_j = F_j) \mid 1 \leq j \leq n\}$ *un système fermé à travers* $\Sigma$ *et* $\{\mathbb{E}_1, \ldots, \mathbb{E}_k\}$*. Alors*

$$\mathcal{X} \textit{ admet une solution rationnelle à travers } \mathrm{free}(\mathcal{X}).$$

*De plus, un* $n$*-tuple D'ERA dénotant cette solution peut être calculé.*

*Démonstration.* Par récurrence sur le cardinal de $\mathcal{X}$.

1. Supposons que l'équation $\mathbb{E}_n = F_n$ n'est pas récursive.
   (a) Si $n = 1$, alors $F_1$ est une ERA et $L(F_1)$ représente la solution unique pour $\mathcal{X}$. Sachant que $\mathcal{X}$ est fermé, $L(F_1) \subset T_{\mathrm{free}(\mathcal{X})}$.
   (b) Sinon, considérons le système $\mathcal{X}' = \mathcal{X}^k \setminus \{\mathbb{E}_n = F_n\}$. A partir du Corollaire 5.2, le système $\mathcal{X}'$ est fermé. En se référant sur l'hypothèse de récurrence, $\mathcal{X}'$ admet une solution rationnelle $Z = (L_1, \ldots, L_{n-1})$ à travers $\mathrm{free}(\mathcal{X})$ dénotée par $(E_1, \ldots, E_{n-1})$. Moyennant le Lemme 5.2, ceci implique que $(L_1, \ldots, L_{n-1}, L_Z(F_n))$ est une solution pour $\mathcal{X}$ et donc l'est aussi pour $\mathrm{free}(\mathcal{X})$. Utilisant le Lemme 5.1, $L_Z(F_n)$ est dénoté par $E_n = (\ldots(F_n)_{\mathbb{E}_1 \leftarrow E_1} \ldots)_{\mathbb{E}_{n-1} \leftarrow E_{n-1}}$, qui est en effet une ERA sans variables. Par

conséquent, $\mathcal{X}$ admet une solution rationnelle $(L_1, \ldots, L_{n-1}, L_Z(F_n))$ à travers $\mathrm{free}(\mathcal{X})$ dénotée par $(E_1, \ldots, E_n)$.

2. Considérons que $\mathbb{E}_n = F_n$ est récursive. Soit $k\mathrm{split}(F_n) = (F', F'')$. Soit $a$ un symbole hors $\Sigma$. Soit $F'_n = (F')_{\mathbb{E}_k \leftarrow a} \cdot_a \mathbb{E}_k + F''$. Sachant que $\mathcal{X}$ est fermé, il est claire en appliquant Proposition 5.7 que $\mathcal{X}$ admet une solution à travers $\mathrm{free}(\mathcal{X})$ si et seulement si $\mathcal{X}' = (\mathcal{X} \setminus \{\mathbb{E}_n = F_n\}) \cup \{\mathbb{E}_n = F'_n\}$ admet lui aussi une solution. En se référant au Corollaire 5.3, $\mathcal{X}'$ est fermé. En utilisant la Proposition 5.8, $\mathcal{X}'$ admet une solution à travers $\mathrm{free}(\mathcal{X})$ si et seulement si $\mathcal{X}'_n$ admet lui aussi une solution. En utilisant le Lemme 5.4, $\mathcal{X}'_n$ est fermé , et contient l'équation $\mathbb{E}_k = F'^{*_c} \cdot_c F''$, qui n'est pas récursive. L'existence de la solution est don prouvée en utilisant le point (1).

□

En d'autres termes,

**Théorème 17.** *Tout système fermé peut être effectivement résolu.*

### 5.1.4 Construction d'une ERA depuis un AAFA

L'idée derrière cette construction est d'extraire un système d'équations depuis un AAFA. Pour chaque état de l'automate, nous pouvons définir une équation rationnelle dénotant son langage accepté.

Premièrement, nous montrons une propriété intéressante pour le langage accepté d'un état :

**Lemme 5.5.** *Soit $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ un AAFA. Soit $q \in Q$. Alors :*

$$L(q) = \bigcup_{(f, q_1, \ldots, q_n, q) \in \Delta} f(L(q_1), \ldots, L(q_n))$$

*Démonstration.* On pose $L'(q) = \bigcup_{(f,q_1,\ldots,q_n,q)\in\Delta} f(L(q_1), \ldots, L(q_n))$. Soit $t = f(t_1, \ldots, t_n)$ un arbre dans $T_\Sigma$. Nous montrons que $t \in L(q) \Leftrightarrow t \in L'(q)$. Par définition, $t \in L(q) \Leftrightarrow q \in \delta(t)$. Alors :

$$\begin{aligned} q \in \delta(t) &\Leftrightarrow \exists (f, q_1, \ldots, q_n, q) \in \Delta, (\forall 1 \leq i \leq n, q_i \in \delta(t_i)) \\ &\Leftrightarrow \exists (f, q_1, \ldots, q_n, q) \in \Delta, (\forall 1 \leq i \leq n, t_i \in L(q_i)) \\ &\Leftrightarrow \exists (f, q_1, \ldots, q_n, q) \in \Delta, t \in f(L(q_1), \ldots, L(q_n)) \\ &\Leftrightarrow t \in L'(q) \end{aligned}$$

□

A travers ce résultat, on peut définir un système d'équations pour un AAFA.

**Définition 5.6.** *Soit $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ un AAFA avec $Q = \{1, \ldots, n\}$. Le* système d'équations *associé à $\mathcal{A}$ est l'ensemble d'équations $\mathcal{X}_\mathcal{A}$ à travers $\mathbb{E}_1, \ldots, \mathbb{E}_n$ défini par $\mathcal{X}_\mathcal{A} = \{\mathcal{E}_q \mid q \in Q\}$ où pour chaque état $q$ in $Q$, $\mathcal{E}_q$ est l'équation $\mathbb{E}_q = F_q$ avec $F_q = \sum_{(f,q_1,\ldots,q_n,q)\in\Delta} f(\mathbb{E}_{q_1}, \ldots, \mathbb{E}_{q_n})$.*

Alors :

**Proposition 5.9.** *Soit $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ un AAFA avec $Q = \{1, \ldots, n\}$. Soit $\mathbb{E} = (E_1, \ldots, E_n)$ une solution pour $\mathcal{X}_\mathcal{A}$. Alors :*

$$\forall 1 \leq j \leq n, L(E_j) = L(j).$$

*Démonstration.* Soit $t \in T_\Sigma$. Nous montrons par induction sur $t$ que $t \in L(E_j) \Leftrightarrow t \in L_j$.

1. Supposons que $t \in \Sigma_0$. Alors

$$\begin{aligned} t \in L(E_j) &\Leftrightarrow t \in L(\sum_{(f,q_1,\ldots,q_n,j)\in\Delta} f(E_{q_1}, \ldots, E_{q_n})) \\ &\Leftrightarrow (t, j) \in \Delta \\ &\Leftrightarrow t \in L(j) \end{aligned}$$

2. Sinon, $t = g(t_1, \ldots, t_k)$ et

$$\begin{aligned} t \in L(E_j) &\Leftrightarrow t \in L(\sum_{(f,q_1,\ldots,q_n,j)\in\Delta} f(E_{q_1}, \ldots, E_{q_n})) \\ &\Leftrightarrow \exists (g, q_1, \ldots, q_k, j) \in \Delta \wedge \forall 1 \leq l \leq k, t_l \in L(E_{q_l}) \\ &\Leftrightarrow \exists (g, q_1, \ldots, q_k, j) \in \Delta \wedge \forall 1 \leq l \leq k, t_l \in L(q_l) \qquad \text{(Hypothèse d'induction)} \\ &\Leftrightarrow t \in \bigcup_{(f,q_1,\ldots,q_n,j)\in\Delta} f(L(q_1), \ldots, L(q_n)) \\ &\Leftrightarrow t \in L(j) \qquad \text{(Lemme 5.5)} \end{aligned}$$

□

Sachant que $\mathcal{X}_A$ est par définition fermé, suivant Théorème 17 :

**Théorème 18.** *Soit $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ un AAFA. Alors :*

$$\mathcal{X}_\mathcal{A} \textit{ peut être effectivement resolu.}$$

Par conséquent :

**Théorème 19.** *Soit $\mathcal{A} = (\Sigma, \{1, \ldots, n\}, Q_f, \Delta)$ un AAFA. Soit $(E_1, \ldots, E_n)$ dénotant une solution de $\mathcal{X}_A$. Alors :*

$$L(\mathcal{A}) \textit{ est dénoté par l'ERA } \sum_{j\in Q_f} E_j.$$

**Exemple 5.7.** *Soit $A$ l'AAFA illustré dans Figure 5.3. Le système associé à $A$ est $\mathcal{X}$ dans Exemple 5.2 :*

$$\mathcal{X} = \begin{cases} \mathbb{E}_1 &= f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, \mathbb{E}_4) \\ \mathbb{E}_2 &= b + f(\mathbb{E}_2, \mathbb{E}_4) \\ \mathbb{E}_3 &= a + h(\mathbb{E}_4) \\ \mathbb{E}_4 &= a + h(\mathbb{E}_3) \end{cases}$$

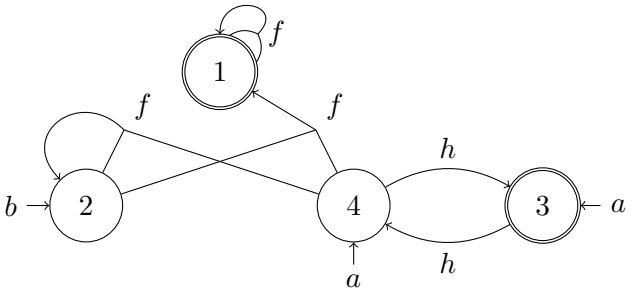



FIGURE 5.3: Exemple d'AAFA.

*Premièrement, nous calculons* $\mathcal{X}^4$ *:*

$$\mathcal{X}^4 = \begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_3 & = a + h(a + h(\mathbb{E}_3)) \\ \mathbb{E}_4 & = a + h(\mathbb{E}_3) \end{cases}$$

*Ensuite, nous résolvons le sous-système suivant :*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_3 & = a + h(a + h(\mathbb{E}_3)) \end{cases} \tag{5.2}$$

$3 - \text{split}$ *de* $a + h(a + h(\mathbb{E}_3))$ *mène à la factorisation de* $h(a + h(x_3)) \cdot_{x_3} \mathbb{E}_3 + a$*, contractée dans* $(h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a$*. Alors :*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h(\mathbb{E}_3)) \\ \mathbb{E}_3 & = (h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a \end{cases}$$

*et en substituant* $\mathbb{E}_3$ *dans*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)) \\ \mathbb{E}_3 & = (h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a \end{cases}$$

*Nous résolvons le nouveau système*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)) \\ \mathbb{E}_2 & = b + f(\mathbb{E}_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)) \end{cases} \tag{5.3}$$

2−split *de* $b + f(\mathbb{E}_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))$ *mène à la factorisation de* $(f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))) \cdot_{x_2} \mathbb{E}_2 + b$, *contractée dans* $((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b$. *et*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(\mathbb{E}_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)) \\ \mathbb{E}_2 & = ((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b \end{cases}$$

*Une autre substitution*

$$\begin{cases} \mathbb{E}_1 & = f(\mathbb{E}_1, \mathbb{E}_1) + f(((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b, \\ & \quad a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)) \\ \mathbb{E}_2 & = ((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b \end{cases}$$

*Factorisation/contraction*

$$\mathbb{E}_1 = (f(x_1, x_1))^{*_{x_1}} \cdot_{x_1} (f(((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b, \\ a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a)))$$

*Enfin, nous obtenons la solution :*

$$\begin{cases} \mathbb{E}_1 & = (f(x_1, x_1))^{*_{x_1}} \cdot_{x_1} (f(((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b, \\ & \quad a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))) \\ \mathbb{E}_2 & = ((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b \\ \mathbb{E}_3 & = (h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a \\ \mathbb{E}_4 & = a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a) \end{cases}$$

*Les états finaux sont* 1 *et* 3. *Le langage* $L(\mathcal{A})$ *est dénoté par l'expression :*

$$(f(x_1, x_1))^{*_{x_1}} \cdot_{x_1} (f(((f(x_2, a + h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))))^{*_{x_2}} \cdot_{x_2} b, a + \\ h((h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a))) + (h(a + h(x_3)))^{*_{x_3}} \cdot_{x_3} a$$

## 5.2 Méthode de résolution de système d'étiquetage

Cette construction utilise une vue graphique sur l'automate d'arbres. Elle consiste à calculer des chemins directs et indirects entres les différents états tout en gardant les symboles rencontrés. Les chemins directs sont des chemins à un saut entre deux états. Un chemin indirect est une succession de chemins directs. En effet, cette technique est une généralisation de l'algorithme de McNaughton-Yamada pour les mots [MY60].

Dans cette partie, L'AAFA $\mathcal{A}(\Sigma, \{1, m\}, Q_f, \Delta)$ où $m$ est un entier naturel strictement positif, est utilisé comme référence. Dans le cas général, nous pouvons proposer un ordre quelconque pour les états. Cet ordre servira pour remplacer chaque état par l'entier naturel indiquant sa position dans la séquence d'états ordonnés.

Nous associons aussi à chaque état $p \in Q$ une variable $x_p$. Ainsi, l'ensemble des variables associées à $Q$ est $X = \{x_1, \ldots, x_m\}$.

Maintenant, nous définissons le *langage de branches* entre deux états.

**Définition 5.7.** *Soient $p, q \in Q$ alors :*

$$\mathcal{P}_{p,q} = \{t(s \leftarrow x_q) \mid s \in L(q), t \in L(p)\}$$

Évidemment, si $t \in \mathcal{P}_{p,q}, t' \in L(p)$ alors $t._{x_p} t' \in L(q)$.

