

# UMONS



Faculté  
des Sciences

## Vérification de la sécurité d'automates à file par apprentissage actif

**Étudiant :** Benjamin André  
**Directrice :** Véronique Bruyère  
18 novembre 2020

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Bases théoriques</b>	<b>6</b>
2.1	Langage . . . . .	6
2.1.1	Définitions . . . . .	6
2.1.2	Opérations sur les langages . . . . .	7
2.1.3	Expressions régulières . . . . .	7
2.2	Automate Fini . . . . .	8
2.2.1	Définitions . . . . .	8
2.2.2	Représentation graphique . . . . .	10
2.2.3	ECLOSE . . . . .	11
2.2.4	Langage . . . . .	11
2.2.5	Équivalence entre expression régulière et automate . . . . .	14
2.2.6	Équivalence entre un automate déterministe fini et un automate non-déterministe fini . . . . .	20
2.3	Table Filling Algorithm . . . . .	23
2.3.1	Relation $R_F$ . . . . .	23
2.3.2	Algorithme . . . . .	25
2.3.3	Appartenance et équivalence . . . . .	27
2.3.4	Minimisation . . . . .	29
2.4	Algorithme d'Angluin . . . . .	31
2.4.1	La relation $R_L$ . . . . .	32
2.4.2	Théorème de Myhill-Nerode . . . . .	33
2.4.3	La table d'observation . . . . .	34
2.4.4	Fermeture et cohérence . . . . .	35
2.4.5	L'algorithme . . . . .	36
<b>3</b>	<b>Notions spécifiques à LeVer</b>	<b>41</b>
3.1	Problème . . . . .	41
3.2	Automate à files . . . . .	42
3.2.1	Définitions . . . . .	42
3.2.2	Produit cartésien . . . . .	43
3.3	Trace d'automate . . . . .	45
3.3.1	Langage tracé . . . . .	45

3.3.2	Alphabet d'annotation . . . . .	46
3.3.3	Trace annotée . . . . .	47
3.3.4	Fonction d'extension de trace . . . . .	47
3.4	Sécurité d'un automate à files . . . . .	49
3.4.1	Définitions . . . . .	49
3.4.2	Traces annotées menant à des états à risques . . . . .	49
<b>4</b>	<b>LeVer</b>	<b>51</b>
4.1	Appartenance . . . . .	52
4.2	Équivalence . . . . .	53
4.3	Vue d'ensemble . . . . .	54
<b>5</b>	<b>Implémentation</b>	<b>56</b>
5.1	Choix . . . . .	56
5.2	Résultats . . . . .	56
<b>6</b>	<b>Conclusion</b>	<b>57</b>
<b>A</b>	<b>Algorithme d'appartenance</b>	<b>59</b>

## Environnements théoriques

Proposition 2.1.1 $\epsilon$ et la concaténation . . . . .	6
Théorème 2.2.1 ADF et expression régulière . . . . .	14
Théorème 2.2.2 ADF $\implies$ expression régulière . . . . .	14
Preuve 2.2.2.1 . . . . .	14
Théorème 2.2.3 Expression régulière $\implies$ ADF . . . . .	17
Preuve 2.2.3.1 . . . . .	17
Théorème 2.2.4 ANF $\Leftrightarrow$ ADF . . . . .	21
Preuve 2.2.4.1 . . . . .	21
Proposition 2.3.1 $R_F$ . . . . .	23
Preuve 2.3.1. $R_F$ est une relation d'équivalence . . . . .	23
Proposition 2.3.2 Congruence de $R_F$ . . . . .	24
Preuve 2.3.2. Congruence de $R_F$ . . . . .	24
Théorème 2.3.3 Table de différenciation . . . . .	25
Preuve 2.3.3.1 . . . . .	26
Théorème 2.3.4 Minimalité de l'automate réduit . . . . .	30
Preuve 2.3.4.1 . . . . .	30
Preuve 2.3.4.2 . . . . .	31
Théorème 2.4.1 . . . . .	32
Lemme 2.4.2 . . . . .	32
Preuve 2.4.2. Equivalence et Congruence à droite . . . . .	32
Lemme 2.4.3 . . . . .	33
Preuve 2.4.3.1 . . . . .	33
Théorème 2.4.4 . . . . .	33
Preuve 2.4.4.1 . . . . .	33
Proposition 2.4.5 . . . . .	34
Preuve 2.4.5.1 . . . . .	35
Théorème 3.3.1 . . . . .	48
Théorème 4.3.1 . . . . .	54

## Exemples

Exemple 1	Langages . . . . .	7
Exemple 2	Expressions régulières . . . . .	8
Exemple 3	Automate déterministe fini . . . . .	9
Exemple 4	$\epsilon$ -ANF . . . . .	9
Exemple 5	Graphe d'automate . . . . .	10
Exemple 6	ECLOSE . . . . .	11
Exemple 7	Chemin . . . . .	12
Exemple 8	. . . . .	12
Exemple 9	. . . . .	13
Exemple 10	Construction d'une expression régulière . . . . .	16

Exemple 11	Transformation ANF vers ADF . . . . .	20
Exemple 12	Table de différenciation . . . . .	25
Exemple 13	. . . . .	29
Exemple 14	. . . . .	36
Exemple 15	. . . . .	42
Exemple 16	. . . . .	44
Exemple 17	. . . . .	46
Exemple 18	. . . . .	47
Exemple 19	. . . . .	52

## Environnements algorithmiques

Algorithme 2.2.1	Appartenance d'un mot à un langage défini par un automate . . . . .	13
Complexité 2.2.1.	Lecture d'un mot par un automate . . . . .	13
Complexité 2.2.2.1	. . . . .	16
Complexité 2.2.3.1	. . . . .	19
Complexité 2.2.4.	Conversion ANF vers ADF . . . . .	22
Complexité 2.2.4.	Conversion ADF vers ANF . . . . .	23
Complexité 2.3.1.1	. . . . .	27
Algorithme 2.3.2	Équivalence entre deux automates . . . . .	28
Complexité 2.3.2.1	. . . . .	29
Algorithme 2.3.3	Minimisation . . . . .	29
Algorithme 2.4.1	Algorithme d'Angluin $L^*$ . . . . .	36
Algorithme 2.4.2	<i>corriger</i> ( $O$ ) . . . . .	37
Algorithme A.0.1	Appartenance à $AL(F)$ . . . . .	59

# Chapitre 1

## Introduction

Huitième version du document.

Le but de ce mémoire est de comprendre et implémenter l'article "Actively learning to verify safety for FIFO automata" [7].

Pour ce faire, plusieurs étapes sont nécessaires. Dans le chapitre 2, les bases théoriques et leurs notations sont rappelées. Celles-ci concernent les automates, les langages et l'algorithme d'Angluin.

Dans le chapitre 3, des notions plus spécifiques telles que les automates à files ou les langages de trace (développés dans l'article) sont abordées.

Ensuite, le chapitre 4 s'appuie sur toutes ces notions et explique l'algorithme LeVer (Learning to Verify) qui permet justement l'apprentissage actif d'automates à files pour vérifier leur sécurité.

Le chapitre 5 mentionne les choix techniques faits pour l'implémentation avant de présenter et discuter les résultats obtenus.

Finalement, le chapitre 6 résume l'apport et les conséquences de ce travail avant de proposer des pistes d'amélioration.

# Chapitre 2

## Bases théoriques

Dans la section 2.1, les notions de langage sont posées. Elles sont ensuite utilisées dans la section 2.2 sur les automates. Le "Table Filling Algorithm" de la section 2.3 se base sur ces automates et permet de les minimiser et de répondre à la requête d'équivalence. Cette requête d'équivalence est une des deux requêtes nécessaire au fonctionnement de l'algorithme d'angluin de la section 2.4.

Tous ces éléments combinés permettent la compréhension des notions plus spécifiques utilisées dans l'article [7] ainsi que celles qui y sont construites. Ces notions se retrouvent dans le chapitre 3.

### 2.1 Langage

Cette section pose les différents concepts et notations pour arriver à la notion de langage. Celles-ci reprennent les notations proposées par Hopcroft et al. [3]. Un sous-ensemble de ces langages correspond à des automates, sujet central de ce document.

#### 2.1.1 Définitions

Un *alphabet*  $\Sigma$  est un ensemble fini et non vide de *symboles*. Un *mot* sur cet alphabet  $\Sigma$  est une suite finie de  $k$  éléments de  $\Sigma$  notée  $w = a_1a_2 \dots a_k$  où  $k$  est un nombre naturel.  $k$  est la *longueur* de ce mot aussi notée  $|w| = k$ . Le *mot vide* est un mot de taille  $k = 0$  noté  $w = \epsilon$ .

La *concaténation* de deux mots  $w = a_1a_2 \dots a_k$  et  $x = b_1b_2 \dots b_j$  est l'opération consistant à créer un nouveau mot  $wx = a_1a_2 \dots a_kb_1b_2 \dots b_j$  de longueur  $i = k + j$ .

**Proposition 2.1.1 ( $\epsilon$  et la concaténation)**  $\epsilon$  est l'identité pour la concaténation, à savoir pour tout mot  $w$ ,  $w\epsilon = \epsilon w = w$ .

Cette proposition est triviale par la définition de la concaténation.

L'*exponentiation* d'un symbole  $a$  à la puissance  $k$ , notée  $a^k$ , retourne un mot de longueur  $k$  obtenu par la concaténation de  $k$  copies du symbole  $a$ . Noter que  $a^0 = \epsilon$ .  $\Sigma^k$  est l'ensemble des mots sur  $\Sigma$  de longueur  $k$ . L'ensemble de tous les mots possibles sur  $\Sigma$  est noté  $\Sigma^* = \bigcup_{k=0}^{\infty} \Sigma^k$ .

Un ensemble quelconque de mots sur  $\Sigma$  est un *langage*, noté  $L \subseteq \Sigma^*$ . Étant donné que  $\Sigma^*$  est infini,  $L$  peut l'être également.

### Exemple 1 (Langages)

Voici des exemples utilisant plusieurs modes de définition.  $\Sigma$  y est implicite mais peut être donné explicitement.

- $L = \{12, 35, 42, 7, 0\}$ , un langage défini explicitement
- $L = \{0^k 1^j \mid k + j = 7\}$ , les mots de 7 symboles sur  $\Sigma = \{0, 1\}$  commençant par zéro, un ou plusieurs 0 et finissant par zéro, un ou plusieurs 1. Ici,  $L$  est donné par notation ensembliste
- $L$  contient tous les noms de villes belges. Ici  $L$  est défini en langage courant.
- $\emptyset$  est un langage sur tout alphabet.
- $L = \{\epsilon\}$  ne contient que le mot vide, et est un langage sur tout alphabet.

## 2.1.2 Opérations sur les langages

Soient  $L$  et  $M$  deux langages. Le langage  $L \cup M = \{w \mid w \in L \vee w \in M\}$  est l'*union* de ces deux langages. Il est composé des mots venant d'un des deux langages.

Le langage composé de tous les mots produits par la concaténation d'un mot de  $L$  avec un mot de  $M$  est une *concaténation* de ces deux langages et s'écrit  $LM = \{vw \mid v \in L \wedge w \in M\}$ .

La *fermeture* de  $L$  est un langage constitué de tous les mots qui peuvent être construits par une concaténation d'un nombre arbitraire de mots de  $L$ , noté  $L^* = \{w_1 w_2 \dots w_n \mid n \in \mathbb{N}, \forall i \in \{1, 2, \dots, n\}, w_i \in L\}$ .

## 2.1.3 Expressions régulières

Certains langages peuvent être exprimés par une *expression régulière*.

Une expression régulière est un mot utilisant les symboles à représenter ainsi que les symboles  $(, ), *, |$  qui sont réservés pour différentes opérations. Une expression régulière est construite à partir d'éléments atomiques (les symboles du langage à représenter) et assemblés pour obtenir des langages plus complexes. Un langage qui peut-être représenté par une expression régulière est dit *langage régulier*.

Si plusieurs expressions régulières peuvent être composées en une expression plus complexe, une expression régulière peut aussi être décomposée en ses différents composants.

**Cas de base** Certains langages peuvent être construits directement sans passer par l'induction :

- $\epsilon$  est une expression régulière. Elle décrit le langage  $L(\epsilon) = \{\epsilon\}$
- $\emptyset$  est une expression régulière décrivant  $L(\emptyset) = \emptyset$
- Si  $a$  est un symbole, alors  $a$  est une expression régulière décrivant le langage  $L(a) = \{a\}$ .

**Induction** Les autres langages réguliers sont construits suivant différentes règles d'induction présentées par ordre décroissant de priorité :

- Si  $E$  est une expression régulière, alors  $(E)$  est une expression régulière et  $L((E)) = L(E)$ .
- Si  $E$  est une expression régulière, alors  $E^*$  est une expression régulière représentant la fermeture de  $L(E)$ , à savoir  $L(E^*) = L(E)^*$ .



- Si  $E$  et  $F$  sont des expressions régulières, alors  $EF$  est une expression régulière décrivant la concaténation des deux langages représentés, à savoir  $L(EF) = L(E)L(F)$ .
- Si  $E$  et  $F$  sont des expressions régulières, alors  $E + F$  est une expression régulière donnant l'union des deux langages représentés, à savoir  $L(E + F) = L(E) \cup L(F)$ . Ici encore, l'opération est associative et la priorité est à gauche.

### Exemple 2 (Expressions régulières)

Soit l'expression  $E = (b + ab)b^*a(a + b)^*$  qui décrit le langage  $L = L(E)$ .

- Le mot  $ba$  fait partie de  $L$ . En effet,  $ba = b\epsilon a\epsilon = (b)b^0a(a + b)^0$ , ce qui respecte bien la définition de  $E$ .
- Le mot  $ababbab$  fait partie de  $L$ . A nouveau,  $ababbab = ab\epsilon a(a + b)(a + b)(a + b)(a + b) = (ab)b^0a(a + b)^4$ .
- Le mot  $aa$  ne fait **pas** partie de  $L$ . Supposons par l'absurde que  $aa \in L$ . Alors il existerait une façon de décomposer  $E$  en  $aa$ . Or, les premiers symboles doivent être soit  $b$ , soit  $ab$ . Il y a contradiction. Donc,  $aa \notin L$ .

Un autre exemple d'expression régulière est  $E = 01^*0$ .  $E$  décrit le langage  $L(E)$  constitué de tous les mots commençant et finissant par 0 avec uniquement des 1 entre les deux.

## 2.2 Automate Fini

Cette section décrit les automates finis, fait le lien avec la notion de langage régulier et propose une représentation visuelle de ces automates. La notation suivie dans cette section s'appuie principalement sur [4], [3] et [5].

### 2.2.1 Définitions

Un *automate fini*  $A = (Q, \Sigma, q_0, \delta, F)$  est défini comme suit :

- $Q$  est un ensemble fini d'états
- $\Sigma$  est un alphabet
- $q_0 \in Q$  est l'état initial
- $\delta$  est la fonction de transition
- $F \subseteq Q$  est un ensemble d'états acceptants.

La fonction de transition  $\delta$  est définie différemment en fonction du type d'automate souhaité :

- **Automate Déterministe Fini (ADF)**  $\delta : Q \times \Sigma \rightarrow Q$ . Soit un état  $q$  et un symbole  $a$ . Alors la transition  $\delta(q, a)$  retourne un état  $p$ .  $\delta(q, a)$  doit être définie pour tout état et tout symbole.
- **Automate Non-déterministe Fini (ANF)**  $\delta : Q \times \Sigma \rightarrow 2^Q$ . Soit un état  $q$  et un symbole  $a$ . Alors la transition  $\delta(q, a)$  retourne un ensemble d'états  $P = \{p_1, p_2, \dots, p_n\} \subseteq Q$ .
- **Automate Non-déterministe Fini avec des transitions sur  $\epsilon$  ( $\epsilon$ -ANF)**  $\delta : Q \times \Sigma \cup \{\epsilon\} \rightarrow 2^Q$ . Pareil que précédemment mais une transition peut exister sans symbole : elle se fait alors sur  $\epsilon$ .

Lorsqu'un automate est mentionné dans ce document, il s'agit implicitement d'un  $\epsilon$ -ANF, sauf mention contraire. En effet, c'est la forme la plus générale. Cependant, ces trois types d'automates ont la même puissance expressive, ce qui est prouvé dans la section 2.2.6.

Soit la transition  $\delta(q, a) = p$  (dans un ADF). Pour  $q$ , c'est une *transition sortante sur  $a$* . Pour  $p$ , c'est une *transition entrante sur  $a$* .

Si  $\delta(q, a) = P = \{p_1, p_2, \dots, p_n\}$  dans un ANF, alors les états  $\{p_1, p_2, \dots, p_n\}$  auront une transition entrante sur  $a$ .

Dans le cas des ANFs et  $\epsilon$ -ANFs, il peut être pratique d'utiliser  $\delta$  sur un ensemble d'états  $S$ . A ce moment,  $\delta(S, a) = \bigcup_{q \in S} \delta(q, a)$  avec  $a \in \Sigma$ .

### Exemple 3 (Automate déterministe fini)

On considère l'automate  $A = (Q, \Sigma, q_0, \delta, F)$  défini comme suit :

- $Q = \{q_0, q_1, q_2, q_3, q_4, q_5, q_6\}$
- $\Sigma = \{a, b\}$
- $q_0$  est l'état du même nom.
- La fonction de transition  $\delta$  est décrite par la table 2.1.
- $F = \{q_3\}$

	a	b
$\rightarrow q_0$	$q_2$	$q_1$
$q_1$	$q_3$	$q_5$
$q_2$	$q_4$	$q_5$
$q_3^*$	$q_3$	$q_3$
$q_4$	$q_4$	$q_4$
$q_5$	$q_3$	$q_1$
$q_6$	$q_4$	$q_5$

TABLE 2.1: La *table de transitions*  $\delta$  d'un ANF

Cette table de transitions est construite comme suit :

- Les en-têtes de colonnes sont des symboles  $a \in \Sigma$ .
- Les en-têtes de lignes sont des états  $q \in Q$ .
- Une cellule à la croisée de la ligne  $q$  et du symbole  $a$  contient un état  $p$  avec  $p = \delta(q, a)$ .