Nous définissons l'ERA $M^{(k)}_{p,q}$ pour états $p, q \in Q$ et un entier $k$ comme suit :

**Définition 5.8.** *Soit $X = \{x_1, \ldots, x_m\}$ l'ensemble de variables associé à $Q$.*

*Alors :*

$$M^{(0)}_{p,q} = \sum_{(f,q_1,\ldots,q,\ldots,q_n,p)\in\Delta} f(x_{q_1}, \ldots, x_q, \ldots, x_{q_n}) \tag{5.4}$$

*Par conséquent, $M^{(k)}_{p,q}$ est calculé par recurence sur un entier naturel $k$ comme suit :*

$$M^{(k)}_{p,q} = M^{(k-1)}_{p,q} + M^{(k-1)}_{p,k}._{x_p}(M^{(k-1)}_{k,k})^{*x_k}._{x_k} M^{(k-1)}_{k,q} \tag{5.5}$$

Il est à noter que $M^{(0)}_{p,q} = 0$ si aucune transition de la forme $(f, q_1, \ldots, q, \ldots, q_n, p)$ n'existe dans $\Delta$

L'idée derrière cette construction est d'effectuer un *développement de branches d'arbres*.

Le cas de base est quand $k = 0$. Il reflète les chemins directs entre l'état en question et l'état cible. En d'autres termes, c'est le calcul d'une matrice $M^0 : Q \to Rat(\Sigma \cup X)$ où les entrée $M^0(i, j)$ sont les transitions contenant $i$ et produisant $j \in \theta$ tout en remplaçant chaque état par la variable correspondante. $M^{(0)}_{p,q} = 0$ dénote le langage vide.
Quant à l'induction, elle calcule les chemins entre $i$ et $\theta$ passant par des états inférieurs à l'état désiré $k$. En d'autres termes, justes les étiquettes dans des chemins ne passant que par des états voulus sont comptées. Si aucun chemin ne passe par $k$ alors le calcul de $M^{(k)}_{p,q}$ revient au calcul de $M^{(k-1)}_{p,q}$. Sinon, le chemin est coupé en trois. La première partie calcule les chemins depuis $p$ jusqu'à $k$ sans cycles (mentionné $M^{(k-1)}_{p,k}$). La deuxième partie calcule les éventuels cycles. La dernière partie dénote le passage de $k$ vers l'état $q$.

Pour simplifier la notation, nous définissons la *concaténation multiple* comme suit :

**Définition 5.9.** *Soient $x_1, \ldots, x_n \in X$. Soient $E, E_1, \ldots, E_n$ $n$ ERA alors :*

$$E._{<x_1,\ldots,x_n>} < E_1, \ldots, E_n >= E._{x_1} E_1 \ldots._{x_n} E_n \tag{5.6}$$

En se basant sur la construction de $M$. Le lemme suivant montre que le calcul de $M^{(|Q|)}_{p,q}$ dénote exactement le langage de branches entre les deux états $p$ et $q$.

**Lemme 5.6.** *Soient $E_1, \ldots, E_n$ $n$ ERA dénotant respectivement $L(q_1), \ldots, L(q_n)$. Nous avons :*

$$\mathcal{P}_{p,q} = [\![M^{(m)}_{p,q}._{<x_1,\ldots,x_{q-1},x_{q+1},\ldots,x_m>} < E_1, \ldots, E_{q-1}, E_{q+1}, \ldots, E_m >]\!]$$

En effet, soit $t \in [\![M_{p,q}^{(k)}]\!]$ pour $k \leq |Q|$. Par définition, $t$ appartient au langage dénoté par $M_{p,q}^{(k-1)}$ ou bien à celui dénoté par $M_{p,k}^{(k-1)} \cdot_{x_p} (M_{k,k}^{(k-1)})^{*x_k} \cdot_{x_k} M_{k,q}^{(k-1)}$. On peut montrer que cette ERA dénote tous les chemins entre $p$ et $q$ passant par tous les états ordonnés. Le résultat contient seulement des symboles de $\cup_{i>0}\Sigma_i$ et des variables de $X$. Après concaténation, seules les occurrences de $x_p$. La condition $|Q|$ assure que tous les états ont bien été visité durant le processus de construction.

Maintenant, nous remplaçons toutes les variables restantes par des éléments de $\Sigma_0$ (qui constituent le point de départ dans la reconnaissance d'un AAFA). Nous définissons l'ensemble $\mathbb{I}$ comme suit :

$$\mathbb{I} = \{p \mid \exists (a, p) \in \Delta, a \in \Sigma_0\}$$

Nous avons :

**Lemme 5.7.** *Soient $p, q \in Q$. Nous avons*

$$L(p) = \bigcup_{q \in \mathbb{I}} (\mathcal{P}_{p,q}) \cdot_{x_q} L(q)$$

*Démonstration.* Soit $t \in \mathcal{P}_{p,q}$ pour deux états $p$ et $q$ dans $Q$. Soit $x_p = \upsilon(p)$. Par définition, nous avons $t \cdot_{x_p} \in L(q)$. Maintenant, soit $t \in L(q), q \in Q$. Donc, $\forall q \in Q, L(q) \neq \emptyset$. Par induction, utilisant la définition de $\delta$, chaque calcul atteint le cas de base qui est les transitions de la forme $(a, p), a \in \Sigma_0, p \in Q$. Par conséquent, il existe $p \in \mathbb{I}$ tel que $t(s \leftarrow x), s \in L(p)$ appartient à $\mathcal{P}_{p,q}$ pour une variable $x$.

□

Afin d'empêcher une transition d'être utilisée plus d'une fois dans un même calcul, nous utilisons $\Delta_m$ pour les transitions qui n'ont pas encore été utilisées.

Pour un état $q$, nous définissons l'ERA $\alpha'(q)$ à travers $(\Sigma, X)$ comme suit (on utilise $\Delta_m$ au lieu de $\Delta$).

$$\alpha'(q) = \sum_{q' \in \mathbb{I}} M_{q,q'}^{(m-1)}$$

Afin d'éviter la génération d'expressions équivalentes à 0 lors d'éventuelle concaténation $\cdot_c$, nous étendons $\alpha'$ pour une état $q$ comme suit :

$$\alpha(q, c) = \begin{cases} \alpha'(q) & \text{si} \quad \alpha'(q) \not\equiv 0 \\ c & \text{sinon} \end{cases}$$

Maintenant, nous définissons les ERA élémentaires $\mathcal{E}_q$ pour $q \in Q$ :

$$\mathcal{E}_q = \sum_{q \in \mathbb{I}} M_{p,q}^{(|Q|)} \cdot_{<x_1,\ldots,x_{q-1},x_{q+1},\ldots,x_m>} < \alpha(1, x_1), \ldots, \alpha(q-1, x_{q-1}), \alpha(q+1, x_{q+1}), \ldots, \alpha(m, x_m) >$$

Par conséquent, nous associons à chaque état un ensemble de symboles de $\Sigma_0$ afin de compléter l'étiquetage. Soit $\mathbb{I} = \{q_1, \ldots, q_n\}$ tel que $q_1, \ldots, q_n \in Q\}$. Pour chaque état $p \in \mathbb{I}$ nous associons l'ERA $\sigma_0(p) = \sum_{(a,p)\in\Delta} a$.
Donc, nous définissons $E_q, q \in Q$ comme suit :

$$E_q = \mathcal{E}_{q\cdot<x_{q_1},\ldots,x_{q_n}>} < \sigma_0(q_1), \ldots, \sigma_0(q_n) > \tag{5.7}$$

Suivant Lemme 5.6 et Lemme 5.7 nous aurons :

**Corollaire 5.5.** *Soit $p \in Q$ alors $[\![E_p]\!] = L(p)$ .*

Enfin :

**Théorème 20.**

$$L(\mathcal{A}) \textit{ est dénoté par } \sum_{p\in Q_f} E_p.$$

En effet, cette construction est simple mais délicate à utiliser à la main. Nous montrons un exemple simple sur un automate à deux états et trois transitions.

**Exemple 5.8.** *Soit l'automate d'arbres suivant.*

$\mathcal{A} = (\{1,2\}, \{a, g(,)\}, \{2\}, \{(a,1), (a,2), (g,1,2,2)\})$.

Nous nous intéressons au seul état final 2. Comme il y a une seule transition avec des entrées et des sorties, qui est $(g,1,2,2)$, nous avons :

$$\begin{aligned}
M_{1,1}^{(0)} &= 0 \\
M_{2,1}^{(0)} &= M_{2,2}^{(0)} = g(x_1, x_2) \\
M_{2,1}^{(1)} &= M_{2,1}^{(0)} + M_{2,1}^{(0)} \cdot_{x_1} (M_{1,1}^{(0)})^{*x_1} \cdot_{x_1} M_{1,1}^{(0)} \\
&= g(x_1, x_2) + g(x_1, x_2) \cdot_{x_1} (0)^{x_1} \cdot_{x_1} 0 \\
&\equiv g(x_1, x_2)
\end{aligned}$$

Nous utilisons les simplifications suivantes :$E^x \cdot_x E \equiv E^x$ et $E + E \equiv E$ pour une ERA $E$ et une variable $x$ (voir le Lemme 1.6). Ceci nous permet de trouver des ERA équivalentes avec une longueur plus petite. De la même manière, nous obtenons :

$$\begin{aligned}
M_{2,1}^{(2)} = M_{2,2}^{(2)} &= M_{2,1}^{(1)} + M_{2,2}^{(1)} \cdot_{x_2} (M_{2,2}^{(1)})^{*x_2} \cdot_{x_2} M_{2,1}^{(1)} \\
&\equiv g(x_1, x_2) + (g(x_1, x_2))^{*x_2} \cdot_{x_2} (g(x_1, x_2))^{*x_2} \\
&\equiv (g(x_1, x_2))^{*x_2}
\end{aligned}$$

Notons que $\Delta^m$ contient seulement les transitions $(a,1)$ et $(a,2)$ qui sont inutiles pour le calcul de $\alpha(1)$ et $\alpha(2)$. Nous aurons $\mathcal{E}_2 = M_{2,1}^{(2)} + M_{2,2}^{(2)}$

Par conséquent, sachant que $\mathbb{I} = \{1,2\}$ et $\sigma_0(1) = \sigma_0(2) = a$, l'ERA $E_2$ est extraite comme suit :

$$\begin{aligned} E_2 &\equiv (g(x_1,x_2))^{*x_2}\cdot_{<x_1,x_2>} < \sigma_0(1), \sigma_0(2) > \\ &\equiv (g(x_1,x_2))^{*x_2}\cdot_{<x_1,x_2>} < a, a > \\ &\equiv ((g(x_1,x_2))^{*x_2}\cdot_{x_1} a)\cdot_{x_2} a \\ &\equiv ((g(a,x_2))^{*x_2})\cdot_{x_2} a \end{aligned}$$

Enfin, nous aurons :

$$[\![((g(a,x_2))^{*x_2})\cdot_{x_2} a]\!] = L(\mathcal{A})$$

## 5.3 Intégration rationnelle

La technique d'intégration est une méthode inverse de la méthode des dérivées partielles. Bien que cette méthode ne définit pas une expression rationnelle équivalente à un automate, mais elle sert bien à prouver quelques propriétés puissantes. En effet, cette technique a été introduite par Smith et Yau [SY72] afin de définir une fonction inverse aux dérivés partielles sur les mots. Dans cette partie, nous généralisons cette notion aux expressions rationnelles d'arbres [GBC15].

L'automate des dérivés pour les mots a été introduit par Brzozowski [Brz64] pour la première fois. Ensuite, Antimirov [Ant96] a amélioré cette construction et a proposé la notion des dérivés partielles. Dans le cas des arbres, Kuske et Meinecke. [KM11] ont généralisé la notion des dérivées partielles pour les ERA d'arbres. Ainsi, la construction d'un automate dit *automate d'équations* est possible.

Nous définissons l'opération de concaténation sur un ensemble ordonné d'ERA :

Soit $\mathcal{N}$ un ensemble de $n$-tuples d'ERA à travers un alphabet $\Sigma$ pour un entier $n \geq 1$. Nous définissons la $n-$*concaténation* avec l'ERA $F$ comme suit :

**Définition 5.10.**

$$\mathcal{N}\cdot_c F = \{(E_1\cdot_c F, \ldots, E_n\cdot_c F) \mid (E_1, \ldots, E_n) \in \mathcal{N}\} \tag{5.8}$$

Maintenant, nous commençons par introduire la notion des *dérivées partielles* comme définie dans [KM11].

**Définition 5.11.** *La dérivée partielle $g^{-1}$ d'une ERA $E$ à travers un alphabet $\Sigma$ pour un symbole $g \in \Sigma_i, i > 0$ est l'ensemble des $n$-tuples défini par induction comme suit :*

$$
\begin{aligned}
g^{-1}(f(E_1,\ldots,E_n)) &= \begin{cases} \{(E_1,\ldots,E_n)\} \textit{ si } f=g \\ \emptyset \textit{ sinon} \end{cases} \\
g^{-1}(E+F) &= g^{-1}(E)\cup g^{-1}(F) \\
g^{-1}(E^{*c}) &= g^{-1}(E).{}_cE^{*c} \\
g^{-1}(E.{}_cF) &= \begin{cases} g^{-1}(E).{}_cF \textit{ si } c\notin[\![E]\!] \\ \\ g^{-1}(E).{}_cF\cup g^{-1}(F) \textit{ sinon} \end{cases}
\end{aligned}
$$

*L'ensemble $\widetilde{\mathcal{N}}$ associé à un $n$-tuple d'ERA est* $\widetilde{\mathcal{N}}=\{F \mid \exists(F_1,\ldots,F,\ldots,F_n)\in\mathcal{N}, n\in\mathbb{N}\}$

*La dérivée de $E$ (notée $\partial(E)$) pour un ensemble de symboles de $(\Sigma/\Sigma_0)^*$ est défini par induction comme suit :*

$$
\begin{aligned}
\partial_\epsilon(E) &= \{E\} \\
\partial_{wg}(E) &= \bigcup_{E'\in\partial_w(E)} \widetilde{g^{-1}E'}, w\in\Sigma^*_{\geq 1}, g\in\Sigma_{\geq 1}
\end{aligned}
$$

*L'automate d'équations* pour une ERA $E$ à travers $\Sigma$ est l'automate défini comme suit :

**Définition 5.12.** *Soit $E$ une ERA définie sur un alphabet $\Sigma$. L'automate d'équations $\mathcal{E}_E=(Q,\Sigma,Q_f,\Delta)$ est défini comme suit :*

$$
\begin{aligned}
Q &= \partial_{\Sigma^*_{\geq 1}}(E) \\
Q_f &= \{E\} \\
\Delta &= \{(f,G_1,\ldots,G_m,F) \mid F\in Q, f\in\Sigma_m, (G_1,\ldots,G_m\in f^{-1}(F))\} \\
&\quad \cup\{(F,c) \mid F\in Q, c\in[\![F]\!]\cap\Sigma_0\}
\end{aligned}
$$

Nous aurons :

**Théorème 21.** *Soit $E$ une ERA. Alors $L(\mathcal{E}_E)=[\![E]\!]$.*

Nous introduisons la notion d'intégration rationnelle et nous montrons que cette dernière peut être vue comme une fonction inverse aux dérivées partielles, nous annonçons la définition suivante.

**Définition 5.13.** *Soit $\mathcal{N}$ un ensemble de $n$-tuple de ERA. L'intégrale de $\mathcal{N}$ par $f\in\Sigma_n$ est*

$$
\int \mathcal{N}\, df = \sum_{(E_1,\ldots,E_n)\in\mathcal{N}} f(E_1,\ldots,E_n) \tag{5.9}
$$

Nous remarquons que $\int \emptyset \; df = 0$ pour tout $f \in \Sigma_>$.

La sémantique de l'intégrale est calculée comme suit :

**Lemme 5.8.**

$$[\![ \int \mathcal{N} \; df ]\!] = \{ f(t_1, \ldots, t_n) \mid (E_1, \ldots, E_n) \in \mathcal{N}, t_1 \in [\![E_1]\!], \ldots, t_n \in [\![E_n]\!], Arity(f) = n \} \tag{5.10}$$

Les résultats suivants (Lemmes 5.9 et 5.10)peuvent être démontrés facilement :

**Lemme 5.9.** *Soient $\mathcal{N}, \mathcal{M}$ deux $n$-tuple d'ERA. Alors :*

$$\int \mathcal{N} \cup \mathcal{M} \; df \equiv \int \mathcal{N} \; df + \int \mathcal{M} \; df$$

**Lemme 5.10.** *Soit $E$ Une ERA. Alors :*

$$\int \mathcal{N} ._c E \; df = (\int \mathcal{N} \; df) ._c E$$

Nous définissons $\lambda_E$ pour une ERA $E$. En effet, la sémantique de cette expression rationnelle regroupe les arbres constantes qui sont dénotés par l'ERA $E$.

**Définition 5.14.** *Soit $n > 0$. Soient $E_1, \ldots, E_n$ $n$ ERA à travers $\Sigma$. Soient $c \in \Sigma_0, f \in \Sigma_n$ alors :*

$$\begin{aligned} \lambda_c &= c \\ \lambda_{f(E_1 \ldots, E_n)} &= 0 \\ \lambda_{E_1 + E_2} &= \lambda_{E_1} + \lambda_{E_2} \\ \lambda_{E_1^{*c}} &= \lambda_{E_1} + c \\ \lambda_{E_1 ._c E_2} &= \lambda_{E_1} ._c \lambda_{E_2} \end{aligned}$$

La proposition suivante montre la relation entre l'intégrale et la dérivée partielle :

**Proposition 5.10.** *Soit $E$ une ERA, alors*

$$E \equiv \sum_{g \in \Sigma_>} \int g^{-1} E \; dg + \lambda_E \tag{5.11}$$

*Démonstration.* La preuve peut être faite par induction sur la structure de $E$.

1. Si $E = a \in \Sigma_0$ alors $\sum_{g \in \Sigma_>} \int g^{-1} E \; dg \equiv \sum 0 + \lambda_E \equiv a = E$.
2. Si $E = f(E_1, \ldots, E_n)$ alors $\lambda_E = 0$. Donc :

$$\begin{aligned} \sum_{g \in \Sigma_>} \int g^{-1} E \; dg &\equiv \int f^{-1} E \; df + \sum_{g \in \Sigma_> / f} \int g^{-1} E \; dg \text{ (Lemme 1.6)} \\ &\equiv \int f^{-1} E \; df + \sum 0 \\ &\equiv \int \{(E_1, \ldots, E_n)\} \; df = f(E_1, \ldots, E_n) = E \end{aligned}$$

3. Si $E = E_1 + E_2$ alors $\lambda_E \equiv \lambda_{E_1} + \lambda_{E_2}$. Donc :

$$\begin{aligned}
\sum_{g\in\Sigma_>} \int (g^{-1}E)\ dg + \lambda_E &= \sum_{g\in\Sigma_>} \int (g^{-1}E_1 \cup g^{-1}E_2)\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} (\int (g^{-1}E_1)\ dg + \int (g^{-1}E_2)\ dg) + \lambda_E \text{ (Lemme 5.9)} \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}E_1)\ dg + \sum_{g\in\Sigma_>} \int (g^{-1}E_2)\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}E_1)\ dg + \lambda_{E_1} + \sum_{g\in\Sigma_>} \int (g^{-1}E_2)\ dg + \lambda_{E_2} \\
&= E_1 + E_2 = E
\end{aligned}$$

4. Si $E = E_1^{*c}$ $(c \in \Sigma_0)$ alors on rencontre deux cas :

   ***cas 1*** Si $c \notin [\![E_1]\!]$ alors $\lambda_E \equiv \lambda_{E_1} + c$. Donc :

$$\begin{aligned}
E &\equiv E_1^{*c} \\
&\equiv E_1 ._c E_1^{*c} + c \text{ (Lemme 1.6)} \\
&\equiv (\sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg + \lambda_{E_1}) ._c E_1^{*c} + c \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg ._c E_1^{*c} + \lambda_{E_1} ._c E_1^{*c} + c \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1) ._c E_1^{*c})\ dg + \lambda_{E_1} + c \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}E)\ dg + \lambda_E
\end{aligned}$$

   ***cas 2*** Si $c \in [\![E_1]\!]$ alors $\lambda_E \equiv \lambda_{E_1}$. Soit $F$ une ERA à travers $\Sigma$ tel que $F + c \equiv E_1$ et $c \notin [\![F]\!]$ donc $\lambda_F \equiv \lambda_{E_1} + c$ et $E^{*c} \equiv (F + c)^{*c} \equiv F^{*c}$. Par conséquent :

$$F^{*c} \equiv \sum_{g\in\Sigma_>} \int (g^{-1}F^{*c})\ dg + \lambda_F + c$$

   Ensuite, nous avons :

$$E_1^{*c} \equiv \sum_{g\in\Sigma_>} \int (g^{-1}E_1^{*c})\ dg + \lambda_E$$

5. Si $E = E_1 ._c E_2$, on distingue deux cas :

***cas 1*** si $c \notin [\![E_1]\!]$ alors $\lambda_E \equiv \lambda_{E_1}$. Nous avons :

$$\begin{aligned}
\sum_{g\in\Sigma_>} \int (g^{-1}(E))\ dg + \lambda_E &= \sum_{g\in\Sigma_>} \int (g^{-1}(E_1.{}_cE_2))\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1).{}_cE_2)\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg.{}_cE_2 + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg.{}_cE_2 + \lambda_{E_1}.{}_cE_2 \text{ (Lemme 1.6)} \\
&\equiv (\sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg + \lambda_{E_1}).{}_cE_2 \\
&\equiv E_1.{}_cE_2 = E
\end{aligned}$$

***cas 2*** si $c \in [\![E_1]\!]$ alors $\lambda_E \equiv \lambda_{E_1}.{}_c\lambda_{E_2} + \lambda_{E_2}$. Nous avons :

$$\begin{aligned}
\sum_{g\in\Sigma_>} \int (g^{-1}(E))\ dg + \lambda_E &= \sum_{g\in\Sigma_>} \int (g^{-1}(E_1.{}_cE_2))\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1).{}_cE_2 \cup g^{-1}(E_2))\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1).{}_cE_2)\ dg + \sum_{g\in\Sigma_>} \int (g^{-1}(E_2))\ dg + \lambda_E \\
&\equiv \sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg.{}_cE_2 + \sum_{g\in\Sigma_>} \int (g^{-1}(E_2))\ dg + \lambda_{E_1}.{}_c\lambda_{E_2} + \lambda_{E_2} \\
&\equiv (\sum_{g\in\Sigma_>} \int (g^{-1}(E_1))\ dg + \lambda_{E_1}).{}_cE_2 + \sum_{g\in\Sigma_>} \int (g^{-1}(E_2))\ dg + \lambda_{E_2} \\
&\equiv E_1.{}_cE_2 + E_2 \equiv E_1.{}_cE_2 = E
\end{aligned}$$

□

Maintenant, nous montrons la relation entre l'intégration et les expressions rationnelles dénotant les langages acceptés dans un AAFA.

Soit $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$ un AAFA. Il faut d'abords s'assurer que l'ERA $\lambda_E$ coïncide avec les constantes reconnues par le langage d'un état donné :

**Lemme 5.11.** *Soit $q \in Q$. Soit $E$ une ERA $\Sigma$ tel que $[\![E]\!] = L(q)$ alors.*

$$\lambda_E \equiv \sum_{(c,q)\in\Delta} c \tag{5.12}$$

Par conséquent :

**Lemme 5.12.**

$$E(q) \equiv \sum_{\substack{(f,q_1,\ldots,q_n,q)\in\Delta \\ n\geq 1}} \int \{(E(q_1),\ldots,E(q_n))\}\ df + \lambda_{E(q)}$$

Enfin, il suffit de sommer toutes les ERA des états finaux afin d'arriver l'ERA voulue :

**Théorème 22.**

$$L(\mathcal{A}) = [\![ \sum_{q\in Q_f} E(q) ]\!]$$

## 5.4 Algorithme de passage d'un AAFA vers une ERA

Nous présentons dans cette partie une heuristique qui permet de calculer une ERA à partir d'un AAFA $\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$.

Ainsi définies, les méthodes de passage d'un AAFA vers une ERA effectuent le calcul par analyse des états de l'AAFA. La question qui se pose est quel est le premier cycle qu'on doit traiter en premier ? En d'autres termes, s'il existe plusieurs cycles alors toutes les techniques ne sont pas déterministes.

Dans cette section, nous définissons notre heuristique qui permet d'optimiser la taille de l'ERA résultante ainsi que le temps de calcul.

L'heuristique repose sur deux astuces :

- Définir un préordre basé sur la notion de largeur d'un état.
- Définir un ordre arbitraire $\overrightarrow{Q}$ sur les états de l'automate en cas d'équivalence d'états vis à vis du préordre. Cet ordre est utilisé pour rendre le calcul déterministe.

Ensuite, nous définissons la notion de *largeur* d'un état. Tout simplement, elle consiste en la taille de toutes les transitions produisant un état $q$ :

**Définition 5.15.** *(Taille d'un état) La taille d'un état $q \in Q$ est $wid(q) = \sum_{(f,q_1,\ldots,q_n,q)\in\Delta} |(f, q_1, \ldots, q_n, q)|$ où $f \in \Sigma_n$ et $q_1, \ldots, q_n \in Q$.*

Ici, la détection de cycles est aussi importante dans le calcul des ERA. Par conséquent, l'heuristique qu'on va utiliser dans notre algorithme est basée sur une relation de préordre sur les états en se basant sur leur tailles, leurs niveaux (définition 4.3) et leurs appartenance au même cycle.

Nous définissons la relation de préordre $\preceq \subseteq Q^2$ comme suit :

**Définition 5.16.** *(Préordre d'états) Soient $p, q$ deux états dans un même cycle, alors $p \preceq q \Leftrightarrow \mathcal{L}(p) + \frac{wid(p)}{|\mathcal{A}|} \leq \mathcal{L}(q) + \frac{wid(q)}{|\mathcal{A}|}$.*

Évidemment, cette heuristique permet d'ordonner les états suivant leurs niveaux. Si une égalité est rencontrée, la taille est le deuxième paramètre à prendre en compte. Il est clair que pour un état $q$, $\frac{wid(p)}{|\mathcal{A}|}$. Alors, cette quantité est décisive si les niveaux de de deux états sont identiques.

L'Algorithme 19 calcule pour chaque état son appartenance à un cycle, son niveau ainsi que sa taille. Pour comprendre l'algorithme, nous définissons :

- $D(q) = \{q', \delta(f, q_1, \ldots, q', \ldots, q_n) = q\}$,
- $P(q)$ identique à l'ensemble des chemins de l'état $q$ (présenté dans Définition 4.2).

D'abord, tous les niveaux d'états sont mis à "nul". La variable $temp$ est un ensemble qui mémorise l'ensemble d'états concernés par l'itération en cours. $temp2$ collecte les états concernés par la prochaine itération. $CurrentLevel$ calcule les niveaux d'états en cours de traitement. La valeur du niveau est mise à 0 et s'incrémente automatiquement à chaque itération. L'algorithme se termine après $|Q|$ itération au maximum (le chemin le plus long).