Via la notation de la table 2.1,  $Q$  et  $\Sigma$  sont explicites. En dénotant l'état initial par  $\rightarrow$  et les états acceptants par  $*$  en exposant, on obtient une définition complète d'un automate :  $(Q, \Sigma, q_0, \delta, F)$ .

### Exemple 4 ( $\epsilon$ -ANF)

De la même façon que pour l'exemple précédent, considérons un automate  $A = (Q, \Sigma, q_0, \delta, F)$  défini comme suit :

- $Q = \{q_0, q_1, q_2\}$
- $\Sigma = \{a, b, c\}$
- $q_0$  est l'état du même nom
- $\delta$  est donnée par la table 2.2.

—  $F = \{q_2\}$

$A$  est un  $\epsilon$ -ANF ; une colonne supplémentaire sert à représenter la transition sur  $\epsilon$ .

	$\epsilon$	a	b	c
$\rightarrow q_0$	$\{q_1, q_2\}$	$\emptyset$	$\{q_1\}$	$\{q_2\}$
$q_1$	$\emptyset$	$\{q_0\}$	$\{q_2\}$	$\{q_0, q_1\}$
$q_2^*$	$\emptyset$	$\emptyset$	$\emptyset$	$\emptyset$

TABLE 2.2: La table de transitions  $\delta$  d'un  $\Sigma$ -ANF

Une table similaire sans la colonne  $\epsilon$  représenterait un  $ANF$  au sens strict. Celui-ci ne serait pas pour autant équivalent à l' $\epsilon$ -ANF de la table 2.2.

## 2.2.2 Représentation graphique

Le *graphe d'un automate fini*  $A = (Q, \Sigma, q_0, \delta, F)$  est un graphe dirigé construit comme suit :

- Chaque état de  $Q$  est représenté par un nœud.
- Chaque transition  $\delta(q, a)$  est représenté par un arc étiqueté  $a$ . Dans le cas d'un automate non-déterministe, un arc existe pour chacun des états obtenus en suivant la transition. Si il y a plusieurs transitions sortant d'un même état et entrant dans un même autre état, les arcs peuvent être fusionnés en listant les étiquettes.
- L'état initial est mis en évidence par une flèche entrante.
- Les états acceptants sont représentés par un double cercle, en opposition au simple cercle des autres nœuds.

### Exemple 5 (Graphe d'automate)

Voici les graphes représentant les automates définis dans les tables 2.1 et 2.2 :

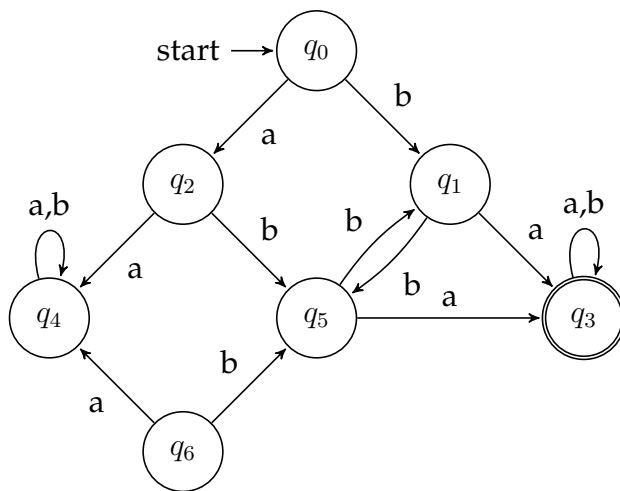


FIGURE 2.1: Graphe de l'ADF de la table 2.1

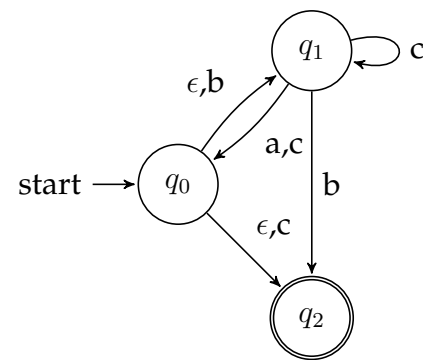


FIGURE 2.2: Graphe du  $\epsilon$ -ANF de la table 2.2

Cette représentation a l'avantage d'être plus visuelle, alors que la table de transition est plus structurée.

### 2.2.3 ECLOSE

Cette sous-section introduit l'algorithme ECLOSE. Cet algorithme concerne les  $\epsilon$ -ANFs. Il permet, à partir d'un état spécifique, de calculer l'ensemble des états atteignables uniquement par des transitions sur  $\epsilon$ . Ce calcul sert notamment au test d'appartenance d'un mot à un langage défini par un  $\epsilon$ -ANF comme présenté à la section 2.2.4.

Soit un  $\epsilon$ -ANF  $A = (Q, \Sigma, q_0, \delta, F)$ . Il est possible de construire une fonction retournant l'ensemble des états atteints uniquement en suivant des transitions sur  $\epsilon$  pour un état  $q$  donné. Cette fonction est la *fermeture sur epsilon*  $ECLOSE : Q \rightarrow 2^Q$ . Sa définition est inductive.

Soit  $q$  un état dans  $Q$ .

**Cas de base**  $q$  est dans  $ECLOSE(q)$

**Pas de récurrence** Si  $p$  est dans  $ECLOSE(q)$  et qu'il existe un état  $r$  tel quel  $r \in \delta(p, \epsilon)$ , alors  $r$  est dans  $ECLOSE(q)$ .

ECLOSE peut être utilisé indifféremment sur un ensemble d'états  $S$  ( $ECLOSE : 2^Q \rightarrow 2^Q$ ). Alors,  $ECLOSE(S) = \bigcup_{q \in S} ECLOSE(q)$ .

#### Exemple 6 (ECLOSE)

Considérons l'automate  $A$  de l'exemple 4. Les différentes fermetures peuvent être calculées :

- $ECLOSE(q_0) = \{q_0, q_1, q_2\}$ . En effet,  $q_0$  appartient à sa fermeture, selon le cas de base. Aussi,  $q_1, q_2 \in \delta(q_0, \epsilon)$ .
- $ECLOSE(q_1) = \{q_1\}$  par le cas de base.
- $ECLOSE(q_2) = \{q_2\}$  par le cas de base.

### 2.2.4 Langage

Un automate représente un langage. Cette section explique comment il le fait pour les 3 catégories d'automates.

#### Fonction de transition étendue

La *fonction de transition étendue*  $\hat{\delta}$  est une extension de la fonction de transition, acceptant plusieurs symboles de façon consécutive. Intuitivement, il s'agit de suivre plusieurs arcs sur le graphe.

Comme  $\delta$  est différente en fonction du type d'automate considéré,  $\hat{\delta}$  l'est aussi.

- **ADF** :  $\hat{\delta} : Q \times \Sigma^* \rightarrow Q$ .  $\hat{\delta}$  prend en entrée un état de  $Q$  et un mot  $w$  sur  $\Sigma$  et retourne un état de  $Q$ .
- **ANF et  $\epsilon$ -ANF** :  $\hat{\delta} : Q \times \Sigma^* \rightarrow 2^Q$ .  $\hat{\delta}$  prend en entrée un état de  $Q$  et un mot  $w$  sur  $\Sigma$  et retourne un ensemble d'états de  $Q$ .

Soit un état  $q \in Q$  et un mot  $w \in \Sigma^*$ . Alors  $\hat{\delta}$  est définie par :

**Cas de base**  $w$  est un mot vide :

- Pour un ADF ou ANF :  $\hat{\delta}(q, \epsilon) = q$ .
- Pour un  $\epsilon$ -ANF :  $\hat{\delta}(q, \epsilon) = ECLOSE(q)$ .

**Pas de récurrence** Si  $|w| \geq 1$ , alors  $w = xa$  avec  $x$  un mot sur  $\Sigma$  et  $a$  un symbole de  $\Sigma$ .

- Pour un ADF ou ANF :  $\hat{\delta}(q, w) = \hat{\delta}(q, xa) = \delta(\hat{\delta}(q, x), a)$
- Pour un  $\epsilon$ -ANF :  $\hat{\delta}(q, w) = \hat{\delta}(q, xa) = ECLOSE(\delta(\hat{\delta}(q, x), a))$ .

Dans le cas particulier où  $|w| = 1$ ,  $w = xa = \epsilon a$ , ce qui mène au cas de base.

Il se peut que la fonction de transition  $\delta$  ne soit pas définie pour une paire d'arguments. Auquel cas,  $\hat{\delta}$  ne l'est pas non plus.

## Chemin

Un *chemin* est une suite d'états d'un automate. Chaque état de cette suite doit être atteignable par une transition depuis l'état précédent dans cette suite.

### Exemple 7 (Chemin)

Considérons l'automate  $A$  de la figure 2.1. Il existe un chemin de  $q_0$  à  $q_5$  :  $(q_0, q_2, q_5)$ . En effet, les transitions sont définies :  $\delta(q_0, a) = q_2$  et  $\delta(q_2, b) = q_5$ . Un mot représenté par ce chemin est  $ab$ .

Dans le cas où plusieurs transitions mènent d'un état à un autre, plusieurs mots peuvent être représentés par un même chemin. Chaque état d'un chemin entre deux états  $p$  et  $q$  et distinct de ceux-ci est dit *état intermédiaire*. S'il existe un chemin menant de  $q_0$  à un état  $p$ ,  $p$  est dit *atteignable*. Dans le cas contraire, il est *inatteignable*.

## Langage

Le langage représenté par un automate  $A = (Q, \Sigma, q_0, \delta, F)$   $L(A)$  peut alors se définir comme les mots qui, par l'application de  $\hat{\delta}$  sur l'état initial, donnent un état acceptant. Si deux automates représentent le même langage, ils sont dit *équivalents*. Voici les définitions ensemblistes, respectivement pour un ADF et pour un  $(\epsilon)$ -ANF

$$L(A) = \{w \in \Sigma^* \mid \hat{\delta}(q_0, w) \in F\} \quad L(A) = \{w \in \Sigma^* \mid \exists q \in \hat{\delta}(q_0, w), q \in F\}$$

Un mot appartenant à  $L(A)$  est dit *accepté* par l'automate  $A$ .

### Exemple 8

Soit  $A$  l'ADF de la figure 2.1. Alors  $abaa$  est un exemple de mot appartenant à  $L(A)$ . En effet :

$$\begin{aligned} \hat{\delta}(q_0, abaa) &= \\ \delta(\hat{\delta}(q_0, aba), a) &= \\ \delta(\delta(\hat{\delta}(q_0, ab), a), a) &= \\ \delta(\delta(\delta(\hat{\delta}(q_0, a), b), a), a) &= \\ \delta(\delta(\delta(\delta(q_0, a), b), a), a) &= \\ \delta(\delta(\delta(q_2, b), a), a) &= \\ \delta(\delta(q_5, a), a) &= \\ \delta(q_3, a) &= \\ q_3 &\in F \end{aligned}$$

**Exemple 9**

Soit  $A$  le  $\Sigma$ -ANF de la figure 2.2. Alors  $cb$  est un exemple de mot appartenant à  $L(A)$ . En effet :

$$\begin{aligned}
\hat{\delta}(q_0, cb) &= \\
ECLOSE(\delta(\hat{\delta}(q_0, c), b)) &= \\
ECLOSE(\delta(ECLOSE(\delta(\hat{\delta}(q_0, \epsilon), c), b)) &= \\
ECLOSE(\delta(ECLOSE(\delta(ECLOSE(q_0, c), b)) &= \\
ECLOSE(\delta(ECLOSE(\delta(\{q_0, q_1, q_2\}, c), b)) &= \\
ECLOSE(\delta(ECLOSE(\{q_0, q_1, q_2\}, b)) &= \\
ECLOSE(\delta(\{q_0, q_1, q_2\}, b)) &= \\
ECLOSE(\{q_1, q_2\}) &= \\
\{q_1, q_2\}
\end{aligned}$$

On a bien l'ensemble d'états  $\{q_1, q_2\}$  avec  $q_2 \in F$ .

L'algorithme 2.2.1 représente cette appartenance pour un mot.

**Algorithme 2.2.1 (Appartenance d'un mot à un langage défini par un automate)**

**Requis:** un mot  $w$ , un automate  $A = (Q, \Sigma, q_0, \delta, F)$  représentant  $L = L(A)$

**Promet:** si  $w$  appartient à  $L$

```

1:  $C \leftarrow \{q_0\}$   $\{C$  est l'ensemble des états courants $\}$ 
2: si  $A$  est un  $\epsilon$ -ANF alors
3:    $C \leftarrow ECLOSE(C)$ 
4: fin si
5: tant que  $|w| > 0$  faire
6:   décomposer  $w$  en  $ax$  avec  $a \in \Sigma$  et  $x$  le reste du mot
7:    $C \leftarrow \delta(C, a)$   $\{\text{passage à l'ensemble des états suivants}\}$ 
8:    $\{\text{Si l'automate est un ADF, } C \text{ ne contient toujours qu'un seul état}\}$ 
9:    $w \leftarrow x$ 
10: si  $A$  est un  $\epsilon$ -ANF alors
11:    $C \leftarrow ECLOSE(C)$ 
12: fin si
13: fin tant que
14: retourner s'il existe un état  $q \in C$  appartenant à  $F$ 

```

**Complexité 2.2.1.1 (Lecture d'un mot par un automate)**

La complexité de l'algorithme 2.2.1 dépend du type d'automate considéré. Considérons que  $|w| = n$ .

L'opération 1 est en temps constant. Les opérations 2 à 4 ont la complexité d'ECLOSE. Pour un ADF ou ANF, cette opération est triviale ( $\mathcal{O}(1)$ ). Pour un  $\epsilon$ -ANF, cette opération est en  $\mathcal{O}(n)$ . En effet,  $n$  états au plus peuvent être ajoutés à l'ensemble.

Les opérations 6 et 9 sont en temps constant. L'opération 7 a la complexité de  $\delta$ . Cette fonction est triviale pour un ADF. Si celui-ci est stocké dans un tableau, l'opération est en temps constant.

Pour un ANF ou  $\epsilon$ -ANF, comme  $\delta$  porte sur un sous-ensemble de  $Q$ , l'opération est en  $\mathcal{O}(n)$ . Le pire des cas étant  $C = Q$  avant ou après l'application de  $\delta$ . Les opérations 10 à 12 sont à nouveau une application de *ECLOSE*.

Dès lors, le pire des cas du corps de la boucle de l'opération 5 est le maximum entre le pire des cas de *ECLOSE* et de  $\delta$ . Cette boucle s'effectue au plus  $n$  fois. Une lettre est enlevée à chaque itération.

L'opération 14 se fait en  $\mathcal{O}(\log_2(n))$  pour un ADF et  $\mathcal{O}(n \log_2(n))$  pour un ANF ou  $\epsilon$ -ANF. Cette valeur suppose une recherche efficace dans  $F$  (en  $\mathcal{O}(\log_2(n))$ ).

En conclusion :

ADF	ANF	$\epsilon$ -ANF
$\mathcal{O}(1) + n(\mathcal{O}(1) + \mathcal{O}(1)) + \mathcal{O}(\log_2(n))$	$\mathcal{O}(1) + n(\mathcal{O}(1) + \mathcal{O}(n)) + \mathcal{O}(n \log_2(n))$	$\mathcal{O}(n) + n(\mathcal{O}(n) + \mathcal{O}(n)) + \mathcal{O}(n \log_2(n))$
$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

## 2.2.5 Équivalence entre expression régulière et automate

Certains problèmes peuvent être exprimés sous forme d'appartenance à un langage régulier, et par extension à une expression régulière. Pouvoir convertir une expression régulière en automate permet d'exécuter cet automate sur une machine pour tester l'appartenance. Ainsi, grâce à cette méthode, une grande classe de problèmes peut être résolue.

**Théorème 2.2.1 (ADF et expression régulière)** *Un langage peut être exprimé par un automate déterministe fini si et seulement si il peut être décrit par une expression régulière.*

Ce théorème étant une double implication, il est vrai si les deux implications le sont. Étudions celles-ci séparément.

**Théorème 2.2.2 (ADF  $\implies$  expression régulière)** *Soit un langage  $L$ . Il existe un automate déterministe  $A$  tel que  $L(A) = L \implies$  il existe une expression régulière  $E$  telle que  $L(E) = L$ .*

### Preuve 2.2.2.1

Soit un langage  $L$ . Supposons qu'il existe un ADF  $A = (Q, \Sigma, q_0, \delta, F)$  tel que  $L(A) = L$ .  $Q$  étant un ensemble fini, on peut définir sa cardinalité :  $|Q| = n$ . Supposons que ses états soient nommés  $\{1, 2, \dots, n\}$ . Il est possible de construire des expressions régulières par induction sur le nombre d'états considérés. De plus, un tel automate est aisément représenté dans un ordre séquentiel, de gauche à droite. Ceci permet de séparer visuellement les  $k$  premiers états du reste.

Soient  $i, j, k \in Q$ , tous équivalents à des nombres inférieurs ou égaux à  $n$ . Définissons  $E_{i,j}^k$  comme étant l'expression régulière décrivant un langage constitué des mots  $w$  tels que  $\hat{\delta}(i, w) = j$  et qu'aucun état intermédiaire n'ait un nombre strictement supérieur à  $k$ .

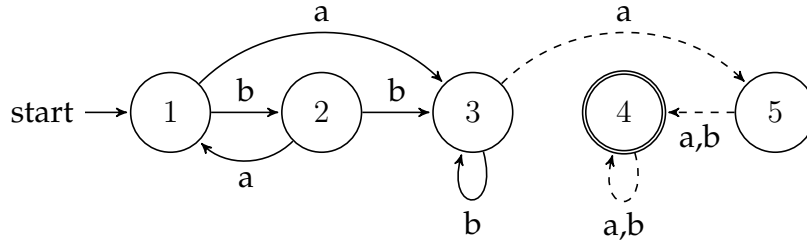


FIGURE 2.3: Exemple : automate mettant  $E_{1,3}^3$  en évidence. Les mots représentés par un chemin passant par un arc discontinu n'appartiennent pas à  $E_{1,3}^3$  : un des états intermédiaires est dénommé par un nombre supérieur à  $k = 3$ .