**Algorithme 19** Calcul des cycles et des niveaux

```
1:  Fonction CyclesLevelComp
2:      CurrentLevel ← 1
3:      temp, temp_2, ← ∅
4:      pour tout q ∈ Q_f faire
5:          L(q) = 1
6:      Fin pour
7:      temp ← Q_f
8:      répéter
9:          pour tout q ∈ temp faire
10:             pour tout q' ∈ D(q) faire
11:                 temp_2 ← temp_2 ∪ {q'}
12:                 P(q) ← P(q) ∪ {q'}
13:                 si q ∈ P(q) alors
14:                     q appartient à un cycle
15:                 Fin si
16:                 si L(q') = 0 alors
17:                     L(q') ← CurrentLevel
18:                 Fin si
19:             Fin pour
20:         Fin pour
21:         temp ← temp2
22:         temp_2 ← ∅
23:         CurrentLevel ← CurrentLevel + 1
24:     jusqu'a CurrentLevel = |Q|
25: Fin Fonction
```

**Lemme 5.13.** *La complexité de l'Algorithme 19 est* $\mathcal{O}(|\mathcal{A}| \log(|Q|))$.

Nous distinguons deux types d'états suivants leurs niveaux :

- $Free$ est l'ensemble des états où les ERA qui dénotent leurs langages acceptés peuvent être directement calculées. Par conséquent, $p \in Free \Leftrightarrow P(q) = \emptyset$.
- $Cycled$ l'ensemble des états participant à des cycles.
- $Linked = Q - (Free \cap Cycled)$ qui ne figurent pas dans des cycles, mais leurs ERA nécessite le calcul des ERA d'autres états participant dans des cycles. Par conséquent, $q \in Linked \Leftrightarrow$ ($p$ ne figure pas dans un cycle et $P(q) \neq \emptyset$).

Pour le calcul des ERA à partir d'AAFA, nous utilisons une idée simple. Elle consiste à transformer les cycles à plusieurs sauts à des cycles en un seul saut pour un état donné. La substitution est la technique qui permet une telle transformation. Afin d'éviter le non déterminisme, le préordre défini au-dessus est utilisé comme heuristique.

L'ERA construite par l'Algorithme 20 dénote exactement le langage accepté par l'AAFA entré comme paramètre. Premièrement, $ComputeRTE$ est la fonction qui retourne $E(q)$ pour un état $q \in Q$ suivant le lemme 5.12 et mémorise le résultat dans $M(q)$. Pour chaque état $q$, nous associons son ensemble de prédécesseurs $Pred(q) = \{q', \exists(f, q_1, \ldots, q, \ldots q_n, q') \in \Delta\}$ où $f \in \Sigma_n$ et $q_1, \ldots q_n \in Q$. $Substitue(E, s)$ substitue l'ERA $E$ dans tous les $M(q), q \in s$. Donc, pour chaque état dans $Free$, $E(q)$ est substituée dans tous les états appartenant à $Pred(q)$.

Concernant les états de $Cycled$. Nous utilisons l'heuristique $\varphi$ qui ordonnent ses états suivant $\preceq$ et $\overrightarrow{Q}$. A chaque cycle directe trouvé (après substitutions), $ComputeCycle$ retourne une ERA non récursive (en se référant à Proposition 5.6 ou bien à Définition 5.8). Si un état $q$ est résolu suivant un cycle direct, les états de $Linked$ ou $q$ figure comme prédécesseur sont traités. Au terme, l'ERA finale est la somme de toutes les ERA dans $M$ correspondante à des états finaux.

**Algorithme 20** Construction d'une ERA à partir d'un AAFA

**Fonction** COMPUTERTEFROMFTA($\mathcal{A} = (\Sigma, Q, Q_f, \Delta)$)
  **pour tout** $q \in Q$ **faire**
    $M(q) \leftarrow ComputeRTE(q)$
  **Fin pour**
  **pour tout** $q \in Free$ **faire**
    $Computed \leftarrow Computed \cup \{q\}$
    $Substitue(M(q), Pred(q))$
  **Fin pour**
  **pour tout** $q \in \varphi(Cycled)$ **faire** ▷
    **si** $M(q)$ contient des cycles directs **alors**
      $M(q) \leftarrow ComputeCycle(q)$
      $Substitue(M(q), Pred(q))$
    **Fin si**
  **Fin pour**
  **répéter**
    **pour tout** $q \in Q - Computed$ **faire**
      **si** Aucun index d'ERA ne figure dans $M(q)$ **alors**
        $Computed \leftarrow Computed \cup \{q\}$
      **sinon**
        **pour tout** $q' \in Pred(q)$ **faire**
          $Substitute(M(q'), \{q\})$
        **Fin pour**
      **Fin si**
    **Fin pour**
  **jusqu'a** $Computed = Q$
**retourner** $\sum_{q \in Q_f} M(q)$
**Fin Fonction**

Notons que malgré l'utilisation d'une heuristique, le temps de calcul est exponentiel au pire cas ($\mathcal{O}(|Q|2^{|Q|})$.

## 5.5 Conclusion

Dans ce chapitre, nous avons solutionné le problème de transformation d'un AAFA en ERA. En effet, Le Théorème de Kleene pour les arbres permit deux constructions, la première construction se matérialise dans le passage d'une ERA vers un AAFA et la deuxième pour l'inverse. Nous avons proposé trois techniques pour la résolution de la deuxième construction. Premièrement, une technique qui consiste à extraire une équation rationnelle depuis chaque état de l'automate. Un système d'équations relatives aux états est alors composé et résolu afin de fournir une ERA équivalente. La seconde construction est de faire identifier et composer les étiquettes de chaque passage depuis un état vers un autre moyennant les transitions de l'automate en question. La dernière construction se base sur une opération inverse à l'automate d'équations. En plus, nous avons proposé une heuristique qui permet de chercher une ERA compacte en terme de longueur alphabétique (pas nécessairement la plus petite) équivalente à un automate donné.

# Chapitre 6

# Boite à outils

Les automates d'arbres et les expressions rationnelles sont de puissants outils théoriques utilisés dans plusieurs domaines. Une panoplie d'algorithmes efficaces, qui leur sont dédiés, ont été proposés dans la littérature. Cependant, très peu de logiciel et d'outils ont été implémentés en vue de leurs manipulations. Dans ce chapitre, nous proposons une boite à outils dédiée aux automates d'arbres et expressions rationnelles ainsi qu'à leurs manipulations. La spécificité de cette boite à outils est qu'elle est basée sur une taxonomie des algorithmes existants prenant en compte un nombre important de critères de classification. Une version préliminaire de cette boite à outils est décrite dans laquelle les algorithmes de minimisation et de conversions sont implémentés.

Les objectifs visés par la conception de la boite à outils sont multiples :

- Avoir une conception simple et cohérente.
- Être compréhensible et extensible.
- Fournir une interface graphique simple et conviviale.
- Avoir un niveau maximal d'efficacité
- Permettre d'effectuer des tests empiriques avec divers mesures de performance.

Concernant les automates d'arbres, cette boite à outils contiendra les algorithmes d'acceptation (reconnaissance d'un arbre), de fermeture (union, composition), de procédures primitives (test du langage vide, équivalence) ainsi que les fonctions fondamentales (déterminisation, minimisation).

Cette boite à outils contiendra aussi, pour le compte des expressions rationnelles, des algorithmes calculant le langage engendré, l'équivalence d'expressions rationnelles ....

Nous commencerons dans ce chapitre par la revue des boites à outils existantes. Ensuite, une description de l'architecture de la boite à outils est décrite en termes de : format de fichier d'interopérabilité choisi, classes d'objets, générateur aléatoires d'arbres et d'automates d'arbres.

## 6.1 Boites à outils existantes

La majorité des techniques et algorithmes liés aux AAFA sont basés sur des généralisations de techniques précédentes sur les mots. Néanmoins, le manque de boite à outils ou de plates-formes logicielles se fait sentir en comparant avec ce qui existe pour les arbres.

Une liste non exhaustive des boites à outils pour les automates d'arbres est présentée dans le Tableau 6.1. [1]

| Boite à outils | Type d'automates | Algorithmes | lien |
|---|---|---|---|
| MONA | Ascendant | Vérification WS1S et WS2S [2] pour les automates | `http://www.brics.dk/mona` |
| TIMBUK | Ascendant, non déterministe | Construction et manipulation d'automates | `http://www.irisa.fr/lande/genet/timbuk` |
| TIBURON | Automates à multiplicité | Construction et manipulation d'automates | `http://www.isi.edu/licensed-sw/tiburon` |
| FOREST FIRE | Ascendant/ Descendant, non déterministe | Reconnaissance, recherche de motifs | `http://www.loekcleophas.com` |
| LIBVATA | Non déterministe | Vérification formelle | `https://github.com/ondrik/libvata` |
| Talib | Multiplicités | Construction et manipulation | `http://www.ims.uni-stuttgart.de/forschung/ressourcen/werkzeuge/talib.en.html` |

TABLE 6.1: Détails sur quelques logiciels sur les automates d'abres

La première plate-forme connue incluant les automates d'arbres est la plate-forme MONA, pour MONAdic second-order logic in practice [EKM98]. Elle est en effet considérée comme un outil très riche pour la communauté de logique mathématique. Cependant, la boite à outil la plus connue maintenant est TIMBUK [GT01].TIMBUK est une collection d'outils afin d'effectuer des preuves de joignabilité (atteinte) pour les systèmes de réécriture et pour manipuler des AAFA. Cet outil est basé sur une plate-forme plus ancienne connu sous le nom de ELAN [BKK+96]. La plate-forme, implémentée avec le langage fonctionnel Ocaml, contient plusieurs composantes :

- *Taml* est une implémentation les opérations de base pour manipuler des AAFA tel que l'acceptation et la déterminisation.
- *Tree Automata Completion Checker* qui est un module permettant de vérifier si automate d'arbres complété vis à vis d'un système de réécriture. Ce module suit le système de gestion des preuves formelles *Coq* [IRI16].
- *TimbukLTA* est un outil dédié à la manipulation des treillis d'arbres [GGLM13].
- *Tabi* un outil de visualisation graphique des arbres et des automates d'arbres.

Une autre boite à outils basée sur TIMBUK est TIBURON [MK06]. Ce logiciel ajoute la dimension de multiplicité aux automates traités par TIMBUK. Récemment, Lengál et al. [LSV12] ont

1. Les liens pour ces outils sont consulté le 03/11/2016
2. Weak second order logic for one/two successors

implémenté la bibliothèque LIBVATA qui est dédiée à la vérification formelle des modèles mathématiques discrets. Plus récemment, FORESTFIRE [CH09], développé par Cleophas et al., est une boite à outil dédiée à l'implémentation des algorithmes de reconnaissance/recherche de motifs.

Suite à cette revue des boites à outils existantes, nous pourrons souligner les faits suivants :

- Mise à part l'outil FORESTFIRE de Cleophas et Hemerik [CH09], peu de ces boites à outils sont basées sur des taxonomies. Ce qui rend leur extension problématique ;
- La plus-part des plateformes sont restées des artefacts de recherche sans existence réelle. En effet, la plus part ne sont pas disponible bien qu'ils sont déclarées Open Source par leurs auteurs comme l'indique le Tableau 6.1
- FORESTFIRE souffre lui d'un manque d'opérations basique tel que la minimisation.

## 6.2 Architecture de notre boite à outils

Dans cette section, nous discutons la conception de notre boite à outil qui se veut extensible et facile à utiliser, tout en implémentant les fonctionnalités absentes dans les autres plates-formes à savoir la minimisation et le passage d'une ERA et un automate.

Cette boite à outils est développée à base de taxonomie. En effet, une taxonomie suit souvent la forme d'arborescence de techniques déjà étudiées lors d'une étape d'*enquête*. Dans cette arborescence, la racine représente la vue la plus généraliste du problème traité. Les différentes techniques sont rencontrées lors d'une exploration plus profonde de l'arborescence. L'usage de l'arbre des taxonomies est bénéfique sur plusieurs plans :

- Rendre la compréhension des algorithmes ainsi que les mutuelles connections pouvant exister plus simple.
- Exploiter le processus de généralisation pour construire des méta-algorithmes suivant l'arbre de taxonomie.
- Permettre de constater des nouvelles tendances inexplorées dans l'arbre de taxonomie et de proposer ainsi de nouvelles solutions.

### 6.2.1 Structure globale

Notre la boite à outil est organisée en *modules indépendants*. Cette vue doit être plutôt schématisée grossièrement avec des modules qui sont implémentés en se basant sur une taxonomie expliquant les différents stages des méthodes tournant autour des intérêts du module implémenté (voir Figure 6.1). Chaque module sera alors organisé hiérarchiquement du plus abstrait vers le plus concret. Cette hiérarchie sera expliquée via des relations d'héritage (l'héritage multiple peut être exploité selon le langage de programmation utilisé).

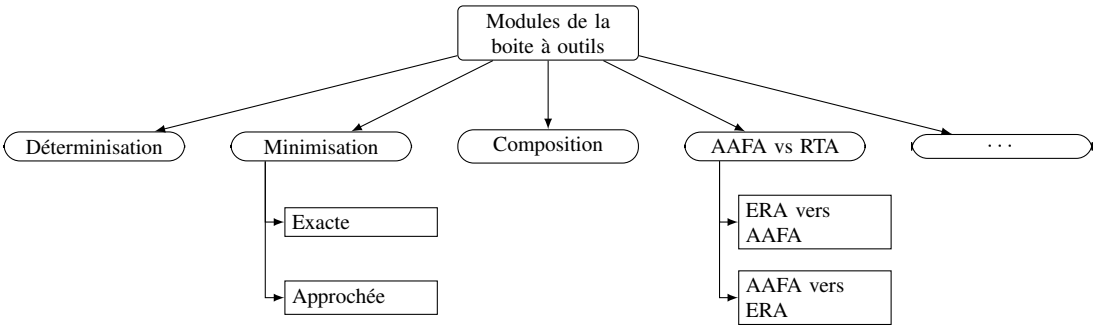



FIGURE 6.1: Extrait des modules de la boite à outils

### 6.2.2 Format d'interopérabilité

Afin d'être extensible, la boite à outils permet l'intégration de code écrit avec différents langages de programmations. En effet, le seul échange permis entre modules se fait via des formats de fichiers standardisés. L'utilisateur peut ainsi développer des nouveaux modules avec le langage de programmation souhaité. Néanmoins, la mise à jour ou l'utilisation d'un module doit subir les contraintes des langages avec lesquels il est implémenté.