L'exemple ci-dessus illustre ce fait qu'aucun état supérieur à  $k$  ne peut faire partie des états intermédiaires. Intuitivement, il s'agit d'un automate auquel on a enlevé les transitions :

- Allant de  $i$  à un nombre supérieur à  $k$  (sauf  $j$ )
- Entre deux nombres supérieurs à  $k$
- Allant d'un nombre supérieur à  $k$  (sauf  $i$ ) à  $j$

En pratiques, celles-ci ne sont juste pas considérées lors de l'application de  $\delta$ .

**Cas de base**  $k = 0$ . Comme tout état est numéroté 1 ou plus, aucun intermédiaire n'est accepté.

Une possibilité est  $i \neq j$ . Alors les chemins possibles ne se composent que d'un arc allant directement de  $i$  à  $j$ . Pour les construire :

Pour chaque paire  $i, j$  :

- Il n'existe pas de symbole  $a$  tel que  $\delta(i, a) = j$ . Alors,  $E_{ij}^0 = \emptyset$
- Il existe un unique symbole  $a$  tel que  $\delta(i, a) = j$ . Alors,  $E_{ij}^0 = a$
- Il existe des symboles  $a_1, a_2, \dots, a_k$  tels que  $\forall l \in \{1, \dots, k\}, \delta(i, a_l) = j$ . Alors,  $E_{ij}^0 = a_1 + a_2 + \dots + a_k$

Une autre possibilité est  $i = j$  et indique un chemin de longueur 0. Auquel cas l'expression régulière représentant un chemin sans symbole est  $\epsilon$ . Ce chemin doit être ajouté au langage décrit par  $E_{i,j}^k$  si  $i = j$ .

**Pas de récurrence** Supposons qu'il existe un chemin allant de  $i$  à  $j$  ne passant par aucun état ayant un numéro supérieur à  $k$ . La première possibilité est que le-dit chemin ne passe pas par  $k$ . Alors, le mot représenté par ce chemin fait partie du langage de  $E_{ij}^{k-1}$ . Seconde possibilité, le chemin passe par  $k$  une ou plusieurs fois comme représenté à la figure 2.4.

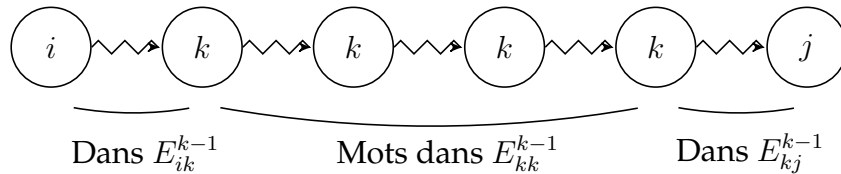


FIGURE 2.4: Un chemin de  $i$  à  $j$  peut être découpé en différent segment en fonction de  $k$



Auquel cas, ces chemins sont composés d'un sous-chemin donnant un mot dans  $E_{ik}^{k-1}$ , suivi d'un sous-chemin donnant un ou plusieurs mots dans  $E_{kk}^{k-1}$  et finalement un mot dans  $E_{kj}^{k-1}$ .

En combinant les expressions des deux types, on obtient :

$$E_{ij}^k = E_{ij}^{k-1} + E_{ik}^{k-1}(E_{kk}^{k-1})^*E_{kj}^{k-1}$$

En commençant cette construction sur  $E_{ij}^n$ , comme l'appel se fait toujours à des chaînes plus courtes, éventuellement on retombe sur le cas de base. Si l'état initial est numéroté 1, alors l'expression régulière  $E$  exprimant  $L$  est l'union (+) des  $E_{1j}^n$  tels que  $j$  est un état acceptant.  $\square$

### Complexité 2.2.2.1

Soit un ADF  $A = (Q, \Sigma, q_0, \delta, F)$  comportant  $n$  états. Pour connaître la complexité totale de cet algorithme, il faut connaître le nombre total d'expressions régulières construites et la longueur de chacune de celles-ci.

A chacune des  $n$  itérations (ajoutant progressivement des nouveaux états admis pour état intermédiaire), la longueur de l'expression peut quadrupler : elle est exprimée par 4 facteurs. Ainsi, après  $n$  étapes, cette expression peut être de taille  $\mathcal{O}(4^n)$ .

Le nombre d'expressions à construire, lui, est décomposable en deux facteurs également : le nombre d'itérations et celui de paires  $i, j$  possibles. Le premier facteur est  $n$ , quand aux paires, leur nombre s'exprime par  $n^2$ .  $n^3$  expressions sont construites.

En regroupant ces deux facteurs, on obtient  $n^3\mathcal{O}(4^n) = \mathcal{O}(n^34^n) = \mathcal{O}(2^n)$ . Comme  $n$  correspond au nombre d'états, si la transformation se fait depuis un ANF, via un ADF, vers une expression régulière, la complexité devient doublement exponentielle. La première transformation étant elle-même exponentielle en le nombre d'états de l'ANF.

### Exemple 10 (Construction d'une expression régulière)

Construction d'une expression régulière à partir de l'automate de la figure suivante :

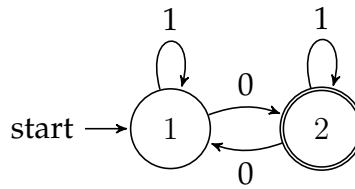


FIGURE 2.5: Un automate acceptant tout mot ayant un nombre impair de 0

La construction par récurrence commençant avec  $k = 0$  le processus peut être représenté par des tableaux correspondant à différents  $k$  de façon croissante.

**Première itération** Dans la première itération, chaque expression se résume à un des trois cas de base, avec éventuellement  $\epsilon$  si  $i = j$  pour l'expression analysée. Ici,  $k = 0$ .

	Cas de base
$E_{11}^0$	$1 + \epsilon$
$E_{12}^0$	$0$
$E_{21}^0$	$0$
$E_{22}^0$	$1 + \epsilon$

**Seconde itération** Ensuite, l'état 1 est autorisé comme état intermédiaire :  $k = 1$ . Ayant potentiellement un état intermédiaire, la formule de récurrence est utilisée.

	Formule de récurrence	Détail	Simplification
$E_{11}^1$	$E_{11}^0 + E_{11}^0 (E_{11}^0)^* E_{11}^0$	$(1 + \epsilon) + (1 + \epsilon)(1 + \epsilon)^*(1 + \epsilon)$	$1^*$
$E_{12}^1$	$E_{12}^0 + E_{11}^0 (E_{11}^0)^* E_{12}^0$	$0 + (1 + \epsilon)(1 + \epsilon)^* 0$	$1^* 0$
$E_{21}^1$	$E_{21}^0 + E_{21}^0 (E_{11}^0)^* E_{11}^0$	$0 + 0(1 + \epsilon)^*(1 + \epsilon)$	$01^*$
$E_{22}^1$	$E_{22}^0 + E_{21}^0 (E_{11}^0)^* E_{12}^0$	$(1 + \epsilon) + 0(1 + \epsilon)^* 0$	$\epsilon + 1 + 01^* 0$

**Troisième itération** A la troisième itération, l'état 2 est autorisé comme état intermédiaire ( $k = 2$ ).

	Résultat
$E_{11}^2$	$1^* + 1^* 0(1 + 01^* 0)^* 01^*$
$E_{12}^2$	$1^* 0(1 + 01^* 0)^*$
$E_{21}^2$	$(1 + 01^* 0)^* 01^*$
$E_{22}^2$	$(1 + 01^* 0)^*$

Pour obtenir une expression régulière correspondant à l'automate, on s'intéresse à celle qui décrit un chemin entre l'état initial (1) et les états acceptants (uniquement 2 ici). Dès lors,  $E_{12}^2 = 1^* 0(1 + 01^* 0)^* = L$ .

Cette expression régulière  $1^* 0(1 + 01^* 0)^*$  décrit bien un nombre impair de 0. Il en faut absolument un, et tout ajout supplémentaire se fait par paire.

**Théorème 2.2.3 (Expression régulière  $\implies$  ADF)** ( $\Leftarrow$ ) Soit un langage  $L$ . Il existe une expression régulière  $E$  telle que  $L(E) = L \implies$  il existe un automate déterministe  $A$  tel que  $L(A) = L$ .

### Preuve 2.2.3.1

Comme tout  $\epsilon$ -ANF a un ADF équivalent (théorème 2.2.4), montrer qu'une expression régulière  $E$  a un ANF équivalent est suffisant pour obtenir cet ADF.

Soit un langage  $L$ . Soit  $E$  une expression régulière telle que  $L(E) = L$ . On peut construire l'automate récursivement sur la définition des expressions régulières à la section 2.1.3. Cette preuve par récurrence repose sur trois invariants portant sur chaque ANF construit :

1. Il y a un unique état acceptant
2. Aucune transition ne mène à l'état initial
3. Aucune transition ne part de l'état acceptant

Intuitivement, ces invariants servent à assurer que l'automate ainsi créé est lu de gauche à droite tel une expression régulière.

**Cas de base** Les ANFs de la figure 2.6 représentent les automates correspondant aux trois cas de base.

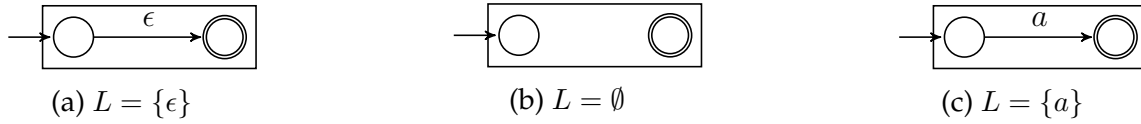


FIGURE 2.6: Blocs de base pour la construction d'un automate à partir d'une expression régulière

En effet, l'automate (a) représente le langage  $\{\epsilon\}$  égal à  $L(E)$  : le seul arc de l'état initial à un état final est  $\epsilon$ . L'automate (b) ne propose pas d'arc atteignant l'état final. Aucun mot n'appartient au langage représenté par cet automate qui vaut donc  $\emptyset = L(\emptyset)$ . Finalement, (c) propose un arc pour  $a$ . Dès lors, il existe un unique chemin de longueur 1 correspondant au mot  $a$ . Ainsi, le langage exprimé par cet automate  $\{a\}$  est bien égal à  $L(E) = L(a)$ . De plus, ces automates respectent bien l'invariant de récurrence proposé.

**Pas de récurrence** Les ANF *abstrait*s de la figure 2.7 représentent la façon dont un automate peut être construit récursivement en fonction des règles de récurrence des expressions régulières. Ces ANF sont abstraits car le contenu d'un bloc  $E$  ou  $F$  n'est pas représenté explicitement. Cependant, celui-ci respecte les invariants de récurrence.

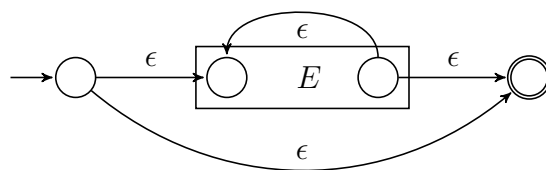
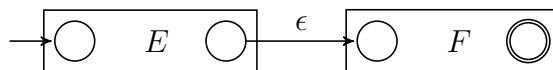
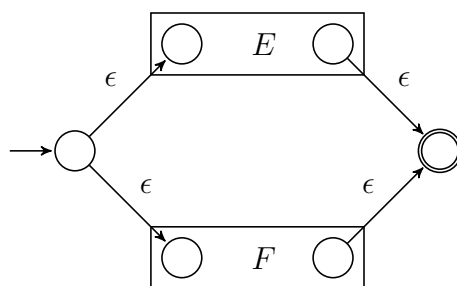
(a)  $L = L(E)^*$ (b)  $L = L(E)L(F)$ (c)  $L = L(E) + L(F)$ 

FIGURE 2.7: Enchaînement de blocs pour une construction récursive

Les quatre règles de récurrence sur une expression régulière permettent de construire les automates :

- $L((E)) = L(E)$  ne nécessite pas de construction supplémentaire.
- $L(E^*) = L(E)^*$  est construit comme en (a). En effet, l'arc revenant au début de  $E$  permet d'exprimer  $E^1, E^2, E^3, \dots$
- $L(EF) = L(E)L(F)$  est construit comme en (b). En effet, tout mot de cet automate est de la forme  $v\epsilon w$  avec  $v \in L(E)$  et  $w \in L(F)$ .
- $L(E + F) = L(E) \cup L(F)$  est construit comme en (c). En effet, tout mot de cet automate est de la forme  $\epsilon v \epsilon$  ou  $\epsilon w \epsilon$  avec  $v \in L(E)$  et  $w \in L(F)$ , en accord avec la définition de l'union ensembliste.

Les automates (a), (b) et (c) respectent bien l'invariant de récurrence : pas de transition vers l'état initial, un seul état acceptant n'ayant pas de transition sortante. Chaque automate abstrait pour  $E$  ou  $F$  peut lui même être construit récursivement jusqu'au cas de base.

□

### Complexité 2.2.3.1

Soit une expression régulière  $E$  contenant  $n$  symboles (alphabet et opérations comprises) représentant un langage  $L = L(E)$ . La construction d'un ANF pour  $L$  peut se faire en  $\mathcal{O}(n)$ . En effet :

- Cas de base : Au plus  $n$  ANFs sont créés. Chacun correspond à un symbole (opérations non comprises). Chaque ANF a un état, rendant la création en temps constant. La création de tous ces ANFs est alors en  $\mathcal{O}(n)$
- Récurrence : Il reste au plus  $n$  symboles correspondant à des opérations, ce qui implique au plus  $n$  opérations. Chaque opération se base sur les 4 règles de récurrences définies précédemment. Dans les cas nécessitant une construction, celle-ci peut se faire en temps constant (ajout d'au plus deux états et quatre transitions). Chaque opération n'est à effectuer qu'une seule fois, consommant le symbole, et que chacune se fait en temps constant. Dès lors, la totalité des étapes de récurrence se fait au plus en  $\mathcal{O}(n)$

La complexité totale de cette conversion est en  $\mathcal{O}(n)$  vers un ANF. La conversion vers un ADF, comme mentionné dans la section ci-après peut quand à elle être exponentielle.

### 2.2.6 Équivalence entre un automate déterministe fini et un automate non-déterministe fini

Cette section présente une méthode permettant de créer un ADF à partir d'un ANF et réciproquement. Ici, ANF est considéré au sens-large et peut tout aussi bien être un ANF normal qu'un  $\epsilon$ -ANF. Ceci permet de justifier l'abstraction faite entre les différents types d'automates finis et l'utilisation du même terme pour tous ceux-ci.

Soit un ANF  $A = (Q, \Sigma, q_0, \delta, F)$ . Alors l'ADF équivalent

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

est défini par :

- $Q_D = \{T \mid T = \text{ECLOSE}(S) \text{ et } S \subseteq Q\}$ . Concrètement,  $Q_D$  est l'ensemble des parties de  $Q$  fermées sur  $\epsilon$ . Ceci qui signifie que chaque transition sur  $\epsilon$  depuis un état de  $T$  mène à un état également dans  $T$ . L'ensemble  $\emptyset$  est fermé sur  $\epsilon$ .
- $q_D = \text{ECLOSE}(q_0)$ . L'état initial de  $D$  est l'ensemble des états dans la fermeture sur  $\epsilon$  des états de  $A$ .
- $F_D = \{T \mid T \in Q_D \text{ et } T \cap F \neq \emptyset\}$  contient les ensembles dont au moins un état est acceptant pour  $A$ .
- $\delta_D(T, a)$  est construit,  $\forall a \in \Sigma, \forall T \in Q_D$  par :
  1. Soit  $T = \{p_1, p_2, \dots, p_k\}$ .
  2. Calculer  $\bigcup_{i=1}^k \delta(p_i, a)$ . Renommer cet ensemble en  $\{r_1, r_2, \dots, r_m\}$ .
  3. Alors  $\delta_D(T, a) = \bigcup_{j=1}^m \text{ECLOSE}(r_j)$ .

#### Exemple 11 (Transformation ANF vers ADF)

Considérons l'automate  $A = (Q, \Sigma, q_0, \delta, F)$  de l'exemple 4 et les fermetures calculées dans l'exemple 6.

Alors, l'automate  $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$  est donné par :

- $Q_D = \{\emptyset, \{q_1\}, \{q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$ . Les ensembles  $\{q_0, q_1\}$  et  $\{q_0, q_2\}$  sont des sous-ensembles de  $Q$  mais ne sont pas fermés sur  $\epsilon$ .
- $q_D = \{q_0, q_1, q_2\} = \text{ECLOSE}(q_0)$ .

- $F_D = \{\{q_2\}, \{q_1, q_2\}, \{q_0, q_1, q_2\}\}$ , les ensembles contenant  $q_2$ , étant acceptant de  $A$ .
- $\delta_D$  est exprimé sur le graphe de la figure 2.8.

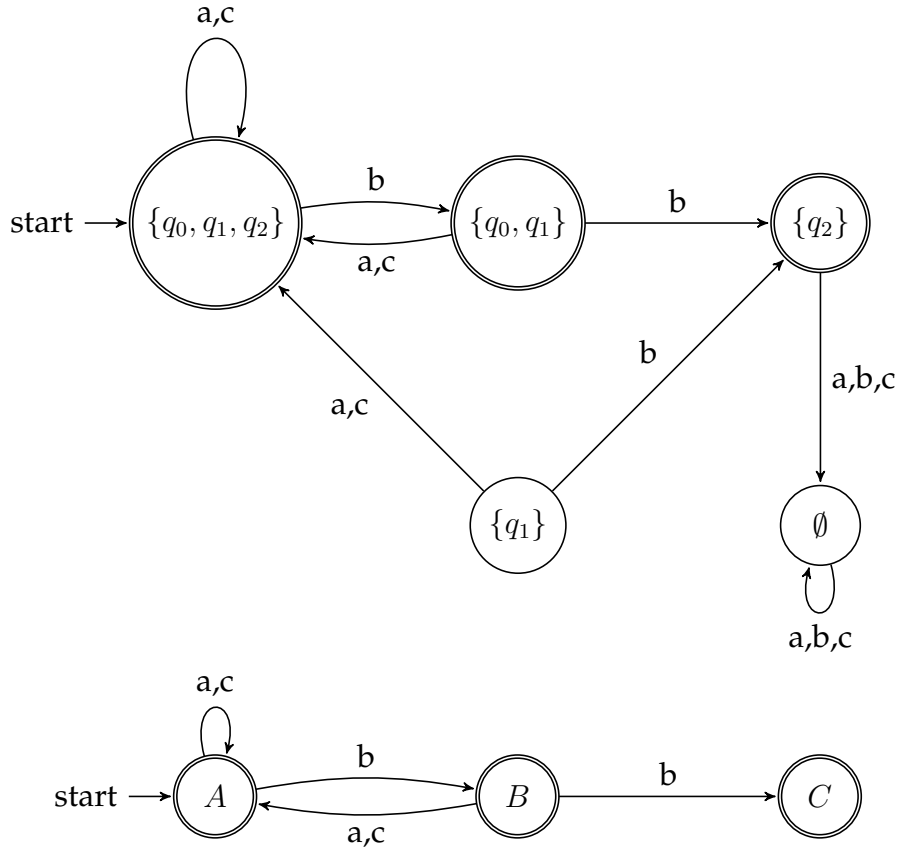


FIGURE 2.8: Automate  $D$ . De par la construction de  $Q$ , le nombre d'états de  $D$  est exponentiel. Les états inatteignables et  $\emptyset$  sont souvent omis pour clarifier la représentation.

**Théorème 2.2.4 (ANF  $\Leftrightarrow$  ADF)** *Un langage  $L$  peut être représenté par un ANF si et seulement si il peut l'être par un ADF.*

#### Preuve 2.2.4.1

Soit  $L$  un langage. Cette preuve étant une double implication, chacune peut être prouvée séparément.

( $\Leftarrow$ )  $L$  peut être représenté par un ADF  $\Rightarrow L$  peut être représenté par un ANF. Supposons qu'un automate  $D = (Q_D, \Sigma, \delta_D, q_D, F_D)$  représente  $L : L(D) = L$ . L'ANF  $A = (Q, \Sigma, q_0, \delta, F)$  correspondant est construit comme suit :

- $Q = \{\{q\} | q \in Q_D\} \cup \emptyset$
- $\delta$  contient les transitions de  $D$  modifiées. Les objets retournés deviennent des ensembles d'états. C'est-à-dire, si  $\delta_D(q, a) = p$  alors  $\delta(\{q\}, a) = \{p\}$ . De plus, pour chaque état  $q \in Q_D$ ,  $\delta(\{q\}, \epsilon) = \emptyset$ .
- $q_0 = \{q_D\}$
- $F = \{\{q\} | q \in F_D\}$

Dès lors, les transitions sont les mêmes entre  $D$  et  $A$ , mais  $A$  précise explicitement qu'il n'y a pas de transition sur  $\epsilon$ . Comme  $A$  représente le même langage, il existe donc bien un ANF qui représente  $L$ .

( $\Rightarrow$ )  $L$  peut être représenté par un ANF  $\implies L$  peut être représenté par un ADF. Soit l'automate  $A = (Q, \Sigma, q_0, \delta, F)$ . Supposons qu'il représente  $L = L(A)$ . Considérons l'automate obtenu par la transformation détaillée à la section précédente (page 20) :

$$D = (Q_D, \Sigma, \delta_D, q_D, F_D)$$

Montrons que  $L(D) = L(A)$ . Pour ce faire, montrons que les fonctions de transition étendues sont équivalentes. Auquel cas, les chemins sont équivalents et donc les langages sont égaux. Montrons que  $\hat{\delta}(q_0, w) = \hat{\delta}_D(q_D, w)$  pour tout mot  $w$ , par récurrence sur  $w$ .

**Cas de base** Si  $|w| = 0$ , alors  $w = \epsilon$ .  $\hat{\delta}(q_0, \epsilon) = \text{ECLOSE}(q_0)$ , par définition de la fonction de transition étendue.  $q_D = \text{ECLOSE}(q_0)$  par la construction de  $q_D$ . Pour un ADF (ici,  $D$ ),  $\hat{\delta}(p, \epsilon) = p$ , pour tout état  $p$ . Par conséquent,  $\hat{\delta}_D(q_D, \epsilon) = q_D = \text{ECLOSE}(q_0) = \hat{\delta}(q_0, \epsilon)$ .

**Pas de récurrence** Supposons  $w = xa$  avec  $a$  le dernier symbole de  $w$ . Notre hypothèse de récurrence est que  $\hat{\delta}_D(q_D, x) = \hat{\delta}(q_0, x)$ . Ce sont bien les mêmes objets car  $\hat{\delta}_D$  retourne un état de  $D$  qui correspond à un ensemble d'états de  $A$ . Notons celui-ci  $\{p_1, p_2, \dots, p_k\}$ . Par définition de  $\hat{\delta}$  pour un ANF,  $\hat{\delta}(q_0, w)$  est obtenu en :

1. Construisant  $\{r_1, r_2, \dots, r_m\} = \bigcup_{i=1}^k \delta(p_i, a)$ . Cet ensemble correspond aux états obtenus par la lecture du symbole  $a$  à partir de  $\{p_1, p_2, \dots, p_k\}$ .
2. Calculant  $\hat{\delta}(q_0, w) = \bigcup_{j=1}^m \text{ECLOSE}(r_j)$ . Un état atteint par la lecture de  $a$  l'est aussi par  $a\epsilon$ .

$D$  a été construit avec ces deux mêmes étapes pour  $\delta_D(\{p_1, p_2, \dots, p_k\}, a)$ . Dès lors,  $\hat{\delta}_D(q_D, w) = \delta_D(\{p_1, p_2, \dots, p_k\}, a) = \bigcup_{j=1}^k \text{ECLOSE}(p_j) = \hat{\delta}(q_0, w)$ .

On a bien  $\hat{\delta}_D(q_D, w) = \hat{\delta}(q_0, w)$ .

Les langages sont donc bien égaux.

□

### Complexité 2.2.4.1 (Conversion ANF vers ADF)

La complexité d'une conversion ANF vers ADF peut être exprimée en fonction de  $n$  le nombre d'états de l'ANF. La taille de l'alphabet  $\Sigma$  est ici comptée comme une constante  $k$ . Elle est ignorée dans l'analyse grand-O. L'algorithme de conversion se fait en deux étapes. Le calcul de ECLOSE et la construction à proprement parler. Ici, l'automate est stocké sous forme d'une table de transitions. Cette solution est plus facile à manipuler mais peut engendrer un surcoût en mémoire, qui n'est pas analysé ici.

- ECLOSE : Chacun des  $n$  états ayant une entrée pour  $\epsilon$  dans la fonction  $\delta$ , le temps de calcul sur chaque nœud ajouté est en temps constant. Chacune des  $n$  fermetures pour chacun des  $n$  états  $q \in Q$  pouvant au plus compter les  $n$  états, le temps total de cette opération est en  $n\mathcal{O}(n) = \mathcal{O}(n^2)$ .

- Construction : Posons  $s$  le nombre d'états dans l'ADF (qui, dans le pire des cas vaut  $s = 2^n$  par la construction des sous-ensembles). La création d'un état de l'ADF est en  $\mathcal{O}(n)$ . En effet, il faut garder des références vers les états de l'ANF concernés. Ceux-ci sont au plus  $n$ .

Comme il y a  $s$  états dans l'ADF, il y a  $ks$  transitions. Chacune de celle-ci peut être construite en  $\mathcal{O}(n)$ . En effet, chaque état de l'ADF étant constitué d'au plus  $n$  états de l'ANF, il y a au plus  $n$  transitions à suivre pour obtenir l'ensemble d'états résultant dans l'ANF. Cet ensemble correspond alors à un état de l'ADF obtenu. Les transitions sont construites en  $\mathcal{O}(nks) = \mathcal{O}(nks)$ .  $k$  est toujours considéré comme une constante.

La complexité dans le pire des cas est  $\mathcal{O}(n^2) + \mathcal{O}(sn) + \mathcal{O}(sn) = \mathcal{O}(sn) = \mathcal{O}(n2^n)$ . Le détail est donné sur  $s$  car, comme prouvé dans [3], en pratique le nombre d'états dans l'ADF obtenu est rarement de l'ordre de  $2^n$ , typiquement de l'ordre de  $n$ . Dans ce cas là, la complexité devient  $\mathcal{O}(n^2)$ .

#### Complexité 2.2.4.2 (Conversion ADF vers ANF)

La conversion d'un ADF  $A = (Q, \Sigma, q_0, \delta, F)$  vers un ANF consiste au remplacement de  $n$  états par  $n$  ensembles d'un seul état. Chaque copie individuelle étant en temps constant, cette opération est en  $\mathcal{O}(n)$ . Ensuite, une nouvelle table de transition doit être créée. Si l'alphabet  $\Sigma$  est de taille  $k$ , celle-ci a toujours  $n$  lignes mais  $k + 1$  colonnes. En effet, une colonne est ajoutée pour  $\epsilon$ . La création de cette nouvelle table se fait alors en  $\mathcal{O}(kn)$ . La complexité totale d'une conversion d'un ADF vers un ANF est en  $\mathcal{O}(kn)$ .

## 2.3 Table Filling Algorithm

Cette section décrit le *Table Filling Algorithm*, algorithme permettant de minimiser un automate déterministe mais aussi de tester l'équivalence entre deux de ceux-ci. C'est un élément important pour l'algorithme d'Angluin, sujet principal de ce mémoire.

### 2.3.1 Relation $R_F$

Soit un ADF  $A = (Q, \Sigma, q_0, \delta, F)$ . Définissons la relation  $R_F$  entre deux états :

$$xR_Fy \iff (\forall w \in \Sigma^*, \hat{\delta}(x, w) \in F \iff \hat{\delta}(y, w) \in F)$$

Intuitivement, ces deux états sont en relation si tout mot lu à partir de ceux-ci mène à des états étant simultanément acceptants ou non.

**Proposition 2.3.1 ( $R_F$ )**  $R_F$  est une relation d'équivalence.

*Preuve 2.3.1.1 ( $R_F$  est une relation d'équivalence)*

Montrer que  $R_F$  est une relation d'équivalence revient à montrer qu'elle est réflexive, transitive et symétrique.

- **Réflexive** : Soient un état  $x \in Q_M$  et un mot  $w \in \Sigma^*$ . Alors,  $\hat{\delta}(x, w) \in F \iff \hat{\delta}(x, w) \in F$  et par définition,  $xR_Fx$ .



- **Transitive** : Soient les états  $x, y, z \in Q_M$  tels que  $xR_E y$  et  $yR_E z$  ainsi que  $w \in \Sigma^*$ . Par hypothèse,  $\hat{\delta}(x, w) \in F \iff \hat{\delta}(y, w) \in F$  et  $\hat{\delta}(y, w) \in F \iff \hat{\delta}(z, w) \in F$ . Par transitivité de l'implication, on obtient  $\hat{\delta}(x, w) \in F \iff \hat{\delta}(z, w) \in F$ . On a donc  $xR_E z$ .
- **Symétrique** : Soient les états  $x, y \in Q_M$  tels que  $xR_E y$  et un mot  $w \in \Sigma^*$ . Par hypothèse,  $\hat{\delta}(x, w) \in F \iff \hat{\delta}(y, w) \in F$ . En lisant la double implication depuis la droite, on a bien  $\hat{\delta}(y, w) \in F \iff \hat{\delta}(x, w) \in F$  et donc  $yR_E x$ .

□

### Corollaire 2.3.1.1

$R_F$  répartit les états de  $Q$  en classes d'équivalence.

La classe d'équivalence de tous les états en relation  $R_F$  avec  $q$  (qui sert alors de *représentant*) se note  $[[q]]$  ou par une lettre majuscule, typiquement  $S$  ou  $T$ .

### Corollaire 2.3.1.2

Si  $Q$  est fini, alors le nombre de classe d'équivalences est fini aussi. Chaque classe d'équivalence  $[[q]]$  contient un nombre d'états fini.

**Proposition 2.3.2 (Congruence de  $R_F$ )**  $R_F$  est congruente à droite, c'est-à-dire

$$xR_F y \implies \forall a \in \Sigma, \delta(x, a)R_F \delta(y, a)$$

### Preuve 2.3.2.1 (Congruence de $R_F$ )

Montrons que si la relation est vraie pour deux états, elle reste valable pour les états atteints par la lecture d'un symbole quelconque. Soient les états  $x, y \in Q_M$  tels que  $xR_F y$ . Soit un symbole  $a \in \Sigma$ . Par hypothèse,

$$\forall w \in \Sigma^*, \hat{\delta}(x, w) \in F \iff \hat{\delta}(y, w) \in F$$

C'est donc vrai en particulier pour  $w = au, u \in \Sigma^*$ . Dès lors,

$$\hat{\delta}(x, au) \in F \iff \hat{\delta}(y, au) \in F$$

$$\hat{\delta}(\delta(x, a), u) \in F \iff \hat{\delta}(\delta(y, a), u) \in F$$

$$\hat{\delta}(p, u) \in F \iff \hat{\delta}(q, u) \in F$$

□

### Corollaire 2.3.2.1

Toutes les transitions étiquetées par un symbole  $a$  sortant d'une classe d'équivalence mènent à une même classe d'équivalence :  $\forall$  classe d'équivalence  $S, \forall a \in \Sigma, \exists$  une classe d'équivalence  $T, \forall q \in S, \delta(q, a) \in T$ .

### 2.3.2 Algorithme

Certains états d'un automate peuvent être *équivalents* selon la relation  $R_F$ . L'information que ceux-ci proposent est alors redondante. Dès lors, l'automate peut être simplifié pour offrir la même information de façon plus compacte. Une façon de détecter ces équivalences est de construire un tableau via le *table filling algorithm*. Le tableau obtenu est la *table de différenciation*.

Celui-ci détecte les paires *différenciables* récursivement sur un automate  $A = (Q, \Sigma, q_0, \delta, F)$ . Une paire  $\{p, q\}$  est différenciable s'il existe un mot  $w$  tel que  $\hat{\delta}(p, w)$  est un état acceptant et  $\hat{\delta}(q, w)$  ne l'est pas ou vice-versa.  $w$  sert alors de *mot témoin*. L'algorithme procède récursivement comme suit :

**Cas de base :** Si  $p$  est un état acceptant et que  $q$  ne l'est pas, alors la paire  $\{p, q\}$  est différenciable. Le mot témoin est  $\epsilon$ . La cellule correspondante dans le tableau est cochée.

**Pas de récurrence :** Soient  $p, q, r, s$  des états de  $Q$  et un symbole  $a \in \Sigma$  tel que  $\delta(p, a) = r$  et  $\delta(q, a) = s$ . Si  $r$  et  $s$  sont différenciables (les cellules sont cochées dans le tableau) alors  $p$  et  $q$  le sont aussi. En effet, il existe un mot *témoin*  $w$  qui permet de différencier  $r$  et  $s$ . Alors le mot  $aw$  est le mot témoin qui permet de différencier  $p$  et  $q$ . Les cellules de ces états sont cochées.

**Théorème 2.3.3 (Table de différenciation)** *Si deux états ne sont pas différenciés par le table filling algorithm, les états sont équivalents (ils respectent la relation  $R_F$ ).*

#### Exemple 12 (Table de différenciation)

Voici une application du table filling algorithm sur l'automate  $A_2$ , version réduite de l'automate  $A_1$  de la figure 2.1.

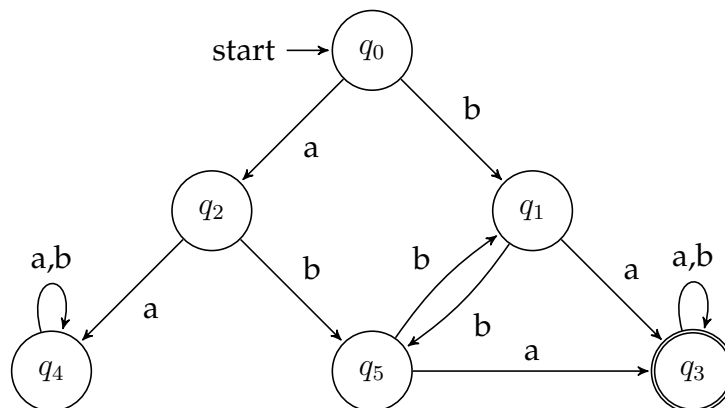


FIGURE 2.9: Automate  $A_2$

La première étape est de remplir la table de différenciation avec l'algorithme précédent. Tout état est différenciable de  $q_3$  : il est le seul état acceptant et tous les autres ne le sont pas. 5 cases peuvent déjà être cochées. Le reste de la table est remplie par induction comme représenté dans la table 2.3.

$q_1$					
$q_2$					
$q_3$	X	X	X		
$q_4$				X	
$q_5$				X	
	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$

(a) Cas de base : tous les états sont différents de  $q_3$ 

$q_1$	X				
$q_2$		X			
$q_3$	X	X	X		
$q_4$		X		X	
$q_5$	X		X	X	X
	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$

(b) Première itération : Les nouvelles paires d'états différenciables mènent, via un symbole  $a \in \Sigma$  à deux autres états différenciables.

$q_1$	X				
$q_2$		X			
$q_3$	X	X	X		
$q_4$	X	X	X	X	
$q_5$	X		X	X	X
	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$

(c) Deuxième itération

$q_1$	X				
$q_2$	X	X			
$q_3$	X	X	X		
$q_4$	X	X	X	X	
$q_5$	X		X	X	X
	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$

(d) Troisième itération :  $q_1$  et  $q_5$  ne sont pas différenciés.TABLE 2.3: Table de différenciation de l'automate  $A_2$  2.9

D'après le théorème, comme  $q_1$  et  $q_5$  ne sont pas différenciés, on a  $q_1 R_F q_5$ .

### Preuve 2.3.3.1

Considérons un automate déterministe fini quelconque  $A = (Q, \Sigma, q_0, \delta, F)$ . Supposons par l'absurde qu'il existe une paire d'états  $\{p, q\}$  telle que :

1.  $p$  et  $q$  ne sont pas différenciés par l'algorithme de remplissage de table.
2. Les états ne sont pas équivalents :  $\neg(p R_E q)$ . Par extension, il existe un mot témoin  $w$  différenciant  $p$  et  $q$ .