Nous définissons un format standardisé pour chaque composante (arbre, automate, expression ...etc). Pour les automates d'arbres, ce format est un document XML expliquant la nature d'un automate ainsi que ses différentes composantes.

Le premier bénéfice de l'utilisation des fichiers XML est l'indépendance entre le contenu d'un objet (arbre, automate ou bien expression) de sa présentation. En outre, cet usage permet une meilleure structuration et une facilité d'accès/modification aux objets sauvegardés ainsi que le codage (Unicode par exemple). Nous pouvons citer aussi la nature ouverte de ce format. En effet, un grand nombre d'outils/bibliothèques libres peuvent en effet être exploités.

Premièrement, la balise `nature` permet de définir la nature de l'objet traité. Si cet objet est un arbre, il peut être rangé ou non (ranked/ unranked), ordonné ou non (ordered, unordererd). S'il s'agit d'un automate ou bien d'une expression rationnelle, de plus des propriétés mentionnées, on peut traiter des objets avec multiplicités (weighted/ unweigted). Dans ce cas, il faut ajouter le symbole $\#$ suivi du poids souhaité.

Principalement, trois types d'objets sont présentés et stockés via leurs représentations XML :

1. Un arbre est défini via la balise `tree`. La balise `structure` présente l'arbre dans une forme naturelle moyennant la balise `node`. Par exemple, l'arbre rangé $f(g(b), g(a))$ est présenté dans la Figure 6.2
2. Pour un automate, nous utilisons la balise `automaton`. Les composants sont définis grâce aux balises `alphabet`, `states`, `FStates` et `rules`. Le document XML de l'automate de référence illustré par l'Exemple 3.2 est présenté dans Figure 6.3.
3. Une ERA est définie par la balise `rte`. Les opérations racine, étoile, concaténation et plus sont illustrées par `root`, `star`, `product` et `plus`. Par exemple, l'ERA $f(a, g(a)^{*a}) + b$ est présenté dans la Figure 6.4.

Notons qu'il existe un format standard pour les automates d'arbres qui est le format TIMBUK. Ce format est aussi pris en charge par la boite à outils.

```
<?xml version="1.0"?>
<tree>
<nature>Ranked,Ordered</nature>
<alphabet>a,0;b,0;g,1;f,2;</alphabet>
<structure>
        <node>f
                <node>g
                        <node>b</node>
                </node>
                <node>g
                        <node>a</node>
                </node>
        </node>
</structure>
</tree>
```

FIGURE 6.2: Document décrivant l'arbre $f(g(b), g(a))$

```
<?xml version="1.0"?>
<automaton>
<nature>Ranked,Unweighted</nature>
<auto>
<alphabet>a,0;b,0;g,1;f,2;</alphabet>
<states>q1,q2,q3,q4,</states>
<Fstates>q3,q4,</Fstates>
<delta>
        <rule>a,q1,</rule>
        <rule>b,q2,</rule>
        <rule>g,q1,q3,</rule>
        <rule>g,q2,q4,</rule>
        <rule>f,q2,q4,q4,</rule>
        <rule>f,q1,q3,q4,</rule>
</delta>
</auto>
</automaton>
```

FIGURE 6.3: Document décrivant un AAFA

```
<?xml version="1.0"?>
<RTE>
<nature>Ranked,Unweighted</nature>
<rte>
<alphabet>a,0;b,0;g,1;f,2;</alphabet>
<structure>
        <plus>
                <root>f
                        <symbol>a</symbol>
                        <star>a
                                <root>g
                                        <symbol>a</symbol>
                                </root>
                        </star>
                </root>
                <symbol>b</symbol>
        </plus>
</structure>
</rte>
</RTE>
```

FIGURE 6.4: Document décrivant l'ERA $f(a, g(a)^{*a}) + b$

| Ensemble | Arbre | Alphabet | ERA | Automate |
|---|---|---|---|---|
| Union | Taille | Arité | Taille alphabétique | Chemin(deux états) |
| Intersection | Sous-arbres | Étiquette | Longueur | Langage accepté (ERA) |
| Différence | Racine | | Dérivée | Langage résiduel (ERA) |
| Cardinal | Feuilles | | Dérivée partielle | Déterminiser |
| Complément | Domaine d'arbre | | Arbre syntaxique | Minimiser |
| Partitionner | Indexer | | First (Folllow et Last) | Libérer du vide |
| Appartient | | | Opérations ($+$, Racine, $._c$, $^{*c}$) | |

TABLE 6.2: Opérations à implémenter

### 6.2.3 Classes d'objets de base

L'application manipule un ensemble d'objets. Nous organisons ces objets dans les classes suivantes. La Figure 6.5 donne une vue générale sur les interactions possibles entre les classes.

1. Ensemble (*Set*) : Une classe "template" qui permet de regrouper des objets de la même nature. Les opérations comme l'union et l'intersection sont implémentées indépendamment de types d'objets manipulés. En effet, la manipulation d'ensembles d'objets tels que les ER, arbres, symboles, est omniprésente dans les différents traitements et méthodes ciblés par la boite à outils.
2. Arbre (*Tree*) : L'élément clé de l'opération de reconnaissance, on peut implémenter des opérations comme la taille, la racine, le nombre des feuilles, etc.
3. Alphabet :Un symbole qui peut être rangé ou non, il est utilisé par les automates d'arbres et les expressions rationnelles
4. Expression rationnelle (*RTE*) : On peut définir des opérations tel que la taille alphabétique, les dérivée partielles, Les fonctions liées aux automates de positions, la transformation ne un automate d'arbres...etc.
5. Automate d'arbres (*FTA*) : Contient des symboles d'un alphabet, des états et des transitions.

Le tableau 6.2 résume la majorité des opérations liées à chaque objet. Chaque classe est organisée hiérarchiquement utilisant une relation d'héritage. Un exemple illustrant la hiérarchie de la classe FTA (acronyme de "Finite Tree Automata" pour un AAFA) est montré par Figure 6.6. Une classe abstraite contenant les opérations générales est basique se situe au sommet de la hiérarchie. Les autres classes (RankedFTA, UnrankedFTA et MultiplicityFTA pour les AAFA rangés, non-rangés et avec multiplicité) héritent directement ou indirectement de la superclasse (Une implémentation *D* est disponible dans l'Annexe 1.). La classe *StringFA* représente un automate de mots, le format de fichiers reconnu est le format fst (OpenFst [RAJ09]). La classe *Graph* représente la version graphique d'un automate d'arbres quelconque. Les deux classes *StringFA* et *Graph* sont nécessaires pour la minimisation proposée dans ce travail.

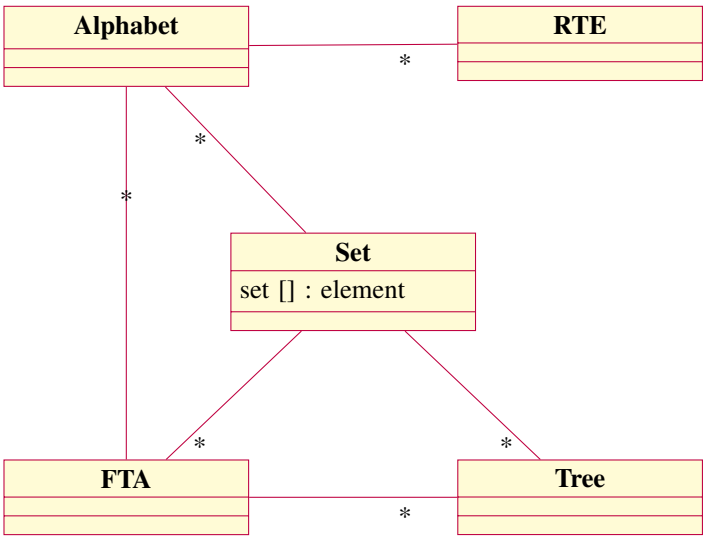



FIGURE 6.5: Classes principales de la boite à outils

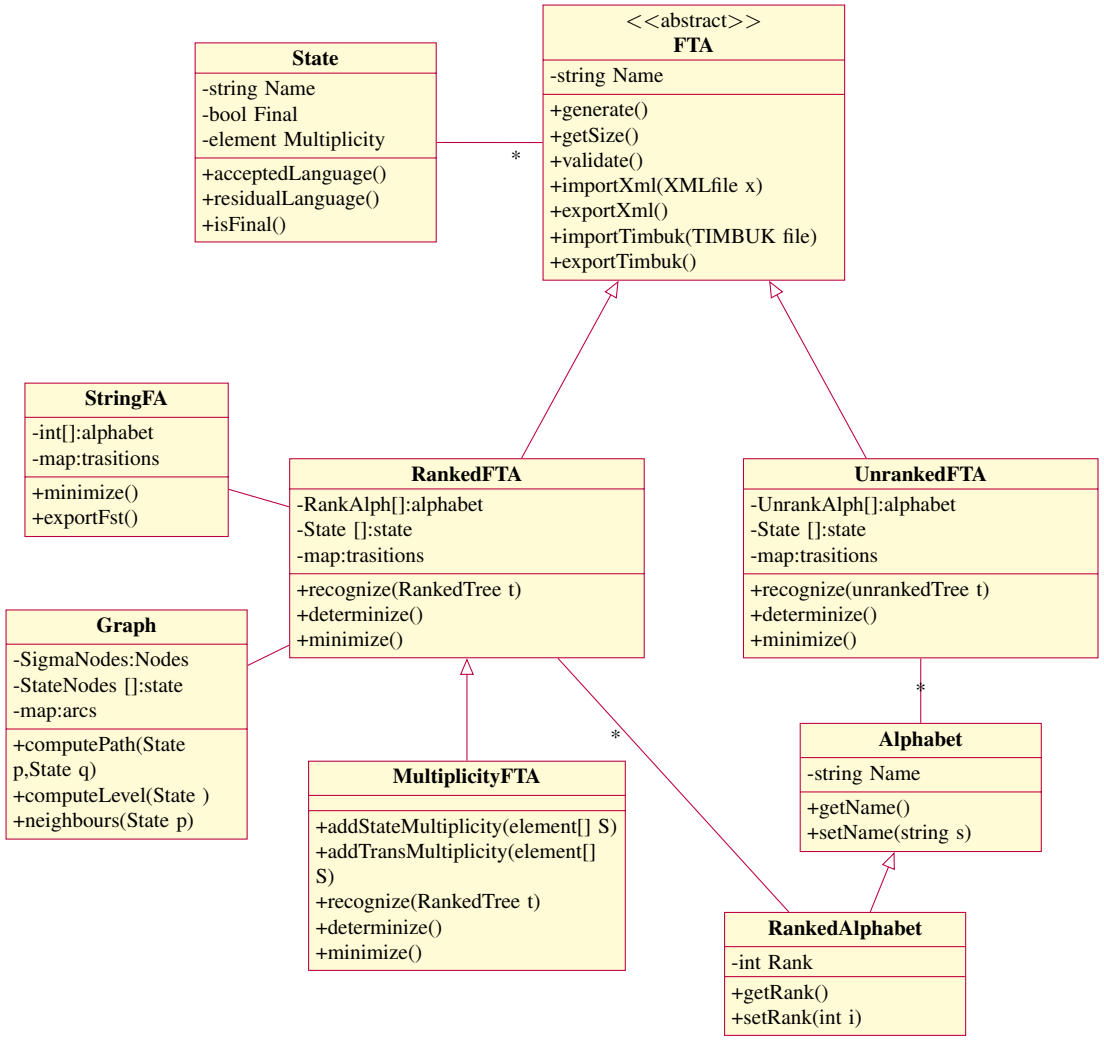



FIGURE 6.6: Hiérarchie de la classe AAFA (FTA)

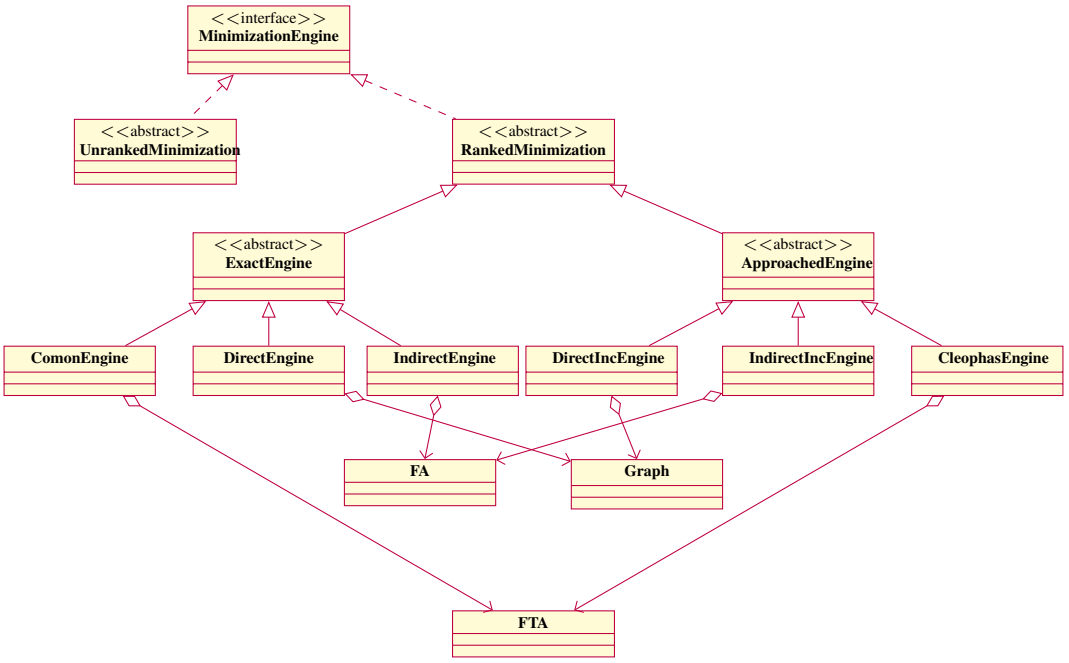



FIGURE 6.7: Hiérarchie des algorithmes de minimisation

### 6.2.4 Classes d'algorithmes

Les algorithmes présents sur les deux taxonomies (Figures 3.1 et 5.1) sont des entités paramétrables à part entière. Une hiérarchie de classes est dédiée à chaque taxonomie.