Une telle paire est une *mauvaise paire*. Si il y a des mauvaises paires, chacune associée à un mot témoin, il doit exister un paire distinguée par un mot témoin le plus court. Posons  $\{p, q\}$  comme étant cette paire et  $w = a_1 a_2 \dots a_n$  le mot témoin le plus court montrant que  $\neg(p R_E q)$ . Dès lors, soit  $\hat{\delta}(p, w)$  est acceptant, soit  $\hat{\delta}(q, w)$  l'est, mais pas les deux.

Ce mot  $w$  ne peut pas être  $\epsilon$ . Auquel cas, la table aurait été remplie dès le cas de base de l'algorithme avec la paire différenciable  $\{p, q\}$ . La paire  $\{p, q\}$  ne serait pas une mauvaise paire.

$w$  n'étant pas  $\epsilon$ ,  $|w| \geq 1$ . Considérons les états  $r = \delta(p, a_1)$  et  $s = \delta(q, a_1)$ . Ces états sont différenciés par  $a_2 a_3 \dots a_n$  car  $\hat{\delta}(p, w) = \hat{\delta}(r, a_2 a_3 \dots a_n)$  et  $\hat{\delta}(q, w) = \hat{\delta}(s, a_2 a_3 \dots a_n)$  et  $p$  et  $q$  sont différenciables.

Cela signifie qu'il existe un mot plus petit que  $w$  qui différencie deux états : le mot  $a_2 a_3 \dots a_n$ . Comme on a supposé que  $w$  est le mot le plus petit qui différencie une mauvaise paire,  $r$  et  $s$  ne peuvent pas être une mauvaise paire. Donc, l'algorithme a du découvrir qu'ils sont différenciables.

Cependant, le pas de récurrence impose que  $\delta(p, a_1)$  et  $\delta(q, a_1)$  mènent à deux états différenciables implique que  $p$  et  $q$  le sont aussi. On a une contradiction de notre hypothèse :  $\{p, q\}$  n'est pas une mauvaise paire.

Ainsi, s'il n'existe pas de mauvaise paire, c'est que chaque paire différenciable est reconnue par l'algorithme. □

### Complexité 2.3.1.1

Considérons  $n$  le nombre d'états d'un automate, et  $k$  la taille de l'alphabet  $\Sigma$  supporté.

Considérons aussi une version optimisée de l'algorithme. Une optimisation simple est de construire pour chaque paire  $\{r, s\}$  une liste des paires *dépendantes*  $\{p, q\}$  telles que, pour un même symbole  $a$ ,  $\delta(p, a) = r$  et  $\delta(q, a) = s$ .

Cette liste peut être construite en considérant chaque symbole  $a \in \Sigma$  et ajoutant les paires  $\{p, q\}$  à chacune de leur dépendance  $\{\delta(p, a), \delta(q, a)\}$ . Cette étape prend au plus  $k \cdot \mathcal{O}(n^2) = \mathcal{O}(kn^2)$ . (Le nombre de symboles multiplié par le nombre de paires à considérer).

Dès lors, on peut prendre la récurrence et la propager en ajoutant les paires dépendantes de celles différenciées au cas de base. Celui-ci s'effectue en  $\mathcal{O}(n^2)$  opérations, cochant au plus la moitié des paires ( $\frac{n(n-1)}{2}$ ). Ayant au plus  $\frac{n(n-1)}{2}$  paires à atteindre, il y a au plus de l'ordre de  $\mathcal{O}(n^2)$  opérations.

La complexité totale alors revient à  $\mathcal{O}(kn^2)$ .

## 2.3.3 Appartenance et équivalence

Comme mentionné en début de section, le test d'équivalence entre deux automates est un des piliers de l'algorithme d'Angluin. Comparer des langages représentés de façon ensembliste n'est pas toujours possible et rarement efficace. Grâce à de légères modifications apportées à l'algorithme de minimisation 2.3.3, il est possible de comparer deux automates déterministes finis et déterminer si ceux-ci sont équivalents. L'algorithme obtenu est seulement en temps quadratique par rapport au nombre d'états de l'ADF.

Considérons les ADF  $A_H$  et  $A_I$  donnés dans les figures 2.10 et 2.11. Les états ont été renommés pour simplifier la suite de l'exemple.

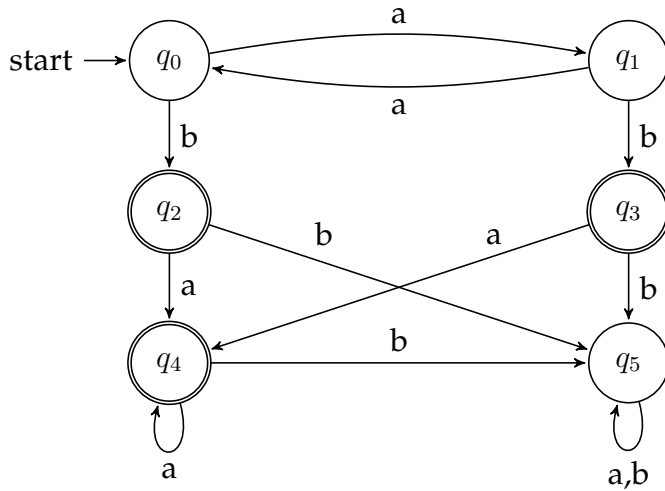


FIGURE 2.10: Automate  $A_H$ , du livre [4] (Fig3.2)

Il est possible de remplir un tableau via le table filling algorithm. Pour ce faire, les deux ADFs sont considérés comme un seul dont les états sont disjoints.

$q_1$								
$q_2$	x	x						
$q_3$	x	x						
$q_4$	x	x						
$q_5$	x	x	x	x	x			
$q_6$			x	x	x	x		
$q_7$	x	x				x	x	
$q_8$	x	x	x	x	x		x	x
	$q_0$	$q_1$	$q_2$	$q_3$	$q_4$	$q_5$	$q_6$	$q_7$

FIGURE 2.12: Tableau généré par l'application de l'algorithme sur  $A_H$  et  $A_I$

De cette table, toujours grâce aux conclusions précédentes, il est possible d'extraire des classes d'équivalences :

- $C_0 = \{q_0, q_1, q_6\}$
- $C_1 = \{q_2, q_3, q_4, q_7\}$
- $C_2 = \{q_5, q_8\}$

En particulier, la classe  $C_0$  souligne que les états initiaux sont équivalents. Cela signifie, par définition, que tout mot  $w$  lu en partant d'un de ces états sera soit accepté dans les deux automates, soit refusé dans les deux.  $A_H$  et  $A_I$  définissent donc le même langage.

Écrivons concrètement l'algorithme de test d'équivalence entre deux automates déterministes finis.

### Algorithme 2.3.2 (Équivalence entre deux automates)

Soient les ADFs  $A = (Q, \Sigma, q_0, \delta, F)$  et  $B = (Q_B, \Sigma_b, q_b, \delta_b, F_b)$ . Des automates utilisant des alphabets différents représenteront probablement des langages différents mais pas nécessairement.

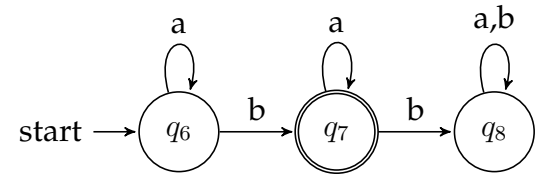


FIGURE 2.11: Automate  $A_I$ , provenant également de [4].

1. Considérer les deux automates comme un seul automate disjoint. Le choix de l'état initial de l'automate ainsi construit n'a pas d'importance, qu'il soit  $q_0$  ou  $q_b$ .
2. Construire la table de différenciation par le table filling algorithm.
3. Si  $q_0$  et  $q_b$  sont équivalents (non différenciés par la table), alors  $A$  et  $B$  sont équivalents.

### Complexité 2.3.2.1

Reposant sur la construction de la table d'équivalence d'états, la complexité est en  $\mathcal{O}(kn^2)$ , avec  $k$  la taille de l'alphabet et  $n$  le nombre d'états. L'étape supplémentaire, la lecture de cette table, est en temps constant et n'impacte pas la complexité.

Les différentes notions liées à l'égalité : les propriétés de réflexivité, transitivité et symétrie ont été démontrées dans la section 2.3.1.

## 2.3.4 Minimisation

Plusieurs automates peuvent représenter un même langage. Parmi ceux-ci, l'*automate minimal* est celui comportant le moins d'états.

La minimisation d'automate se fait en deux étapes :

1. Se débarrasser de tous les états inatteignables : ils ne participent pas à la construction du langage représenté
2. Grâce aux équivalences d'états trouvées grâce au table filling algorithm défini au point 2.3.2, construire un nouvel automate.

Décrivons en détail cette minimisation dans l'algorithme 2.3.3.

### Algorithme 2.3.3 (Minimisation)

Soit un automate déterministe fini  $A = (Q, \Sigma, q_0, \delta, F)$ . Les états inatteignables peuvent être supprimés de  $Q$  et de  $\delta$ .

Pour minimiser cet automate, il faut :

1. Construire la table de différenciation.
2. Séparer  $Q$  en classes d'équivalences.
3. Construire l'automate minimal  $C = (Q_C, \Sigma, \delta_C, q_C, F_C)$  :
  - (a) Soit  $S$  une des classes d'équivalence obtenues par la table de différenciation.
  - (b) Ajouter  $S$  à  $Q_C$  ainsi qu'à  $F_C$  si  $S$  contient un état acceptant  $q \in F$ . Cette opération est valide, comme mentionné dans le corollaire 2.3.2.1.
  - (c) Si  $S$  contient  $q_0$  l'état initial de  $A$ , alors  $S$  est  $q_C$  l'état initial de  $C$ .
  - (d) Pour un symbole  $a \in \Sigma$ , alors il doit exister une classe d'équivalence  $T$  tel que pour chaque état  $\forall q \in S, \delta(q, a) \in T$  par le corollaire 2.3.2.1. On définit alors  $\delta_C(S, a) = T$ .

### Exemple 13

Considérons l'automate  $A_1$  représenté à la figure 2.1. En supprimant l'état  $q_6$  qui n'est pas atteignable, on obtient l'automate  $A_2$  de la figure 2.9.

Le tableau 2.3 sert d'exemple pour l'algorithme de remplissage de tableau, sur  $A_2$ .

En appliquant l'algorithme de minimisation ci-dessus, qui peut se résumer intuitivement à fusionner les états équivalents  $q_1$  et  $q_5$ , on obtient l'automate  $A_3$  de la figure 2.13.



FIGURE 2.13: Automate  $A_3$

Une expression régulière  $((b + ab)b^*a(a + b)^*)$  peut être déduite pour  $L = L(A_3)$  grâce à cet automate  $A_3$ . Cette expression régulière est celle de l'exemple 2

**Théorème 2.3.4 (Minimalité de l'automate réduit)** *Soit un ADF  $A$  et soit  $C$  l'automate construit par cet algorithme de minimisation. Aucun automate équivalent à  $A$  n'a moins d'états que  $C$ . De plus, chaque automate ayant autant d'états que  $C$  lui est homomorphique.*

#### Preuve 2.3.4.1

Prouvons que l'algorithme de minimisation fournit un automate minimum (il n'en existe aucun comportant moins d'états pour un même langage). Soient un ADF  $A$  et  $C$  l'automate obtenu par l'algorithme de minimisation. Posons que  $C$  comporte  $k$  états.

Par l'absurde, supposons qu'il existe  $M$  un ADF minimisé équivalent à  $A$  mais comptant moins d'états que  $C$ . Posons qu'il en comporte  $l < k$ . Appliquons le table filling algorithm sur  $C$  et  $M$ , comme s'ils étaient un seul ADF, comme proposé dans la section 2.3.3. Les états initiaux sont équivalents puisque  $L(C) = L(M)$ . Dès lors, les successeurs pour chaque symbole sont eux aussi équivalents. Le cas contraire impliquerait que les états initiaux sont différenciables, ce qui n'est pas le cas. De plus, ni  $C$  ni  $M$  n'ont un état inaccessible, sinon il pourrait être éliminé, résultant en un automate comportant moins d'états pour un même langage.

Soit  $p$  un état de  $C$ . Soit un mot  $a_1a_2 \dots a_i$ , qui mène de l'état initial de  $C$  à  $p$ . Alors, il existe un état  $q$  de  $M$  équivalent à  $p$ . Puisque les états initiaux sont équivalents et que, par induction, les états obtenus par la lecture d'un symbole le sont aussi, l'état  $q$  dans  $M$  obtenu par la lecture du mot  $a_1a_2 \dots a_i$  est équivalent à  $p$ . Ceci signifie que tout état de  $C$  est équivalent à au moins un état de  $M$ . Or,  $k > l$ . Cela signifie qu'il doit exister au moins deux états de  $C$  équivalents à un même état de  $M$  et donc équivalents entre eux. Il y a la contradiction : par construction, les états de  $C$  sont tous différenciables les uns des autres. La supposition de l'existence de  $M$  est fausse. Il n'existe pas d'automate équivalent à  $A$  comportant moins d'états que  $C$ .

□

**Preuve 2.3.4.2**

Prouvons que tout automate minimal pour un langage est  $C$ , à un isomorphisme sur les noms des états près.

Soit  $A$  un ADF pour un langage  $L$ . Soient  $C$  un ADF obtenu par l'algorithme de minimisation et  $M$  un automate minimal comportant autant d'états que  $C$ .

Comme mentionné dans la preuve précédente, il doit y avoir une équivalence 1 à 1 entre chaque état de  $C$  et de  $M$ . (Au minimum 1 et au plus 1). De plus, aucun état de  $M$  ne peut être équivalent à 2 états de  $C$ , selon le même argument.

Dès lors, l'automate minimisé, dit *canonique* est unique à l'exception du renommage des différents états.

□

## 2.4 Algorithme d'Angluin

L'algorithme d'Angluin[1] est un algorithme d'apprentissage actif d'automate. Il prend la forme d'un couple professeur/élève où :

- L'élève applique l'algorithme d'Angluin  $L^*$  en temps que tel pour construire un automate représentant le langage cible.
- Le professeur a accès au langage que l'élève veut apprendre.

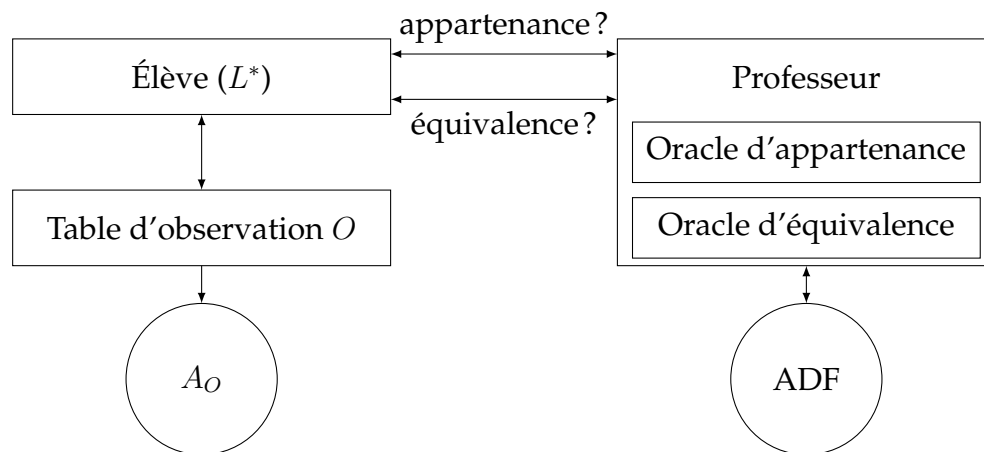


FIGURE 2.14: Vue schématique de l'algorithme d'Angluin

L'algorithme d'Angluin est nommé  $L^*$  et est exécuté par l'élève. Par abus de langage, il peut aussi désigner l'environnement complet dont il a besoin pour être fonctionnel.

Celui-ci consiste, côté élève, d'une table d'observation définie à la section 2.4.3, et d'un automate  $A_O$  construit à partir de cette table.

Côté professeur, il faut un langage régulier  $L$  à apprendre, souvent représenté par un ADF. De plus, ce professeur contient deux oracles :

- L'oracle d'appartenance. Soit un mot  $w$ . Appartient-il à  $L$  ?
- L'oracle d'équivalence. Soit un automate  $A_O$ . Représente-t-il  $L$  ? Si non, fournir un contre-exemple  $w$ .



**Théorème 2.4.1** *S'appuyant sur un professeur pour un langage régulier  $L \subseteq \Sigma^*$ , un élève peut utiliser l'algorithme d'Angluin  $L^*$  pour apprendre l'ADF canonique  $A$  représentant  $L$  en un temps polynomial à  $n$  le nombre d'états de  $A$  et  $m$  le nombre de contre-exemples reçus du professeur. Il effectue  $\mathcal{O}(n)$  requêtes d'équivalence et  $\mathcal{O}(mn^2)$  requêtes d'appartenance.[1]*

### Corollaire 2.4.1.1

Si les requêtes d'appartenance et d'équivalence se font en temps polynomial en la taille de  $A$ ,  $L^*$  est en temps polynomial.

Attention cependant : cet algorithme part du postulat que le langage étudié est régulier.

Les prochaines section introduisent les différentes notions notemment nécessaires à la compréhension du fonctionnement de la table d'observation.

## 2.4.1 La relation $R_L$

Soit un langage  $L$  sur un alphabet  $\Sigma$ .

Soit la relation  $R_L \subseteq \Sigma^* \times \Sigma^*$ . Deux mots  $x$  et  $y$  respectent la relation de Myhill-Nérode  $R_W$  si

$$\forall z \in \Sigma^*, xz \in L \Leftrightarrow yz \in L$$

Intuitivement, deux mots sont en relation si pour tout mot qu'on leur concatène, les deux mots résultants sont tous deux dans le langage  $L$  ou non.

Cette relation est utilisée au fondement de l'algorithme d'Anluin pour séparer le langage en différentes classes, jusqu'à identifier celles qui sont acceptées de celles qui ne le sont pas.

**Lemme 2.4.2** *Cette relation est une relation d'équivalence. De plus, elle est congruente à droite. C'est à dire que si  $xR_Ly$ , alors pour tout symbole  $a \in \Sigma$ ,  $xaR_Lya$*

### Preuve 2.4.2.1 (Equivalence et Congruence à droite)

Dire d'une relation qu'elle décrit une équivalence, revient à dire qu'elle est réflexive, transitive et symétrique

- **Réflexive** : Soit  $x \in \Sigma^*$ . Soit  $z \in \Sigma^*$ . Montrer que  $xR_Lx$  est vrai revient à montrer que  $xz \in L \Leftrightarrow xz \in L$  est vrai.  $R_L$  est donc réflexive.
- **Symétrique** : Soient  $x, y \in \Sigma^*$  tels que  $xR_Ly$ . Soit  $w \in \Sigma^*$ . Montrer que  $yR_Lx$  revient à montrer que  $yw \in L \Leftrightarrow xw \in L$ . Or, par hypothèse,  $xz \in L \Leftrightarrow yz \in L$ , qui peut s'écrire aussi  $yz \in L \Leftrightarrow xz \in L$  pour tout  $z \in \Sigma^*$ , et en particulier  $z = w$ .
- **Transitive** : Soient  $x, y, u \in \Sigma^*$  tels que  $xR_Ly$  et  $yR_Lz$ . Soit  $w \in \Sigma^*$ . Comme  $xz \in L \Leftrightarrow yz \in L$  et  $yz \in L \Leftrightarrow uz \in L$  pour tout  $z \in \Sigma^*$  (par hypothèse), c'est vrai en particulier pour  $z = w$ . Dès lors,  $xw \in L \Leftrightarrow yw \in L$  et  $yw \in L \Leftrightarrow uw \in L$ . Par transitivité de l'implication, on obtient  $xw \in L \Leftrightarrow uw \in L$ , à savoir  $xR_Lu$ .

$R_L$  est congruente à droite. Soient  $x, y \in \Sigma^*$  tels que  $xR_Ly$ . Soit  $a \in \Sigma$ . Par hypothèse,  $xz \in L \Leftrightarrow yz \in L$  pour tout  $z \in \Sigma^*$ . Cela doit donc être vrai en particulier pour le mot  $z = aw$  avec  $w$  quelconque. En remplaçant dans l'hypothèse, on obtient  $xaw \in L \Leftrightarrow yaw \in L$ . Ce qui montre que  $xaR_Lya$ .

□

## 2.4.2 Théorème de Myhill-Nerode

Le théorème de Myhill-Nérode est un résultat fort utilisant la relation  $R_L$ . Il permet de construire un automate à partir de celle-ci.

Cependant, avant d'énoncer le théorème de Myhill-Nérode, il faut s'intéresser à la relation d'équivalence  $R_A$ , qui facilite l'écriture de la preuve.  $R_L$  s'intéresse directement au langage alors que  $R_A$  s'intéresse à un automate qui la représente.

**Definition 1 (Relation  $R_A$ )** Soit un automate  $A = (Q, \Sigma, q_0, \delta, F)$ . Soient deux mots  $x, y \in \Sigma^*$ . Alors la relation  $xR_Ay$  est vraie si et seulement si  $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$ .  $\square$

Intuitivement, deux mots sont en relation  $R_A$  par rapport à un automate  $A$  s'ils mènent à un même état dans celui-ci (ou à des états équivalents).

**Lemme 2.4.3**  $R_A$  est une relation d'équivalence congruente à droite.

### Preuve 2.4.3.1

Prouver qu'une relation est dite d'équivalence, il faut prouver que celle-ci est transitive, réflexive et symétrique. Soit un automate  $A = (Q, \Sigma, q_0, \delta, F)$ .

- $R_A$  est transitive. Soient  $x, y, z \in \Sigma^*$ . Supposons que  $xR_Ay$  et  $yR_Az$ . On a bien  $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y) = \hat{\delta}(q_0, z)$  par la transitivité de l'équivalence entre deux états.
- $R_A$  est réflexive. Soit  $y \in \Sigma^*$ . On a bien  $\hat{\delta}(q_0, y) = \hat{\delta}(q_0, y)$  par réflexivité de l'équivalence sur un état.
- $R_A$  est symétrique. Soient  $x, y \in \Sigma^*$ . Supposons que  $xR_Ay$ . On a bien  $\hat{\delta}(q_0, y) = \hat{\delta}(q_0, x)$  par symétrie de l'équivalence entre deux états.

$R_A$  est congruente à droite. Soient  $x, y \in \Sigma^*$  tels que  $xR_Ay$ . Soit  $z \in \Sigma^*$ . Montrons que  $xzR_Ayz$ .  $\hat{\delta}(q_0, xz) = \hat{\delta}(\hat{\delta}(q_0, x), z) = \hat{\delta}(\hat{\delta}(q_0, y), z) = \hat{\delta}(q_0, yz)$ .  $\square$

**Théorème 2.4.4** Les trois énoncés suivants sont équivalents :

1. Un langage  $L \subseteq \Sigma^*$  est accepté par un ADF.
2. Il existe une congruence à droite sur  $\Sigma^*$  d'index fini telle que  $L$  est l'union de certaines classes d'équivalence.
3. La relation d'équivalence  $R_L$  est d'index fini.

### Preuve 2.4.4.1

La preuve d'équivalence se fait en prouvant chaque implication de façon cyclique :

(1)  $\rightarrow$  (2) Supposons que (1) soit vrai, c'est-à-dire que le langage  $L$  est accepté par un automate déterministe  $A = (Q, \Sigma, q_0, \delta, F)$ . Considérons la relation d'équivalence congruente à droite  $R_A$ . Soit un mot  $w \in \Sigma^*$ . Alors tout mot  $x \in \Sigma^*$  tel que  $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, w)$  appartient à la même classe d'équivalence  $[w]$ . Or, la fonction  $\hat{\delta}$  retourne un état  $q \in Q$ . Chaque classe d'équivalence sur  $\Sigma$  correspond alors à un état de l'automate. Comme  $Q$  est fini,  $R_A$  est d'index fini.

De plus, un sous-ensemble des classes d'équivalences doit correspondre aux états acceptants  $q \in F$ . Alors,  $L$  est l'union de ces classes d'équivalence.

(2)  $\rightarrow$  (3) Supposons qu'il existe une relation  $E$  satisfaisant (2). Montrons que chaque classe de celle-ci est intégralement contenue dans une seule classe de  $R_L$ . Puisque  $E$  est d'index fini, c'est un argument suffisant pour montrer que  $R_L$  est d'index fini. Soit  $x, y$  tels que  $xEy$ . Comme  $E$  est congruente à droite, pour tout mot  $z \in \Sigma^*$ , on sait que  $xzEyz$ . Comme  $L$  est un union de ces classes d'équivalence,  $xzEyz$  implique que  $xz \in L \Leftrightarrow yz \in L$ , ce qui revient à  $xR_L y$ . Cela signifie que tout mot dans la classe d'équivalence de  $x$  définie par  $E$  se retrouve dans la même classe d'équivalence que  $x$  cette fois définie par  $R_L$ . Ceci permet de conclure que chaque classe d'équivalence de  $E$  est contenue dans une classe d'équivalence de  $R_L$  et donc que  $R_L$  est d'index fini.

(3)  $\rightarrow$  (1) Considérons la relation  $R_L$  définie précédemment. Soit un automate  $A = (Q, \Sigma, q_0, \delta, F)$  défini comme suit :

- Chaque état  $q \in Q$  correspond à une classe d'équivalence de  $R_L$ .
- Comme  $R_L$  se définit pour un langage, l'alphabet  $\Sigma$  de celui-ci est déjà défini.
- Si  $[[\epsilon]]$  est la classe d'équivalence de  $\epsilon$  sur  $R_L$ ,  $q_0$  correspond à cette classe.
- Si  $q$  représente  $[[x]]$  et  $q_1$  représente  $[[xa]]$ , alors  $\delta(q, a) = q_1$ . Cette définition est cohérente car  $R_L$  est congruente à droite.
- $F = \{[[x]] \mid x \in L\}$ .

Cet automate est déterministe par la définition de  $\delta$  et fini car  $Q$  l'est, le nombre de classes de  $R_L$  étant fini par hypothèse. De plus, cet automate accepte tout mot  $x \in L$  puisque  $\delta(q_0, x) = [[x]] \in F$  (par définition, puisque  $x \in L$ ).  $\square$

#### Corollaire 2.4.4.1

La partie (3)  $\rightarrow$  (1) de la preuve 2.4.4.1 donne une méthode permettant de construire un ADF à partir des classes d'équivalences de la relation  $R_L$ .

On peut prouver que l'automate obtenu de cette façon est l'automate minimal de  $L$ . Une preuve est disponible dans l'ouvrage [4] en lien avec le théorème 3.10.

### 2.4.3 La table d'observation

Une *table d'observation* est un tableau défini par  $O = (R, S, T)$  avec  $R \subseteq \Sigma^*$  un ensemble de mots *représentants*,  $S \subseteq \Sigma^*$  un ensemble de mots *séparateurs* et  $T : (R \cup R.\Sigma) \rightarrow \{0, 1\}$  une fonction représentant le contenu de la table.[6] Soit un langage  $L$  qui est en train d'être appris par l'algorithme d'Angluin. Soit un mot  $w \in L$ . Alors,  $w$  appartient à une classe d'équivalence  $[[u]]$  avec  $u \in L$ . Dans ce cas,  $T(w) = 1$ . Au contraire, si  $w \notin L$ , alors  $T(w) = 0$ .

Pour une table d'observation  $O$ , deux mots  $u, v$  peuvent être *équivalents sur  $O$* , c'est-à-dire  $uR_O v$ .  $u$  et  $v$  sont équivalents sur  $O$  si et seulement si  $\forall w \in S, T(uw) = T(vw)$ . Intuitivement,  $uR_O v$  si les lignes correspondant à leur classe d'équivalence ont la même séquence de 0 et de 1.

**Proposition 2.4.5** Soient  $u, v \in \Sigma^*$ , un langage  $L$  et une table d'observation  $O$  associée à ce langage. Si  $uR_L v$ , alors  $uR_O v$ .

**Preuve 2.4.5.1**

Soient un langage  $L$ ,  $u, v \in \Sigma^*$  tels que  $uR_L v$  et une table d'observation  $O$  associée à  $L$ . Comme  $uR_L v$ , alors pour tout mot  $w \in \Sigma^*$ ,  $uw \in L \iff vw \in L$ . C'est donc vrai en particulier pour tout  $w \in S$ . Dès lors,  $\forall w \in S, T(uw) = T(vw)$ .  $\square$

**Corollaire 2.4.5.1**

Le nombre de classes d'équivalence sur  $R_O$  est inférieur ou égal à celui de classes d'équivalence sur  $R_L$ .

Cette table  $O$  représente la compréhension actuelle de l'élève du langage  $L$ . D'itération en itération,  $R_O$  représente de mieux en mieux  $R_L$  jusqu'à ce que l'automate induit de cette table soit jugé équivalent par le professeur. L'automate induit l'est par l'application du corollaire 2.4.4.1.

**2.4.4 Fermeture et cohérence****Fermeture**

La propriété de *fermeture* s'exprime mathématiquement par

$$\forall u \in R, \forall a \in \Sigma, \exists v \in R, uaR_O v$$

Cette propriété peut être vérifiée par cet algorithme, expliqué de façon visuelle sur la table O :

**Algorithme 1** Vérification de la fermeture

**Promet:** si la fermeture est respectée ou non

```

1: pour chaque élément  $w$  de la section  $R$  faire
2:   pour chaque symbole  $a$  dans  $\Sigma$  faire
3:     si  $wa$  est dans  $R$  alors
4:       continuer
5:     sinon
6:        $\{wa \text{ est dans } R.\Sigma \text{ par construction}\}$ 
7:       si La ligne de  $wa$  dans  $T$  est différente de celle de  $w$  alors
8:         retourner Faux
9:       fin si
10:    fin si
11:  fin pour
12: fin pour
13: retourner Vrai

```

**Cohérence**

La propriété de *cohérence* se définit mathématiquement comme

$$\forall u, v \in R, uR_O v \Rightarrow \forall a \in \Sigma, uaR_O va$$

Concrètement, il s'agit de prendre deux mots  $(u, v)$  dans  $R$  ayant la même ligne dans  $T$  et vérifier, pour chaque symbole  $(a)$ , s'ils  $(ua, va)$  ont la même ligne dans  $T$ .

#### Exemple 14

Soit la table d'observation  $O$  de la table 2.4 :

$O$	$\epsilon$	$a$
$\epsilon$	0	0
$a$	0	0
$aa$	0	1

TABLE 2.4: Table d'observation  $O$

Cette table n'est pas cohérente. En effet,  $\epsilon R_O a$  mais en ajoutant le symbole  $a \in \Sigma$ , on obtient  $\neg(aR_O aa)$ . Les lignes ont les mêmes valeurs, mais les lignes obtenues par la concaténation du symbole  $a$  ont des valeurs différentes.

### 2.4.5 L'algorithme

Le pseudocode de l'algorithme d'Angluin est fourni par l'algorithme 2.4.1[6]. Celui-ci repose sur les oracles du professeur et l'algorithme 2.4.2, remplissant les lignes de  $T$  encore vides. Le code est suivi d'un exemple d'exécution.

#### Algorithme 2.4.1 (Algorithme d'Angluin $L^*$ )

**Requis:** Un professeur pour le langage régulier  $L \subseteq \Sigma^*$

**Promet:** Un automate canonique décrivant  $L$

- 1: Initialiser la table d'observation  $O = (R, S, T)$  avec  $R = S = \epsilon$
- 2: *corriger*( $O$ )
- 3: **répéter**
- 4:   **tant que**  $O$  n'est pas fermée ou pas cohérente **faire**
- 5:     **si**  $O$  n'est pas fermée **alors**
- 6:       Choisir  $r \in R$  et  $a \in \Sigma$  tels que  $[[ua]]_O \cap R = \emptyset$
- 7:        $R \leftarrow R \cup ua$
- 8:       *corriger*( $O$ )
- 9:     **fin si**
- 10:    **si**  $O$  n'est pas cohérente **alors**
- 11:      Choisir  $uR_L v \in R, a \in \Sigma$  et  $w \in S$  tels que  $T(uaw) \neq T(vaw)$
- 12:       $S \leftarrow S \cup aw$
- 13:      *corriger*( $O$ )
- 14:    **fin si**
- 15: **fin tant que**

```

16: Construire  $A_O$ 
17: Soumettre  $A_O$  à l'oracle d'équivalence
18: si le professeur retourne un contre-exemple  $u$  alors
19:    $R \leftarrow R \cup \text{Pref}(u)$ 
20:   corriger( $O$ )
21: fin si
22: jusqu'à ce que le professeur réponde "oui" à l'équivalence
23: retourner  $A_O$ 

```

#### Algorithme 2.4.2 (corriger( $O$ ))

**Requis:** une table d'observation  $O$ , un professeur pour le langage régulier  $L \subseteq \Sigma^*$

**Promet:** les entrées de  $O$  sont valide dans  $L$

```

1: pour chaque entrée  $u \in (R \cup R\Sigma)$  pour laquelle  $T(u)$  n'est pas encore définie faire
2:   si  $u \in L$  par le test d'appartenance alors
3:      $T(u) \leftarrow 1$ 
4:   sinon
5:      $T(u) \leftarrow 0$ 
6:   fin si
7: fin pour

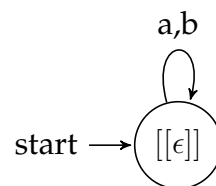
```

Considérons l'automate  $A_3$  de la figure 2.13 construit à la section 2.3.4 sur la minimisation.

#### Première itération

L'algorithme d'Angluin précise, pour son cas de base, une initialisation de la table  $T$  avec les ensembles  $R$  et  $S$  contenant uniquement  $\epsilon$ . Le champ  $R.\{a, b\}$  ( $R.\Sigma$ ) est rempli via des requêtes d'appartenance sur les mots  $a$  et  $b$ .

$O_0$	$\epsilon$
$\epsilon$	0
$a$	0
$b$	0



Automate  $O_0$

L'étape suivante consiste à vérifier la fermeture de la table d'observation  $O_0$ . Pour rappel :

$$\forall u \in R, \forall a \in \Sigma, \exists v \in R, uaR_Ov$$

Intuitivement, pour chaque symbole (ici,  $\{a, b\}$ , et ce sera vrai jusqu'à la dernière itération), tout mot candidat (dans  $R$ , la partie supérieure de la table) doit se retrouver, complété de ce symbole, dans une classe d'équivalence d'un autre candidat de  $R$ . Ici, de toute évidence, les mots  $a$  et  $b$  sont dans la même classe d'équivalence que  $\epsilon$ . Dès lors, la propriété de fermeture est respectée.

Si la fermeture est respectée, alors la question de la cohérence se pose. Pour rappel :

$$\forall u, v \in R, uR_O v \Rightarrow \forall a \in \Sigma, uaR_O va$$

Intuitivement, si deux candidats semblent être dans la même classe d'équivalence (leur lignes dans la table supérieure sont identiques), alors pour n'importe quel symbole, les deux nouveaux mots sont également dans une même classe d'équivalence (leur lignes, potentiellement dans la partie inférieure de la table, sont identiques). N'ayant qu'un seul candidat, cette propriété est forcément respectée ( $R_L$  est réflexive).

Les deux propriétés étant respectées, les classes d'équivalences sont calculées (trivialement ici), et un automate  $O_0$  est proposé à l'enseignant pour vérification.

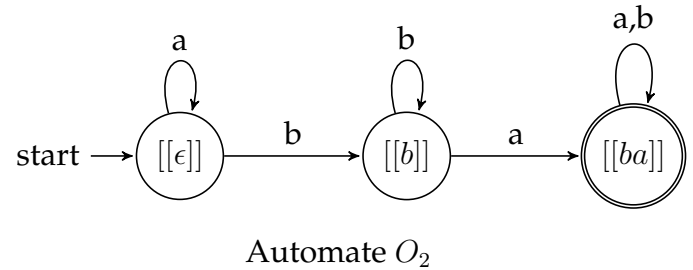
Sur cette itération, un automate initial a été proposé, et aucun état final ne pouvant être atteint avec un seul symbole, la version est minime.

## Seconde itération

L'enseignant répond que non, les automates ne sont pas équivalents. Il fournit le contre-exemple  $ba$ . Comme il est rejeté par  $O_0$ , cela signifie qu'il est accepté par  $A_3$ . Une nouvelle table est alors construite, en ajoutant  $ba$  et ses préfixes (ici, juste  $b$ ) à  $R$ .  $R.\Sigma$  est calculé et les mots n'ayant pas encore reçu une valeur dans  $T$  sont soumis à l'enseignant pour un test d'appartenance. Les valeurs ajoutées ou modifiées dans la table d'observation sont mises en évidence **en rouge**.