A titre d'exemple, la hiérarchie des techniques de minimisations est présentée dans la Figure 6.7. En effet, une super classe (interface) appelée*MinimizationEngine* qui est implémentée par la classe des techniques pour automates non rangés (*UnrankedMinimization*) et celle des automates rangés (*RankedMinimization*). Les deux classes *ExactEngine* pour les techniques standard et *ApproachedEngine* pour les techniques approchées héritent à leurs tour de la super-classe *RankedMinimization*. Enfin, un extrait des techniques mentionnées dans l'arbre de taxonomie plus les techniques proposées qui sont *DirectEngine* et *IndirectEngine* pour la méthode de minimisation directe ainsi que *IndirectEngine* et *IndirectIncEngine* pour la méthode indirecte est présenté. Chaque classe est donc reliée par une relation d'agrégation avec la structure appropriée.

### 6.2.5 Génération aléatoire

Une opération très importante pour pouvoir effectuer les tests empiriques est la génération aléatoire, En effet, les classes AAFA (FTA class ) et ERA (RTE class) contiennent toutes les deux une méthode de génération aléatoire :*generate()*. Nous présentons dans cette partie un générateur aléatoire basique pour les AAFA rangés qu'on va appeler *lazy random generator*. Ce générateur permet de créer des échantillons d'automates pour des tests généraux. Un développeur peut facilement modifier ce générateur et/ou proposer un autre. L'Algorithme 21 décrit le générateur aléatoire le plus abstrait d'AAFA. La fonction $Uniform$ génère uniformément un nombre sur un intervalle donnée. Quant à la fonction $Select\ uniformly$, elle choisit aléatoirement un élément depuis l'alphabet et un nombre adéquat d'états. Chaque état à une probabilité de $1/2$ pour qu'il soit final. Le générateur a quarte

paramètres qui sont l'alphabet, le rang maximal (si l'automate voulu est rangé), le nombre voulu de transitions et le nombre d'états. Ce générateur suit la stratégie "Générer puis valider". La phase de validation est invoquée à la demande : si l'échantillon généré est pour la minimisation, aucune contrainte n'est définie. Cependant, si l'échantillon est pour la reconnaissance, la fonction $Validate$ appelle le test du vide etc. Une implémentation en langage $D$ du générateur pour les AAFA rangés non déterministes et déterministes est reportée dans l'Annexe 1.

**Algorithme 21** Générateur basique des AAFA rangés

1: **Fonction** $GENERATE(Alphabet, Maximum\ rank, Transitions\ number, States)$
2: **pour tout** $f \in Alphabet$ **faire**
3: $Rank(f) \leftarrow Uniform([0, Maximum\ Rank])$
4: **Fin pour**
5: **pour tout** $q \in States$ **faire**
6: $P \leftarrow Uniform([0, 1])$
7: **si** $P < 0.5$ **alors**
8: $q$ est final
9: **Fin si**
10: **Fin pour**
11: **répéter**
12: $Select\ Uniformly(f, Alphabet)$
13: $Select\ Uniformly(Rank(f), states)$
14: $Generate\ transition$
15: **jusqu'a** $Transitions\ number$ est atteint
16: **si** $\overline{Validate()}$ **alors**
17: l'échantillon est rejeté ou mis à jours
18: **Fin si**
19: **Fin Fonction**

## 6.3 Détails d'implémentation

La boite à outils est une collection de modules écrits principalement avec le langage $D$ [Ale10]. Néanmoins, quelques fragments de code sont faits en *C++*. Seules les bibliothèques multiplateformes (pour les systèmes d'exploitation) sont utilisées afin d'augmenter la portabilité de la boite à outils. Le code source est disponible en ligne [3].

La boite à outils est organisée comme suit :

- *Noyau (Kernel)* Contient le code source des classes d'objets de base ou bien celle des algorithmes. L'utilisateur peut définir de nouveaux objets de bases ou bien d'algorithmes soit en héritant des classes implémentées ou bien de définir ses propres classes. La boite à outils n'exige pas un langage spécifique pour cette augmentation.
- *Commande utilisateur (Command User)* Cette partie contient des scripts système afin de manipuler des objets instanciées à partir de la partie *Kernel*. Ceci permet de manipuler idéalement les différents objets de la boite à outils. Un utilisateur doit ajouter des fichiers de commandes afin de pouvoir manipuler ses propres fonctionnalités ajoutés dans la partie Kernel. Chaque

3. https://sourceforge.net/projects/taure2-toolkit/

script doit prendre en compte le langage avec lequel la classe manipulée est implémentée. Le code source de chaque commande est sauvegardé dans le dossier *Command User/src*. Les commandes utilisateurs peuvent directement être lancées à partir du terminal correspondant au système d'exploitation (par défaut, chaque commande est doublement implémentée pour être interprétée par le Batch Windows et par le Shell Linux).

- *Interface textuelle (Comand promot)* Un interpréteur de commande utilisateurs permet de faciliter leurs utilisation.
- *Interface graphique (Graphic interface)* est développée en $GTKD$, une extension de la bibliothèque $GTK$ pour le langage $D$. Ceci permet la portabilité de l'interface. De plus, l'interface est faite d'une manière indépendante du reste de la boite à outil. Grâce à cette interface, nous pouvons crée des projets, manipuler des objets ou bien faire des tests d'une manière efficace. L'interface reconnait automatiquement toutes les commandes implantées dans le dossier *Command User*.
- *Tests* Permet à l'utilisateur d'ajouter des fragments de code afin de tester des fonctionnalités ou de donner des tutoriels pour une manipulation donnée.
- *Exemples (Examples)* Contient des fichiers sauvegardés sous le format approprié (Xml, fst et Timbuk). Les résultats d'une génération aléatoire peuvent aussi être sauvegardés sur ce dossier.

## 6.4 Conclusion

Nous avons décrit les briques de base d'une boite à outils orientée application dans ce dernier chapitre. Nous avons montré l'architecture interne qui tourne autour de modules indépendants, ce qui permet de garantir plus de portabilité.

# Conclusion générale

Dans ce mémoire, nous avons étudié l'algorithmique des automates d'arbres en faveur du traitement des grandes masses de données. Un exemple typique de ce domaine est les outils dédiés au traitement des documents XML. Nous avons étudié d'une manière plus abstraite la représentation des grandes masses de données arborescente (ou même une infinité si le motif suivi est "régulier"). Nous nous somme focalisé sur une classe de langages d'arbres dite régulière. Dans ce contexte, nos apports peuvent être résumés dans trois contributions.

La première contribution présentée dans ce mémoire est la minimisation d'automates d'arbres. En effet, minimiser un automate d'arbre permet d'arriver à la structure -la plus compacte- pour reconnaître un ensemble (important ou infini) d'arbres. Nous avons proposé deux améliorations :

- *Méthode directe :*Cette approche repose sur une vue graphique de l'automate à minimiser. Les états et les transitions sont vus comme des nœuds d'un graphe biparti. Des propriétés pour accélérer le processus de minimisation sont définies et extraites.
- *Méthode indirecte :*Dans cette approche, un filtre est proposé afin de transformer l'automate d'arbres en un automate de mots dont la minimisation coïncide avec celui des arbres. Ce passage permet de bénéficier de la panoplie d'algorithmes puissants présents dans la théorie des automates pour les mots. Par ailleurs, ce filtre permet ainsi d'éviter la ré-implémentation de ces algorithmes pour les arbres vu la présence de boites à outils puissante qui implémentent les différentes techniques de minimisation pour les mots.

Nous avons ainsi montré qu'à travers ces deux propositions, nous avons amélioré la majorité des complexités comparées à ce qui existe dans la littérature. Le tableau suivant résume les résultats obtenus

Le tableau 6.3 compare les complexités existantes et les complexités atteintes par les méthodes directe et indirecte. Le paramètres $\hat{r}, m$ et $n$ représentent respectivement le rang maximal, la taille et le nombre d'états pour un automate d'arbres donné (pour un AAFA $\mathcal{A} = (Q, \Sigma, Q_f, \Delta), m = |A|$ et $n = |Q|$).

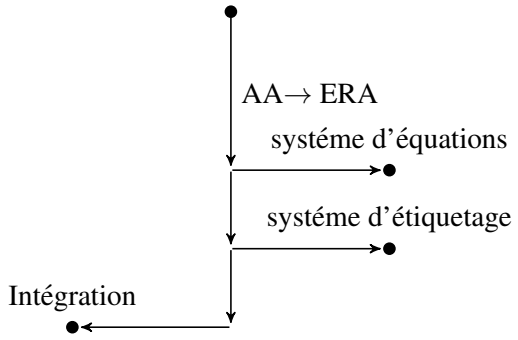



FIGURE 6.8: Expression rationnelle (ERA) ↔ Automate d'arbre (AA)

| **Techniques de minimisation** | **Complexité existante** | **Méthode directe** | **Méthode indirecte** |
|---|---|---|---|
| Standard | $\hat{r} \times m \log(n)$ | $\hat{r} \times m + m \log n$ | $\hat{r} \times m + m \log n$ |
| Acyclique | - | $\hat{r} \times m \log(n)$ | $\hat{r} \times m$ |
| Incrémentale | $m^{n-2}n^2$ | $\hat{r} \times m + n^3 \log(n)$ | $\hat{r} \times m + m \times n^2$ |
| Construction incrémentale | $m \times 2^{\hat{r}} + m$ | - | $m \times 2^{\hat{r}} + m$ |

TABLE 6.3: Comparatif de complexités

La deuxième partie de ce mémoire concerne la relation entre les langages reconnaissables et les expressions rationnelle d'arbres. En effet, cette relation attire plus d'attention dans les dernière années [KM11, MSZ14a, LSZ13, Bel14] vu l'application au filtrage par motifs pour les arbres (tree pattern matching) [AGT89]. En ce qui concerne le passage d'une expression rationnelle vers les automates d'arbres, toutes les techniques présentes pour les mots ont été généralisées aux arbres (automates de Thompson, de positions et de dérivées). Nous avons donc proposé une généralisation des techniques de passage d'un automate d'arbres vers les expressions rationnelle à savoir les techniques de résolution d'un système d'équations rationnelles, de résolution d'un système d'étiquetage et d'intégration (voir Figure 6.8).

Au terme, nous avons proposé un algorithme exponentiel de passage d'un automate vers une expression rationnelle. Cet algorithme peut être considéré comme quadratique si les expressions engendrées sont indexées.

Ce travail peut faire l'objet de plusieurs améliorations comme :

1. La parallélisassions de l'algorithmique proposé ou celle présentes dans la littérature.
2. L'étude de l'algorithme de déterminisation afin de généraliser les résultats aux automates non déterministes. En effet, l'algorithme existant de déterminisation reste polynomial et peut utile dans la pratique.

3. Étudier de plus près l'effet de cette algorithmique sur des semi-anneaux usuels tel que le semi-anneau booléen, le semi-anneau des entiers naturels ...etc.
4. Généraliser les résultats sur d'autres classes de langages d'arbres tel que les langages d'arbres hors contexte [KY15] ,des automates d'arbres à pile [Gue83] et des transducteurs d'arbres [Mal15]...
5. Étudier les algorithmes de minimisation/passage aux expressions rationnelles pour les automates à haie.
6. etc.

# Bibliographie


[ABH+09] Parosh Aziz Abdulla, Ahmed Bouajjani, Lukás Holík, Lisa Kaati, and Tomás Vojnar. Composed bisimulation for tree automata. *Int. J. Found. Comput. Sci.*, 20(4) :685–700, 2009.

[AG68] Michael A. Arbib and Yehoshafat Give'on. Algebra automata I : Parallel programming as a prolegomena to the categorical approach. *Information and Control*, 12(4) :331–345, 1968.

[AG85] Alfred V. Aho and Mahadevan Ganapathi. Efficient tree pattern matching : An aid to code generation. In Mary S. Van Deusen, Zvi Galil, and Brian K. Reid, editors, *POPL*, pages 334–340. ACM Press, 1985.

[AGT89] Alfred V. Aho, Mahadevan Ganapathi, and Steven W. K. Tjiang. Code generation using tree matching and dynamic programming. *ACM Trans. Program. Lang. Syst.*, 11(4) :491–516, 1989.

[Ale10] Andrei Alexandrescu. *The D Programming Language*. Addison Wesley, 2010.

[AMR10] Marco Almeida, Nelma Moreira, and Rogério Reis. Incremental dfa minimisation. In Michael Domaratzki and Kai Salomaa, editors, *CIAA*, volume 6482 of *Lecture Notes in Computer Science*, pages 39–48. Springer, 2010.

[Ant96] Valentin M. Antimirov. Partial derivatives of regular expressions and finite automaton constructions. *Theor. Comput. Sci.*, 155(2) :291–319, 1996.

[Bay71] Rudolf Bayer. Binary b-trees for virtual memory. In *SIGFIDET Workshop*, pages 219–235, 1971.

[BCLQ14] Borja Balle, Xavier Carreras, Franco M. Luque, and Ariadna Quattoni. Spectral learning of weighted automata - A forward-backward perspective. *Machine Learning*, 96(1-2) :33–63, 2014.

[BDR09] Raphaël Bailly, François Denis, and Liva Ralaivola. Grammatical inference as a principal component analysis problem. In *Proceedings of the 26th Annual International Conference on Machine Learning, ICML 2009, Montreal, Quebec, Canada, June 14-18, 2009*, pages 33–40, 2009.