$O_1$	$\epsilon$
$\epsilon$	0
$b$	0
$ba$	1
$a$	0
$bb$	0
$baa$	1
$bab$	1

$O_2$	$\epsilon$	$a$
$\epsilon$	0	0
$b$	0	1
$ba$	1	1
$a$	0	0
$bb$	0	1
$baa$	1	1
$bab$	1	1



Comme pour la première itération, la fermeture est testée, suivie de la cohérence. Celle-ci n'est pas respectée : si on considère les mots  $\epsilon$  et  $b$ , qui ont la même ligne dans la table  $T$  ( $\epsilon R_O b$ ), le symbole  $a$ , on obtient les mots  $a$  et  $ba$  qui n'ont pas la même ligne : ( $\neg a R_O ba$ ). Le symbole  $a$  est alors ajouté à  $S$  et une nouvelle table  $O_2$  est calculée.

La fermeture étant déjà vérifiée, la question de la cohérence est reposée, et cette fois-ci elle est vérifiée ; l'automate est construit et proposé à l'enseignant.

Sur cette itération, l'algorithme a reçu le mot  $ba$  comme étant accepté. Il a du ajouter  $a$  à  $S$  pour permettre de différencier certains états. L'automate se voit ajouter les états  $[[b]]$  et  $[[ba]]$ .

### Troisième itération

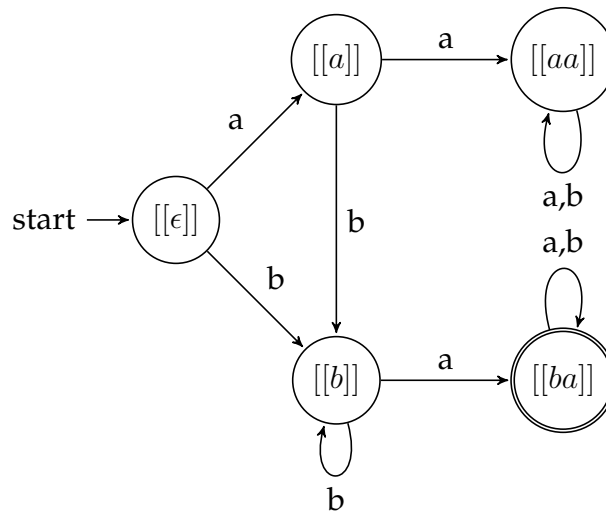
Suivant toujours l'algorithme de comparaison d'automates détaillé dans la section 2.3.3, l'enseignant découvre qu'ils sont différents.

Il sort le contre-exemple  $aaba$ . Si c'est un contre-exemple et qu'il est accepté par  $O_2$ , c'est qu'il ne l'est pas (0) par  $A_4$ . Une nouvelle table  $O_3$  doit être construite.

$O_3$	$\epsilon$	$a$
$\epsilon$	0	0
$a$	0	0
$b$	0	1
$aa$	0	0
$ba$	1	1
$aab$	0	0
$aaba$	0	0
$ab$	0	1
$bb$	0	1
$aaa$	0	0
$baa$	1	1
$bab$	1	1
$aabb$	0	0
$aabaa$	0	0
$aabab$	0	0

$O_4$	$\epsilon$	$a$
$\epsilon$	0	0
$a$	0	0
$b$	0	1
$aa$	0	0
$ab$	0	1
$ba$	1	1
$aab$	0	0
$aaba$	0	0
$bb$	0	1
$aaa$	0	0
$aba$	1	1
$abb$	0	1
$baa$	1	1
$bab$	1	1
$aabb$	0	0
$aabaa$	0	0
$aabab$	0	0

$O_7$	$\epsilon$	$a$	$b$	$ab$	$ba$
$\epsilon$	0	0	0	1	1
$a$	0	0	0	0	1
$b$	0	1	0	1	1
$aa$	0	0	0	0	0
$ab$	0	1	0	1	1
$ba$	1	1	1	1	1
$aab$	0	0	0	0	0
$aaba$	0	0	0	0	0
$bb$	0	1	0	1	1
$aaa$	0	0	0	0	0
$aba$	1	1	1	1	1
$abb$	0	1	0	1	1
$baa$	1	1	1	1	1
$bab$	1	1	1	1	1
$aabb$	0	0	0	0	0
$aabaa$	0	0	0	0	0
$aabab$	0	0	0	0	0



Automate  $O_7$

Ayant reçu  $aaba$ , ce mot et tous ses préfixes sont ajoutés à la table. L'extension  $R.\Sigma$  est recalculée et la table  $O_3$  est construite.

Un manquement à la fermeture est détecté : le mot  $a$ . En effet, en lui ajoutant le symbole



$b$ , on obtient  $ab$  qui n'est ni dans  $R$  ni en relation  $R_O$  avec  $a$ .  $ab$  est alors ajouté à  $R$ , et  $R.\Sigma$  est étendu. La nouvelle table,  $O_4$  est de nouveau testée.

$O_4$  ne respecte pas la cohérence. Les mots  $\epsilon$  et  $aa$  respectent  $R_O$  (leur ligne a la même valeur dans la table) mais  $\neg bR_Oaab$ .  $b$  est alors ajouté à  $S$  et une nouvelle colonne associée est ajoutée à la table, donnant le table  $O_5$ . Celle-ci a toujours un soucis de cohérence entre  $\epsilon$  et  $aa$ , menant à l'ajout de  $ab$  à  $S$  et à la création de  $O_6$ . Finalement, pour régler le soucis de cohérence dans  $O_6$  entre  $a$  et  $aa$ , le mot  $ba$  est ajouté à  $S$  et une table  $O_7$  est ainsi créée avec la nouvelle colonne associée.

Cette table  $O_7$  respectant la fermeture et la cohérence, l'automate associé  $O_7$  est construit et soumis à l'enseignant pour être comparé à  $A_3$ . Celui-ci valide l'égalité et l'algorithme s'arrête : l'automate a été construit.

# Chapitre 3

## Notions spécifiques à LeVer

La section 3.1 définit plus clairement la problématique à l'aide des éléments du chapitre précédent. Elle explique comment les différentes sections de ce chapitre confluent vers une solution à la problématique en question.

Ensuite, la section 3.2 étend le concept des automates aux automates à files et définit une nouvelle opération. Une forme de langage sur ces nouveaux automates est proposée à la section 3.3.

Finalement, la notion de sécurité est définie dans la section 3.4 avec une formule permettant de calculer les états concernés.

Tous ces éléments combinés permettent l'utilisation de LeVer, la technique proposée par [7] et d'en implémenter l'algorithme dans le chapitre 5.

### 3.1 Problème

Cette section décrit le problème rencontré par [7] et la technique générale utilisée pour proposer une solution à ce problème.

Les automates à files définis à la section 3.2 sont plus puissants que les ADF définis dans le chapitre 2. Contrairement à ceux-ci, les automates à files ont potentiellement une infinité d'états. Dans ces conditions, il n'est pas possible d'en faire une exploration exhaustive pour trouver tous les états acceptants.

À la place, une propriété dite de sécurité est définie. Si un état respecte cette propriété, il est *sécurisé*. Si il y a moyen de prouver que la totalité des états de l'automate respectent cette propriété, l'automate est considéré comme sécurisé. Si au contraire, un exemple de violation de la propriété est trouvé, l'automate peut être déclaré comme insécure.

Les sections suivantes donnent les différents éléments utilisés par [7] pour répondre à cette question. Un nouveau langage est donné pour représenter les différents états d'un automate à files. Celui-ci est construit pour pouvoir être régulier pour certains automates. De la sorte, il est possible d'appliquer l'algorithme d'Angluin pour apprendre ce nouveau langage.

Celui-ci n'étant qu'une approximation du langage de l'automate à files, certaines incertitudes peuvent apparaître. Cependant, l'article justifie ces différents cas en ramenant la question à la sécurité, qui peut être répondue en un temps polynomial.

Dès lors, les différentes section permettent de construire ce langage, de se prononcer sur l'appartenance et l'équivalence avec ce langage et d'arrêter l'apprentissage s'il est possible de se prononcer sur la propriété de sécurité pour l'automate à files considéré.

## 3.2 Automate à files

L'article [7] se concentre sur un automate plus général : l'automate à files. Celui-ci est Turing Complete. De la sorte, l'équipe propose une réponse pour un ensemble plus large de langage. Cette section décrit les automates à files.

### 3.2.1 Définitions

**Definition 2** Un *automate à files*  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$  est défini comme suit :

- $Q$  est un ensemble fini d'états de contrôle
- $C$  est un ensemble fini de noms de canaux
- $\Sigma$  est un alphabet
- $q_0 \in Q$  est l'état de contrôle initial
- $\Theta$  est un ensemble fini de noms de transitions
- $\delta$  est la fonction nommante.  $\delta : \Theta \rightarrow Q \times ((C \times \{?, !\} \times \Sigma) \cup \{\tau\}) \times Q$ . Un nom de transition  $\theta$  correspond à une transition de la forme  $\delta(\theta) = (p, \text{"action"}, q)$ . Cette action a une des trois formes suivantes :
  - $c!m$  : C'est une action d'envoi. Le symbole  $m$  est ajouté en fin de canal  $c$ .
  - $c?m$  : C'est une action de réception. Le symbole  $m$  est consommé en début de canal  $c$ .
  - $\tau$  : C'est une action interne. Aucun canal n'est modifié. □

Un automate  $F$  définit un système de transitions  $\mathcal{T} = (S, \Theta, \rightarrow)$ .  $\mathcal{T}$  est l'objet qui permet de passer d'un état à un autre.

En effet, il existe les états de contrôles  $q \in Q$ , mais les états au sens d'un automate à files sont de forme  $s \in S = Q \times (\Sigma^*)^C$ . En particulier,  $s = (q, w)$  avec  $q \in Q$  un état de contrôle et  $w \in (\Sigma^*)^C$  est un vecteur qui fait correspondre à chaque canal  $c \in C$  un mot  $w[c] \in \Sigma^*$  représentant le contenu de ce canal.

Un état  $s$  est composé d'un état de contrôle et du contenu des différents canaux.

De plus, la fonction de transition  $\rightarrow : S \times \Theta \rightarrow S$  associe un état  $s$  et un nom de transition  $\theta$  à un état  $s'$ .

$\mathcal{T}$  respecte trois règles, correspondants chacune à un des types d'actions mentionnés précédemment. En plus de la notation  $w[c]$ , celles-ci utilisent la notation  $w[c \mapsto c']$  signifiant  $w$  à l'exception du canal  $c$  dont le contenu a été remplacé par le mot  $c'$ .

- Si  $\delta(\theta) = (p, c?m, q)$  alors  $(p, w) \xrightarrow{\theta} (q, w')$  si et seulement si  $w = w'[c \mapsto mw'[c]]$
- Si  $\delta(\theta) = (p, c!m, q)$  alors  $(p, w) \xrightarrow{\theta} (q, w')$  si et seulement si  $w' = w[c \mapsto mw[c]]$
- Si  $\delta(\theta) = (p, \tau, q)$  alors  $(p, w) \xrightarrow{\theta} (q, w')$  si et seulement si  $w = w'$

#### Exemple 15

Soit un automate à files  $F$  tel que son système de transitions corresponde à la figure 3.1. Chaque état de l'automate correspondant à un couple état de contrôle/mot, il est imprécis de référer au

système de transitions comme étant l'automate. Cependant, par abus de langage, ceux-ci seront souvent confondus dans ce document.

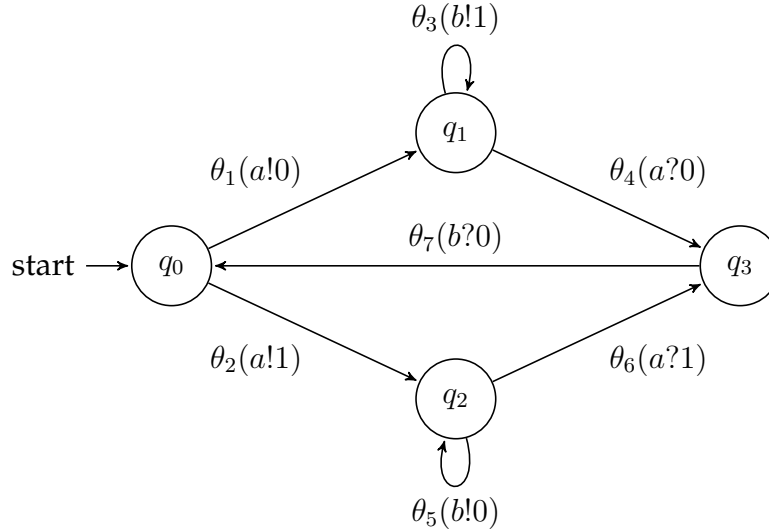


FIGURE 3.1: Système de transitions de l'automate à files  $F$

On retrouve bien la définition d'un automate à files  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$  avec :

- $Q = \{q_0, q_1, q_2, q_3\}$
- $C = \{a, b\}$
- $\Sigma = \{0, 1\}$
- $q_0 \in Q$
- $\Theta = \{\theta_1, \theta_2, \theta_3, \theta_4, \theta_5, \theta_6\}$
- $\delta$  associant à chaque  $\theta_i$  un triplet état/action/état. Celui-ci est représenté entre parenthèses à côté du nom de transition associé

De plus, on peut déduire le système de transition  $\mathcal{T}$  défini par  $F$ . Considérons le mot  $w = [\epsilon, \epsilon]$  où le premier élément du vecteur est le contenu du canal  $a$  et le second celui du canal  $b$ . Dans cet exemple, comme  $\delta(\theta_1) = (q_0, a!0, q_1)$ , alors  $(q_0, w) \xrightarrow{\theta_1} (q_1, w')$ . Dans ce cas,  $w' = [0, \epsilon]$ . A ce moment, on a bien  $w' = w[a \mapsto 0w[a]]$ . En utilisant ce nouveau mot  $w'$ , un nouvel état est atteignable :  $q_3$ . En effet, comme  $\delta(\theta_4) = (q_1, a?0, q_3)$ , alors  $(q_1, w') \xrightarrow{\theta_4} (q_3, w'')$ . Dans ce cas,  $w'' = [\epsilon, \epsilon]$ . A ce moment, on a bien  $w' = w''[a \mapsto 0w''[a]]$ .

Intuitivement, la première transition  $\theta_1$  ajoute le symbole 0 en tête du canal  $a$  en passant de l'état  $q_0$  à l'état  $q_1$ . La transition  $\theta_4$ , elle, permet de passer de l'état  $q_1$  à  $q_3$  en consommant 0 en tête du canal  $a$ .

### 3.2.2 Produit cartésien

Par soucis de simplicité, un automate à files (et son système de transitions servant à le représenter) peut être représenté comme plusieurs systèmes de transitions utilisant les mêmes canaux. Le *produit cartésien* entre deux automates à files  $A$  et  $B$  retourne un nouvel automate  $F = A \times B$ . Dès lors, il est possible de représenter un automate à files en se concentrant sur

ses parties et en les isolant [2]. Cette propriété fait de l'automate à file un choix pertinent pour représenter des automates différents communiquant par des canaux. Dans une application pratique, cela pourrait être des programmes communiquant sur un réseau.

Ce produit cartésien fonctionne comme suit.

Soient les automates à files  $A = (Q_A, C, \Sigma, q_{0A}, \Theta_A, \delta_A)$  et  $B = (Q_B, C, \Sigma, q_{0B}, \Theta_B, \delta_B)$ . Alors le système de transitions  $\mathcal{T} = (S, \Theta, \rightarrow)$  de l'automate à files  $F = A \times B$  est composé de :

- $S \subseteq (Q_A \times Q_B) \times (\Sigma^*)^C$  composé d'un couple d'états de contrôle de  $Q_A$  et  $Q_B$  et du contenu des différents canaux.
- $\Theta$  est un ensemble de noms de transitions.
- $\rightarrow$  est construit comme suit. Soit un état  $((q_A, q_B), w) \in S$ . Soit un triplet  $(p, a, q)$  avec  $p, q \in (Q_A \cup Q_B)$  et  $a \in ((C \times \{?, !\} \times \Sigma) \cup \{\tau\})$ .  $((q_A, q_B), w) \xrightarrow{\theta} ((q_A', q_B'), w')$  si et seulement si l'une des trois conditions suivantes est remplie.
  - $\exists \theta_A \in \Theta_A, \delta_A(\theta_A) = (q_A, a, q_{A'})$  et  $(q_A, w) \xrightarrow{\theta_A} (q_{A'}, w')$  dans l'automate  $A$  et  $\exists \theta_B \in \Theta_B, \delta_B(\theta_B) = (q_B, a, q_{B'})$  et  $(q_B, w) \xrightarrow{\theta_B} (q_{B'}, w')$  dans l'automate  $B$
  - $\exists \theta_A \in \Theta_A, \delta_A(\theta_A) = (q_A, a, q_{A'})$  et  $(q_A, w) \xrightarrow{\theta_A} (q_{A'}, w')$  dans l'automate  $A$ ,  $\forall \theta_B \in \Theta_B, \forall q \in Q_B, \delta_B(\theta_B) \neq (q_B, a, q)$  dans l'automate  $B$  et  $q_{B'} = q_B$
  - $\forall \theta_A \in \Theta_A, \forall q \in Q_A, \delta_A(\theta_A) \neq (q_A, a, q)$  dans l'automate  $A$  et  $q_{A'} = q_A$ ,  $\exists \theta_B \in \Theta_B, \delta_B(\theta_B) = (q_B, a, q_{B'})$  et  $(q_B, w) \xrightarrow{\theta_B} (q_{B'}, w')$  dans l'automate  $B$

Le produit cartésien est un nouvel automate représenté par son système de transitions. Celui-ci étant suffisant pour déduire le langage tracé, il n'est pas nécessaire de décrire formellement  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$ .

De plus, ce nouvel automate est différent des deux autres, il n'est alors pas pertinent de prouver une égalité. Il s'agit juste d'un autre mode de représentation.