[BDZ15] Johanna Björklund, Frank Drewes, and Niklas Zechner. An efficient best-trees algorithm for weighted tree automata over the tropical semiring. In *Language and Automata Theory and Applications - 9th International Conference, LATA 2015, Nice, France, March 2-6, 2015, Proceedings*, pages 97–108, 2015.

[Bel14] Ahlem Belabaci. Recherche de motifs d'arbres. Master's thesis, Université de Laghouat, 2014.

[BKK+96] Peter Borovanský, Claude Kirchner, Hélène Kirchner, Pierre-Etienne Moreau, and Marian Vittek. ELAN : A logical framework based on computational systems. *Electr. Notes Theor. Comput. Sci.*, 4 :35–50, 1996.

[BKMW01] Anne Brüggemann-Klein, Makoto Murata, and Derick Wood. Regular tree and regular hedge languages over unranked alphabets : Version 1, 2001.

[BLMN15] Mireille Bousquet-Mélou, Markus Lohrey, Sebastian Maneth, and Eric Nöth. XML compression via directed acyclic graphs. *Theory Comput. Syst.*, 57(4) :1322–1371, 2015.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that*. Cambridge University Press, 1998.

[Boz99] Symeon Bozapalidis. Equational elements in additive algebras. *Theory Comput. Syst.*, 32(1) :1–33, 1999.

[Bra67] W. S. Brainerd. *Tree Generating Systems and Tree Automata*. PhD thesis, Purdue University, 1967.

[Brz64] Janusz A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4) :481–494, 1964.

[Bub11] Johannes Bubenzer. Minimization of acyclic dfas. In *Stringology*, pages 132–146, 2011.

[CDF07] Rafael C. Carrasco, Jan Daciuk, and Mikel L. Forcada. An implementation of deterministic tree automata minimization. In Jan Holub and Jan Zdárek, editors, *CIAA*, volume 4783 of *Lecture Notes in Computer Science*, pages 122–129. Springer, 2007.

[CDF09] Rafael C. Carrasco, Jan Daciuk, and Mikel L. Forcada. Incremental construction of minimal tree automata. *Algorithmica*, 55(1) :95–110, 2009.

[CDG+07] H. Comon, M. Dauchet, R. Gilleron, C. Löding, F. Jacquemard, D. Lugiez, S. Tison, and M. Tommasi. Tree automata techniques and applications, 2007. release October, 12th 2007.

[CF02] Rafael C. Carrasco and Mikel L. Forcada. Incremental construction and maintenance of minimal finite-state automata. *Computational Linguistics*, 28(2) :207–216, 2002.

[CH09] Loek G. Cleophas and Kees Hemerik. Forest FIRE : A taxonomy-based toolkit of tree automata and regular tree algorithms. In *Implementation and Application of Automata, 14th International Conference, CIAA 2009, Sydney, Australia, July 14-17, 2009. Proceedings*, pages 245–248, 2009.

[Chi99] Boris Chidlovskii. Using regular tree automata as XML schemas. In *Proc. IEEE Advances on Digital Libraries Conference 2000*, pages 89–98, 1999.

[CHM03] Corinna Cortes, Patrick Haffner, and Mehryar Mohri. Weighted automata kernels - general framework and algorithms. In *8th European Conference on Speech Communication and Technology, Geneva, Switzerland, September 1-4*, 2003.

[CKSW09] Loek G. Cleophas, Derrick G. Kourie, Tinus Strauss, and Bruce W. Watson. On minimizing deterministic tree automata. In *Proceedings of the Prague Stringology Conference 2009, Prague, Czech Republic, August 31 - September 2, 2009*, pages 173–182, 2009.

[Cle08] Loek Cleophas. *Tree Algorithms : Two Taxonomies and a Toolkit.* PhD thesis, Eindhoven University of Technology, april 2008.

[CLT05] Julien Cristau, Christof Löding, and Wolfgang Thomas. Deterministic automata on unranked trees. In *Fundamentals of Computation Theory, 15th International Symposium, FCT 2005, Lübeck, Germany, August 17-20, 2005, Proceedings*, pages 68–79, 2005.

[CNT04] Julien Carme, Joachim Niehren, and Marc Tommasi. Querying unranked trees with stepwise tree automata. In *Rewriting Techniques and Applications, 15th International Conference, RTA 2004, Aachen, Germany, June 3-5, 2004, Proceedings*, pages 105–118, 2004.

[Con61] Michigan University . Engineering Summer Conferences. *Theory of computing machine design : an intensive course for engineers, scientists, and mathematicians. Lectures*. University of Michigan, 1961.

[Cor02] T.H. Cormen. *Introduction à l'algorithmique : cours et exercices*. Sciences SUP. : Informatique. Dunod, 2002.

[CZ01] Jean-Marc Champarnaud and Djelloul Ziadi. From c-continuations to new quadratic algorithms for automaton synthesis. *IJAC*, 11(6) :707–736, 2001.

[DH04] Volker Diekert and Michel Habib, editors. *STACS 2004, 21st Annual Symposium on Theoretical Aspects of Computer Science, Montpellier, France, March 25-27, 2004, Proceedings*, volume 2996 of *Lecture Notes in Computer Science*. Springer, 2004.

[DMWW00] Jan Daciuk, Stoyan Mihov, Bruce W. Watson, and Richard Watson. Incremental construction of minimal acyclic finite state automata. *Computational Linguistics*, 26(1) :3–16, 2000.

[Don68] J.E Doner. Decidability of the weak second-order theory of two successors. *Notices of American Math. Society*, 1968.

[EKM98] Jacob Elgaard, Nils Klarlund, and Anders Møller. MONA 1.x : new techniques for WS1S and WS2S. In *Proc. 10th International Conference on Computer-Aided Verification (CAV)*, volume 1427 of *LNCS*, pages 516–520. Springer-Verlag, June/July 1998.

[Eng75] Joost Engelfriet. Bottom-up and top-down tree transformations— a comparison. *Mathematical systems theory*, 9(2) :198–231, 1975.

[ES77] Joost Engelfriet and Erik Meineche Schmidt. IO and OI. I. *J. Comput. Syst. Sci.*, 15(3) :328–353, 1977.

[Esi10] Z. Esik. Axiomatizing the equational theory of regular tree languages. *The Journal of Logic and Algebraic Programming*, 79(2) :189 – 213, 2010.

[GBC15] Younes Guellouma, Ahlem Belabbaci, and Hadda Cherroun. Towards integrals of rational tree expressions. *Conference on Discrete Mathematics and Computer Science, At Sidi Bel Abbess, Algeria*, 2015.

[GC16] Younes Guellouma and Hadda Cherroun. Efficient implementation for deterministic finite tree automata minimization. *CIT journal of Computing and information technology*, 24(4) :311–322, 2016.

[GCNZ12] Younes Guellouma, Hadda. Cherroun, Attia Nehar, and Djelloul. Ziadi. A fast implementation of incremental minimization of tree automata. In *Innovations in Information Technology (IIT), 2012 International Conference on*, pages 322–327, March 2012.

[GCZW15] Younes Guellouma, Hadda Cherroun, Djelloul Ziadi, and Bruce W. Watson. From tree automata to string automata minimization. *3rd International Workshop on Trends in Tree Automata and Tree Transducers TTATT, At London UK*, 2015.

[GF64] Bernard A. Galler and Michael J. Fisher. An improved equivalence algorithm. *Commun. ACM*, 7(5) :301–303, May 1964.

[GGLM13] Thomas Genet, Tristan Le Gall, Axel Legay, and Valérie Murat. A completion algorithm for lattice tree automata. In *Implementation and Application of Automata - 18th International Conference, CIAA 2013, Halifax, NS, Canada, July 16-19, 2013. Proceedings*, pages 134–145, 2013.

[GS80] Ferenc Gécseg and Magnus Steinby. Minimal ascending tree automata. *Acta Cybern.*, 4 :37–44, 1980.

[GS97] Ferenc Gécseg and Magnus Steinby. Handbook of formal languages, vol. 3. chapter Tree Languages, pages 1–68. Springer-Verlag New York, Inc., New York, NY, USA, 1997.

[GT01] Thomas Genet and Valérie Viet Triem Tong. Reachability analysis of term rewriting systems with timbuk. In *Logic for Programming, Artificial Intelligence, and Reasoning, 8th International Conference, LPAR 2001, Havana, Cuba, December 3-7, 2001, Proceedings*, pages 695–706, 2001.

[Gue83] Irène Guessarian. Pushdown tree automata. *Mathematical Systems Theory*, 16(4) :237–263, 1983.

[GVVL15] Pedro García, Manuel Vázquez de Parga, Jairo A. Velasco, and Damián López. A split-based incremental deterministic automata minimization algorithm. *Theory Comput. Syst.*, 57(2) :319–336, 2015.

[HM09] Markus Holzer and Andreas Maletti. An nlogn algorithm for hyper-minimizing states in a (minimized) deterministic automaton. In *CIAA*, pages 4–13, 2009.

[HMM09] Johanna Högberg, Andreas Maletti, and Jonathan May. Backward and forward bisimulation minimization of tree automata. *Theor. Comput. Sci.*, 410(37) :3539–3552, 2009.

[HMV09] Johanna Högberg, Andreas Maletti, and Heiko Vogler. Bisimulation minimisation of weighted automata on unranked trees. *Fundam. Inform.*, 92(1-2) :103–130, 2009.

[Hop71] John E. Hopcroft. An n log n algorithm for minimizing states in a finite automaton. Technical report, Stanford, CA, USA, 1971.

[HU79] J. E. Hopcroft and J. D. Ullman. *Introduction to Automata Theory, Languages and Computation*. Addison-Wesley, 1979.

[IRI16] Institut de Recherche en Informatique Fondamentale IRIF. The coq proof assistant, reference manual 8.6. Technical report, 2016.

[JM12] Artur Jez and Andreas Maletti. Hyper-minimization for deterministic tree automata. In Nelma Moreira and Rogério Reis, editors, *Proc. 17th Int. Conf. Implementation and Application of Automata*, volume 7381 of *LNCS*, pages 217–228. Springer, 2012.

[Joh75] Donald B. Johnson. Finding all the elementary circuits of a directed graph. *SIAM J. Comput.*, 4(1) :77–84, 1975.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[Kle64] S.C. Kleene. *Introduction to metamathematics*. Bibliotheca mathematica. North-Holland Pub. Co., 1964.

[KM11] Dietrich Kuske and Ingmar Meinecke. Construction of tree automata from regular expressions. *RAIRO - Theor. Inf. and Applic.*, 45(3) :347–370, 2011.

[KMW15] Stefan Kiefer, Ines Marusic, and James Worrell. Minimisation of multiplicity tree automata. In *Foundations of Software Science and Computation Structures - 18th International Conference, FoSSaCS 2015, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2015, London, UK, April 11-18, 2015. Proceedings*, pages 297–311, 2015.

[Kos89] S. Rao Kosaraju. Efficient tree pattern matching (preliminary version). In *FOCS*, pages 178–183. IEEE Computer Society, 1989.

[Koz92] Dexter Kozen. On the Myhill-Nerode theorem for trees. *Bulletin of the EATCS*, 47 :170–173, 1992.

[K.S10] N.Kalyani K.Sunitha. *Formal Languages & Automata Theory*. McGraw-Hill Education (India) Pvt Limited, 2010.

[KY15] Makoto Kanazawa and Ryo Yoshinaka. Distributional learning and context/substructure enumerability in nonlinear tree grammars. In *Formal Grammar - 20th and 21st International Conferences, FG 2015, Barcelona, Spain, August 2015, Revised Selected Papers. FG 2016, Bozen, Italy, August 2016, Proceedings*, pages 94–111, 2015.

[LD60] A. H. Land and A. G. Doig. An automatic method of solving discrete programming problems. *Econometrica*, 28(3) :pp. 497–520, 1960.

[LNG06] Aurélien Lemay, Joachim Niehren, and Rémi Gilleron. Learning n-ary Node Selecting Tree Transducers from Completely Annotated Examples. In *8th International Colloquium on Grammatical Inference*, volume 4201 of *Lecture Notes in Artificial Intelligence*, pages 253–267, Tokyo, Japan, September 2006. Springer Verlag.

[LSV12] Ondrej Lengál, Jirí Simácek, and Tomás Vojnar. VATA : A library for efficient manipulation of non-deterministic tree automata. In *Tools and Algorithms for the Construction and Analysis of Systems - 18th International Conference, TACAS 2012, Held as Part of the European Joint Conferences on Theory and Practice of Software, ETAPS 2012, Tallinn, Estonia, March 24 - April 1, 2012. Proceedings*, pages 79–94, 2012.

[LSZ13] Éric Laugerotte, Nadia Ouali Sebti, and Djelloul Ziadi. From regular tree expression to position tree automaton. In *LATA*, pages 395–406, 2013.

[Mal05] Andreas Maletti. The power of tree series transducers of type I and II. In *Developments in Language Theory, 9th International Conference, DLT 2005, Palermo, Italy, July 4-8, 2005, Proceedings*, pages 338–349, 2005.

[Mal09] Andreas Maletti. Minimizing deterministic weighted tree automata. *Inf. Comput.*, 207(11) :1284–1299, 2009.

[Mal15] Andreas Maletti. Extended tree transducers in natural language processing. In *Proceedings of the 12th International Conference on Finite-State Methods and Natural Language Processing, FSMNLP 2015, Düsseldorf, Germany, June 22-24, 2015*, 2015.

[MK06] Jonathan May and Kevin Knight. Tiburon : A weighted tree automata toolkit. In *Implementation and Application of Automata, 11th International Conference, CIAA 2006, Taipei, Taiwan, August 21-23, 2006, Proceedings*, pages 102–113, 2006.

[Moo56] Edward F. Moore. Gedanken Experiments on Sequential Machines. In *Automata Studies*, pages 129–153. Princeton U., 1956.

[MSZ14a] Ludovic Mignot, Nadia Ouali Sebti, and Djelloul Ziadi. An efficient algorithm for the equation tree automaton via the $k$-c-continuations. *CoRR*, abs/1401.5951, 2014.

[MSZ14b] Ludovic Mignot, Nadia Ouali Sebti, and Djelloul Ziadi. $k$-position, follow, equation and $k$-c-continuation tree automata constructions. In *Proceedings 14th International Conference on Automata and Formal Languages, AFL 2014, Szeged, Hungary, May 27-29, 2014.*, pages 327–341, 2014.

[MY60] Robert McNaughton and Hisao Yamada. Regular expressions and state graphs for automata. *IRE Trans. Electronic Computers*, 9(1) :39–47, 1960.

[PT87] Robert Paige and Robert E. Tarjan. Three partition refinement algorithms. *SIAM J. Comput.*, 16(6) :973–989, December 1987.

[Pun09] A.A. Puntambekar. *Formal Languages And Automata Theory*. Technical Publications, 2009.

[RAJ09] Michael Riley, Cyril Allauzen, and Martin Jansche. Openfst : An open-source, weighted finite-state transducer library and its applications to speech and language. In *Human*

*Language Technologies : Conference of the North American Chapter of the Association of Computational Linguistics, Proceedings, May 31 - June 5, 2009, Boulder, Colorado, USA, Tutorial Abstracts*, pages 9–10, 2009.

[RB04] S. Raeymaekers and M. Bruynooghe. Minimization of finite unranked tree automata. 2004.

[RBC16] Guillaume Rabusseau, Borja Balle, and Shay B. Cohen. Low-rank approximation of weighted tree automata. In *Proceedings of the 19th International Conference on Artificial Intelligence and Statistics, AISTATS 2016, Cadiz, Spain, May 9-11, 2016*, pages 839–847, 2016.

[Rev92] Dominique Revuz. Minimisation of acyclic deterministic automata in linear time. *Theor. Comput. Sci.*, 92(1) :181–189, 1992.

[SY72] L. W. Smith and S. S. Yau. Generation of regular expressions for automata by the integral of regular expressions. *The Computer Journal*, 15(3) :222–228, 1972.

[Tar72] Robert Tarjan. Depth first search and linear graph algorithms. *SIAM Journal on Computing*, 1972.

[Tom06] Marc Tommasi. *Structures arborescentes et apprentissage automatique*. PhD thesis, Université Charles de Gaulle - Lille 3, 2006.

[Ull76] J. R. Ullmann. An algorithm for subgraph isomorphism. *J. ACM*, 23(1) :31–42, January 1976.

[Via01] Victor Vianu. A web odyssey : From codd to xml. In *Proceedings of the Twentieth ACM SIGACT-SIGMOD-SIGART Symposium on Principles of Database Systems*, pages 1–15, Santa Barbara, California, 5 2001. ACM Press.

[Wat93] Bruce W. Watson. A taxonomy of finite automata minimization algorithmes. Computing Science Note 93/44, Eindhoven University of Technology, The Netherlands, 1993.

[Wat01] Bruce W. Watson. An incremental dfa minimization algorithm. In *Finite State Methods in Natural Language Processing*, 2001.

[Wat03] Bruce W. Watson. A new algorithm for the construction of minimal acyclic DFAs. *Sci. Comput. Program.*, 48(2-3) :81–97, 2003.

[WD03] Bruce W. Watson and Jan Daciuk. An efficient incremental DFA minimization algorithm. *Natural Language Engineering*, 9 :49–64, 2003.

[ZL03] Silvano Zilio and Denis Lugiez. Xml schema, tree logic and sheaves automata. In Robert Nieuwenhuis, editor, *Rewriting Techniques and Applications*, volume 2706 of *Lecture Notes in Computer Science*, pages 246–263. Springer Berlin Heidelberg, 2003.

# Annexe

## .1 Code *D* pour la génération aléatoire

```
module lazy_rand;
import FTA;
import essential;
import std.random;
import std.conv;
import std.stdio;
import std.algorithm.searching : canFind,count;
import std.algorithm.comparison : equal;
import std.array:replace;
import std.algorithm.sorting:sort;
import std.algorithm.iteration:group;

FTA lazy_generate(int sigma_Size, int max_Rank, int q_Size,
int delta_Size,string automaton_name)
{
    //--------------generate alphabet---------------------
    FTA targeted=new FTA();;
    Alphabet [] alphabet;
    State [] states;
    State [] f_states;
    string s="f";
    string q="q";
    string [] rules;
    for(int i=0;i<sigma_Size;i++)
    {   auto _rank=uniform(0,max_Rank+1);
        Alphabet A;
        A.name=s~(to!string(i));
        A.rank=_rank;
        alphabet~=A;
    }
    //------------generate states-------------------------
    for(int i=0;i<q_Size;i++){
       State state;
       state.name=q~(to!string(i));
       states~=state;
    }
   for (int i=0;i<delta_Size;i++)
    {   string transition;
        auto pos=uniform(0,alphabet.length);
        transition~=alphabet[pos].name~",";
        for (int j=0;j<alphabet[pos].rank+1;j++)
        {
        auto _state=uniform(0,states.length);
        transition~=states[_state].name~",";
        }
        rules~=transition;
    }
```

```
    for (int i=0;i<q_Size;i++)
    {   int k=uniform(0,2);
        if (k==0){f_states~=states[i];}
    }
    targeted.setName(automaton_name);
    targeted.setAlphabet(alphabet);
    targeted.setStates(states);
    targeted.setRules(rules);
    targeted.setFStates(f_states);
   return targeted;
}

FTA make_generated_Det(FTA A)
{   FTA _dfta=new FTA();
    string [] new_Rules;
    string [] rules=A.getRules();
    string [] without_Output;
    int indexMax=0;
    int maxrank=0;
    foreach(r;rules)
    {   string [] _vect=vectorize_Rule(r);
        _vect=_vect[0.._vect.length-1];
        string _temp;
        for(int l=0;l<_vect.length;l++)
                {
                    _temp~=_vect[l]~",";
                }
        without_Output~=_temp;
    }
    without_Output=without_Output.sort;
    int i=0;
    while(i<without_Output.length)
    {
        string _temp=without_Output[i];
        int c=count(without_Output,without_Output[i]);
        string [] _temp2=vectorize_Rule(without_Output[i]);
        string symb=_temp2[0];
        auto pos=uniform(0,A.getStates().length);
        new_Rules~=_temp~A.getStates()[pos].name~",";
        int index=0;
        for(int j=0;j<c;j++)
        {
            auto pos2=uniform(0,A.getStates().length);
            new_Rules~=_temp.replace(symb,"d"~to!string(index)~to!string
                        (count(_temp,",")-1))~A.getStates()[pos2].name~",";
            if(maxrank<count(_temp,",")-1){maxrank=count(_temp,",")-1;}
            index++;
        }
        if (indexMax<index){indexMax=index;}
        i+=c;
    }
    for(int k=0;k<indexMax;k++)
    {       for(int j=0;j<maxrank;j++)        {
            Alphabet a;
            a.name="d"~to!string(k)~to!string(j);
            a.rank=j;
            A.addAlphabet(a);
        }
    }
    A.setRules(new_Rules);
    return _dfta;

}
```

## .2 Code *D* pour la définition d'alphabets, d'état et d'AAFA

```
module essential;
import std.string;
import std.regex;
import std.stdio;
import std.conv;
import std.algorithm.searching :commonPrefix;
//--------Alphabet definition, a ranked symbol is defined with a name and an integer rank
class Alphabet
{
    string name;

    public :
        string getName(){return this.name;}

        void setName(string n){name=n;}
};
class Ranked_Alphabet:Alphabet
{   int rank;
        int getRank(){return this.rank;}
        void setRank(int r){rank=r;}
}
//---------------signature defintion, it will be used in states definition
struct signature
{
    int occurence;
    int rule_number;
};
//-----------------State definition
class State
{
    string name;
    signature [] sig;
    string [] output;// to mention the output if it is weighted

};
```

## .3 Code *D* pour la définition des AAFA, AAFA rangés et AAFA rangés avec mutiplicités

```
module FTA;
import sa;
import std.stdio,std.string;
import std.xml;
import std.file;
import std.algorithm.searching : canFind;
import std.regex;
import std.conv;
import std.algorithm.comparison:equal;

abstract class FTA
{
        private :
                string name;   //giving a name to the automaton
                State [] states;          //States list
        State [] f_states;       //Final states list
        public :
                validate();
                void setName(string n){name=n;}
```

```
        string getName(){return name;}
                State [] getStates(){return this.states;}
        State [] getFinalStates(){return this.f_states;}
                void visualisate()
       {
           writeln("The name of the automaton is: ",getName());
           writeln("The alphabet is \n",getAlphabet());
           foreach(c;getAlphabet()){writeln("symbol: ",c.name,", rank: ",c.rank);};
           writeln("The states set is \n", getStates());
           writeln("The final states set is \n",getFinalStates());
           writeln("The transitions set is \n",getRules());

       }
}

public class Ranked_FTA:FTA //the Ranked FTA implementation
{
   private :
       Ranked_Alphabet [] alphabet ;    // Defining ranked alphabet
       string [] Q_bar;         //States should appear in the associated DFA.
       string [] rules;         //Transitions set
       DFA associated_Dfa;      //Contains the associated DFA which have to be exported in an appropriate f

   public :
       this()  // a nil FTA constructor
       {
           name="default";
       }
   //------------------------Methods to set and get Fta components ---------------
       string[] getRules(){return rules;}
       Ranked_Alphabet [] getAlphabet(){return this.alphabet;}

       void setAlphabet(Alphabet [] A){this.alphabet=A;}
       void setStates(State [] S){this.states=S;}
       void setRules(string [] R){this.rules=R;}
       void setFStates(State [] S){this.f_states=S;}
       void addAlphabet(Alphabet a){this.alphabet~=a;}
       string [] getQbar(){return this.Q_bar;}

       //---------A function that decides if a state is final or no based on its label (name)

       bool isFinal(State s)
       {
           bool rep=false;
           int i=0;
           string [] _temp;
           foreach(q;f_states)
           {
               _temp~=q.name;
           }
           if (canFind(_temp,s.name)){rep=true;}
           return rep;
       }

      //-------Returns a pointer to the state having a given label
       State* getStateByName(string _name)
       {
            bool n_found=true;
            int i=0;
            State* _state;
            while (n_found && i<this.states.length)
            {
                if (this.states[i].name==_name)
                    {
                       n_found=false;
```

```
                    _state=&this.states[i];
                }else
                {
                    i++;
                }
            }
            return _state;
        }
    //----------A function that decides if two states have exactly the same occurrences in left side transit
        bool are_Equivalent(string[] p,string[] q)
        {
            string[] _temp_p,_temp_q;
            _temp_p=p.dup;
            _temp_p.sort;
            //writeln(_temp_p);
            _temp_q=q.dup;
            _temp_q.sort;
            bool result=true;
            int i=0;
            if (_temp_p.length!=_temp_q.length){result=false;}
            else
            {
               result=equal(_temp_p,_temp_q);
            }
          return result;

        }
        //----------Constructor of an fta from a xml file------------------------------
        this(string c)
        {
            string s = cast(string)std.file.read(c);
            auto xml = new DocumentParser(s);   //create an xml parser on the doc in c
            //int i=0;
            xml.onStartTag["auto"] = (ElementParser xml)
                {
                 xml.onEndTag["alphabet"]= (in Element e) { this.alphabet=extractAlphabet(e.text); };
                 xml.onEndTag["states"]= (in Element e) { this.states=extractStates(e.text); };
                 xml.onEndTag["Fstates"]= (in Element e) { this.f_states=extractStates(e.text); };
                 xml.onEndTag["rule"]= (in Element e) { this.rules~=e.text;};
                 xml.parse();
                };
      xml.parse();

        }
//-------------------------constructor of FTA from Timbuk file------------------
this(string c,int a)
{
    // TODO:not yet achieved
    auto file=File(c,"r");
    string b;
    file.readf("%s",&b);
    string []temp=b.splitLines;
    Alphabet [] temp2=extract_Ops(temp[0]);
    this.alphabet=temp2;
}
//----------------------Validate the constructed automaton-----------------------------
void validate()
{
       bool isLeaf=false;
      foreach(s;this.alphabet)
      {
          if (s.rank==0){isLeaf=true;}
      }
      writeln(isLeaf);
}
```

```
void export_TUK(string file)
{
    //exports the automaton in a timbuk file

    auto F=File(file,"w+");
    string _s="Ops: ";
    foreach(a;this.alphabet)
    {
        _s~=a.name~":"~to!string(a.rank)~" ";
    }
    F.writeln(_s);
    F.writeln("Automaton ",this.getName);
    _s="States ";
    foreach(q;this.states)
    {
        _s~=q.name~",";
    }
    F.writeln(_s);
    _s="Final States ";
    foreach(q;this.f_states)
    {
        _s~=q.name~",";
    }
    F.writeln(_s);
    F.writeln("transitions");

    foreach(t;this.rules)
    {
        _s="";
        int i=0;
        while(t[i]!=',){i++;}
        _s~=t[0..i]~"(";
        int j=t.length-2;
        while(t[j]!=',){j--;}
        int k=t.length;
        if(i!=j){ _s~=t[i+1..j]~") -> "~t[j+1..k-1];}else{_s~=") -> "~t[j+1..k-1];}
        F.writeln(_s);
    }
    F.close();
}
}
public class Multiplicity_FTA:Ranked_FTA //the multiplicity version of Ranked FTA
// (the semiring is that of naturals
{
        private: int []out_state_multiplicties;
                 int []trasitions_output;
        public:
        bool validate()
        {
            if (equal(this.getStates.length,this.out_state_multiplicties.length &&
                this.getRules.length==this.trasitions_output))
                {
                  return true;
                }else{ return false;}
        }
    void addStateMultiplicity(int [] values)
        {
        out_state_multiplicties=values;
        }
        void addTransMultiplicity(element[] values)
        {trasitions_output;=values}
}
```