### Exemple 16

Soient deux automates à files  $A$  et  $B$  tels que représentés par leur systèmes de transitions donnés par la figure 3.2.

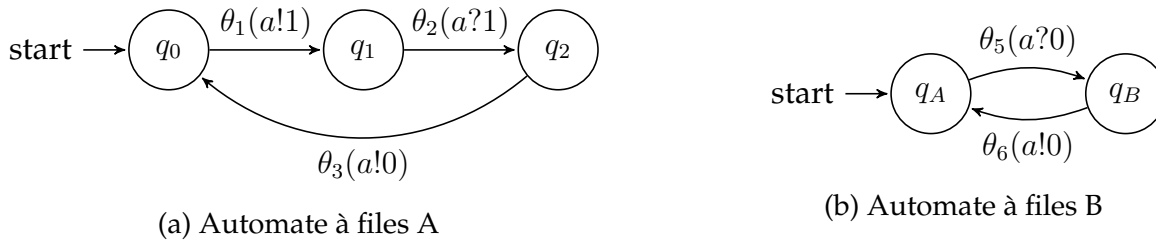
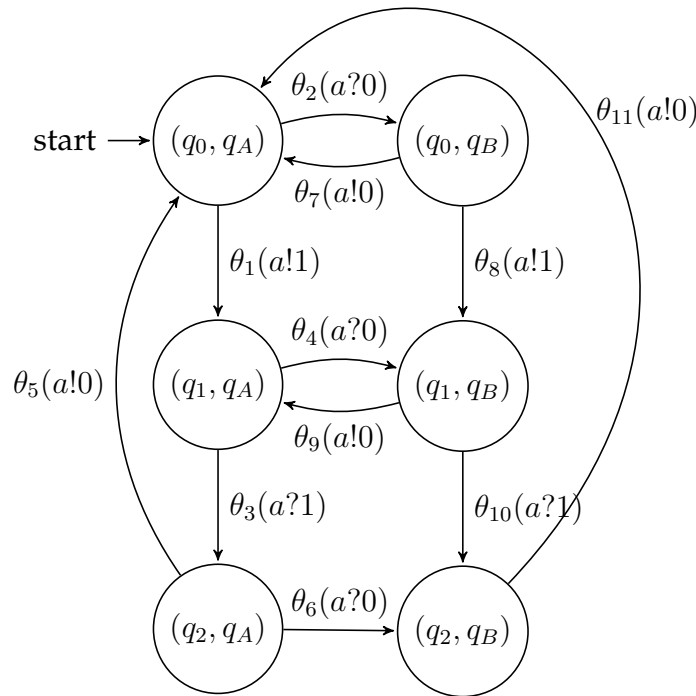


FIGURE 3.2: Automates à files A et B représentés par leur système de transitions

L'automate  $AB = A \times B$  est représenté par son système de transitions à la figure 3.3.

FIGURE 3.3: Automate AB résultant du produit cartésien  $A \times B$ 

L'exemple 3.2.2 suffit à se convaincre qu'on peut parler indistinctement de plusieurs systèmes de transitions comme représentant un seul automate. Dans ce cas, il est sous-entendu que l'automate en question est celui obtenu par produit cartésien.

### 3.3 Trace d'automate

Cette section s'intéresse aux langages qui peuvent être associés à un automate. La section 3.3.1 définit le langage de trace d'un automate. Celui-ci est rarement régulier. Les sections suivantes s'appuient sur [7] pour proposer un langage simplifié, plus souvent régulier, qui représente ce langage de trace. Finalement, la section 3.3.4 définit une fonction  $\mathcal{F}$  qui étend une trace et explique en quoi cela aide à faciliter le processus d'apprentissage.

#### 3.3.1 Langage tracé

Une façon de définir un langage à partir d'un automate à files est de s'intéresser aux noms des transitions suivies lors de l'exécution. Cette section définit les éléments permettant d'arriver à la construction d'un tel langage.

Dans un système de transitions  $\mathcal{T} = (S, \Theta, \rightarrow)$ , la fonction de transition  $\rightarrow: S \times \Theta \rightarrow S$  permet de définir le passage d'un état à un autre.

La *fonction de transition étendue*  $\xrightarrow{*}$  est la fermeture transitive et réflexive de  $\rightarrow$ .

Pour une suite de noms de transitions  $\sigma = \theta_1\theta_2\dots\theta_n \in \Theta^*$ , on note  $(p, w) \xrightarrow{\sigma} (q, w')$  si il existe des états  $(p_1, w_1)(p_2, w_2)\dots(p_{n-1}, w_{n-1})$  tels que  $(p, w) \xrightarrow{\theta_1} (p_1, w_1) \xrightarrow{\theta_2} \dots \xrightarrow{\theta_{n-1}} (p_{n-1}, w_{n-1}) \xrightarrow{\theta_n} (q, w')$ . Dans ce cas,  $\sigma$  est une *trace de chemin*.

**Definition 3** Soit un automate à files  $F$  et l'état initial  $s_0 = (q_0, \epsilon^C)$ . Celui-ci est le couple état de contrôle initial  $q_0$  ainsi que des mots  $w[c] = \epsilon$  pour tout canal  $c \in C$ .

Le *langage de trace* d'un automate  $F$  est

$$L(F) = \{\sigma \in \Theta^* \mid \exists s = (p, w) \text{ tel quel } s_0 \xrightarrow{\sigma} s\}$$

### Exemple 17

Considérons l'automate  $F$  de la figure 3.1.

Pour celui-ci,  $\sigma = \theta_1\theta_4\theta_7$  n'est pas un chemin. En effet,

$$(q_0, [\epsilon, \epsilon]) \xrightarrow{\theta_1} (q_1, [0, \epsilon]) \xrightarrow{\theta_4} (q_3, [\epsilon, \epsilon])$$

Mais, il n'existe pas d'état  $s$  tel que  $(q_3, [\epsilon, \epsilon]) \xrightarrow{\theta_7} s$ . En effet, pour appliquer cette transition, il aurait fallu que le canal  $b$  contienne un symbole 0. Ce n'est pas le cas. Noter que  $\sigma$  est un mot du langage de trace et un chemin dans l'automate à files.

Par contre,  $\sigma = \theta_2\theta_5\theta_5\theta_6\theta_7\theta_1\theta_4\theta_7$  est un chemin dans  $F$  :

$$\begin{aligned} (q_0, [\epsilon, \epsilon]) &\xrightarrow{\theta_2} (q_0, [1, \epsilon]) \xrightarrow{\theta_5} (q_0, [1, 0]) \xrightarrow{\theta_5} (q_0, [1, 00]) \xrightarrow{\theta_6} (q_0, [\epsilon, 00]) \xrightarrow{\theta_7} \\ &(q_0, [\epsilon, 0]) \xrightarrow{\theta_1} (q_0, [0, 0]) \xrightarrow{\theta_4} (q_0, [\epsilon, 0]) \xrightarrow{\theta_7} (q_0, [\epsilon, \epsilon]) \end{aligned}$$

On a bien un état  $s$  (ici  $s = (q_0, [\epsilon, \epsilon]) = s_0$ ) tel que  $s_0 \xrightarrow{\sigma} s$ .

### 3.3.2 Alphabet d'annotation

Le langage de trace en tant que tel n'apporte pas de simplification à l'automate. C'est une autre façon d'écrire un chemin. Pour permettre l'apprentissage par l'algorithme d'Angluin, il faut en construire un nouveau langage, si possible régulier, qui permette de reconstruire le langage de trace. Pour ce faire, ce nouveau langage devrait pouvoir représenter tout état atteignable ainsi qu'un ou plusieurs chemins ou mots témoins permettant d'atteindre ceux-ci.

Pour ce faire, pour chaque nom de transition correspondant à une action d'envoi, un *co-nom* est défini :

$$\bar{\Theta} = \{\bar{\theta} \mid \theta \in \Theta \wedge \exists p, q \in Q, c \in C, a \in \Sigma, \text{ tels que } \delta(\theta) = (p, c!a, q)\}$$

De plus, un *symbole de contrôle* est créé pour chaque état de contrôle :  $T_Q = \{t_q \mid q \in Q\}$ .

En combinant les noms de transitions, les co-noms et les symboles de contrôlé, un nouvel alphabet peut-être défini, l'*alphabet d'annotation* :  $\Phi = (\Theta - \Theta_r) \cup \bar{\Theta} \cup T_Q$ .

$\Theta_r = \{\theta \mid \theta \in \Theta \wedge \exists p, q \in Q, c \in C, a \in \Sigma, \text{ tels que } \delta(\theta) = (p, c?a, q)\}$ , similaire à  $\bar{\Theta}$  mais avec un nom pour chaque transition pour les actions de réception.

### 3.3.3 Trace annotée

Soit  $\mathcal{A} : \Theta^* \rightarrow \Phi^*$  une fonction associant une *trace annotée* à une trace d'automate. Cette fonction est décrite par l'algorithme 2.

---

**Algorithme 2**  $\mathcal{A} : \Theta^* \rightarrow \Phi^*$ 


---

**Requis:** un automate à files  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$ , une suite de noms de transitions  $\sigma \in \Theta^*$

**Promet:** une trace annotée  $\gamma \in \Phi^*$  représentant  $\sigma$

```

1:  $\gamma \leftarrow \epsilon$ 
2: pour chaque nom de transition  $\theta \in \sigma$  faire
3:   si  $\delta(\theta)$  est une action de réception alors
4:     trouver  $\theta_s \in \Theta$  correspondant à une action d'envoi antécédant dans  $\sigma$  telle que les
       actions s'appliquent sur le même canal et le même symbole
5:      $\gamma \leftarrow \gamma$  où  $\theta_s$  est remplacé par  $\bar{\theta}_s \in \bar{\Theta}$  ( $\theta$  n'est pas ajouté à  $\gamma$ )
6:   sinon si  $\delta(\theta)$  est une action d'envoi alors
7:      $\gamma \leftarrow \gamma\theta$ 
8:   fin si
9: fin pour
10: trouver  $q$  l'état de contrôle tel que  $\delta(\theta) = (p, a, q)$  avec  $p \in Q, a \in ((C \times \{?, !\} \times \Sigma) \cup \{\tau\})$ 
11:  $\gamma \leftarrow \gamma t_q$  avec  $t_q \in T_Q$  le symbole de contrôle associé à  $q$ 
12: retourner  $\gamma$ 

```

---

Soit  $AL(F) = \{\mathcal{A}(\sigma) | \sigma \in L(F)\}$  le langage de traces annotées de l'automate  $F$ .  $AL(F)$  est un ensemble de traces annotées correspondant à des exécutions valides de l'automate  $F$ . Intuitivement,  $AL(F)$  contient l'ensemble des états atteignables par  $F$  ainsi que les traces annotées servant de témoins de cette atteignabilité.

Soit un mot  $\gamma \in \Phi^*$ .  $\gamma$  est *correctement formaté* si il fini par un symbole de  $T_Q$  qui qu'aucun autre symbole de cet ensemble n'apparaît dans le mot. Soit un langage arbitraire  $L$ .  $L$  est *correctement formaté* si tous les mots  $w \in L$  le sont.

#### Exemple 18

Soit l'automate  $F$  représenté par la figure 3.2. Soient les traces  $\sigma_1 = \theta_2\theta_8$  et  $\sigma_2 = \theta_1\theta_3\theta_5\theta_2$ . Alors, les traces annotées de ces traces sont :  $\mathcal{A}(\sigma_1) = \theta_2\theta_8 t_{(q_1, q_B)} = \gamma_1$  et  $\mathcal{A}(\sigma_2) = \bar{\theta}_1\bar{\theta}_5 t_{(q_0, q_B)} = \gamma_2$ . Bien qu'elles soient toutes deux correctement formatées,  $\gamma_1$  ne correspond à aucune exécution valide de  $F$ . Dès lors,  $\gamma_1$  n'appartient pas au langage de traces annotées de  $AL(F)$  contrairement à  $\gamma_2$ .

Soit le mot  $\gamma_3 = t_{q_0, q_A}\theta_2 t_{q_0, q_B} \in \Phi^*$ .  $\gamma_3$  n'est pas correctement formatée : il est impossible que cette trace annotée appartienne à  $AL(F)$ .

### 3.3.4 Fonction d'extension de trace

Cette section défini  $\mathcal{F}(L)$  pour un langage arbitraire  $L$  et démontre que  $AL(F)$  en est un point fixe minimum. De la sorte, tout langage qui n'est pas un point fixe minimum de  $\mathcal{F}(L)$  ne peut pas être  $AL(F)$ . Si c'est le cas, la question d'équivalence est répondue : les langages ne sont pas égaux. Il reste alors à générer un contre-exemple.



La fonction d'extension  $Post(L)$  permet d'étendre une trace annotée  $\gamma$  avec le symbole  $\theta$ . Si  $\gamma$  est correctement formée,  $source(\theta)$  et  $cible(\theta)$  donnent respectivement la source et la cible d'une transition  $\delta$ .

$$Post(\gamma, \theta) = \begin{cases} \emptyset & \text{si } \gamma \text{ n'est pas correctement formé ou si } \mathcal{C}(\gamma) \neq source(\theta) \\ \{\mathcal{T}(\gamma)t_{cible(\theta)}\} & \text{sinon si } \delta(\theta) = (p, \tau, q) \text{ ou } \delta(\theta) = (p, c_i!a_j, q) \text{ avec } p, q \in Q \\ \{deriv(\mathcal{T}(\gamma), \theta)t_{cible(\theta)}\} & \text{sinon si } \delta(\theta) = (p, c_i?a_j, q) \text{ avec } p, q \in Q \end{cases}$$

Sachant que  $deriv(\mathcal{T}(\gamma), \theta)$  fonctionne comme l'algorithme  $\mathcal{A}$  si  $\theta$  est une action de réception. Elle le fait en remplaçant un  $\theta_e \in \Theta$  associé à une action d'envoi et le remplace par  $\bar{\theta}_e \in \bar{\Theta}$  si l'action porte sur le même canal et le même symbole que  $\theta$ .

Posons  $Post(\gamma) = \bigcup_{\theta \in \Theta} Post(\gamma, \theta)$  et  $Post(L) = \bigcup_{\gamma \in L} Post(\gamma)$ .

**Théorème 3.3.1** Soit  $\mathcal{F}(L) = Post(L) \cup \{t_{q_0}\}$  où  $q_0$  est l'état de contrôle initial.  $\mathcal{F}(L)$  est une opération monotone sur les ensemble c'est-à-dire qu'elle préserve l'inclusion d'ensembles. De plus,  $AL(F)$  est un point fixe minimal de  $\mathcal{F}(L)$ .

La preuve de ce théorème est disponible en annexe de [7].

Le théorème 3.3.1 donne bien une façon de répondre à la requête d'équivalence dans une direction. Alors, si  $A \oplus B$  donne la différence symétrique entre deux ensembles (l'union à l'exception de leur intersection), le contre-exemple à l'équivalence se trouve dans  $AL(F) \oplus L$ .

Plusieurs cas sont possibles :

1.  $\mathcal{F}(L) - L \neq \emptyset$ . Il existe un mot  $w \in \mathcal{F}(L)$  qui n'appartient pas à  $L$ .
  - Si  $w = t_{q_0}$ , alors il appartient à  $AL(F) \oplus L$
  - Sinon, il faut vérifier si  $w$  est correctement formaté.
    - Si c'est le cas, alors  $w \in (AL(F) \oplus L)$
    - Sinon, c'est qu'il existe  $w' \in L$  tel que  $w \in Post(w')$ .  $Post()$  d'une annotation valide retourne un annotation valide. Par contraposée, si  $w$  est invalide,  $w'$  l'est aussi. Dès lors,  $w' \notin AL(F)$  et donc  $w' \in (AL(F) \oplus L)$ .
2.  $\mathcal{F}(L) \subsetneq L$ . **TODO : expliquer la théorie des prefixpoints là**
3.  $\mathcal{F}(L) = L$ . Soit  $\mathcal{W}(L)$  l'ensemble des traces annotées menant à des états n'étant pas sûrs.  $\mathcal{W}(L)$  est défini et une formule est donnée pour le calculer dans la section 3.4.
  - Si  $\mathcal{W}(L)$  est vide, comme  $L$  est un point fixe (notre hypothèse pour ce point), le processus d'apprentissage peut être arrêté et la sécurité de l'automate est confirmée.
  - Sinon, soit  $\gamma$  une trace annotée dans cet ensemble  $\mathcal{W}(L)$ . Si  $\gamma$  est valide, il peut être retourné comme contre-exemple à la sécurité de  $F$ . Si  $\gamma$  est invalide,  $\gamma \in (AL(F) \oplus L)$

Le langage  $L$  n'est pas forcément  $AL(F)$  dû à cette équivalence limitée (à un sens). Ce pourrait très bien être un autre point fixe contenant  $AL(F)$  ou un autre ensemble contenant une trace annotée menant à un état qui n'est pas sûr. Cependant, ce n'est pas important tant que seule la propriété de sécurité est considérée.

### 3.4 Sécurité d'un automate à files

Comme mentionné au début du chapitre, ce travail s'intéresse à la propriété de sécurité dans les automates à files. Contrairement aux ADFs construits dans le chapitre 2, les automates à files ont un nombre potentiellement infini d'états. Dans ces conditions, il n'est pas possible d'énumérer l'ensemble des états acceptants.

Au lieu de proposer un ensemble d'état acceptants, on va fixer une propriété. Celle-ci permet de catégoriser les états qui sont souhaitables de ceux qui ne le sont pas. Par la suite, la section 3.4.2 propose une technique permettant de calculer l'ensemble des états indésirables à partir d'une trace annotée.

#### 3.4.1 Définitions

Dans un automate à files  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$ , chaque état de contrôle  $q \in Q$  est associé à un union finie de langage réguliers pour chacun des canaux  $c \in C$ .

$$\bigcup_{0 \leq i \leq n_q} \Pi_{0 \leq j \leq k} U_q(i, c_j)$$

Où  $U_q(i, c_j)$  est un langage régulier pour le contenu du canal  $c_j$  sur l'état  $q$ .  $n_q$  est le nombre de langages réguliers utilisés pour définir cette propriété par union.

Un état  $s = (q, [w_0, w_1, \dots, w_k])$  est *sécurisé* s'il n'existe pas  $i, j \in \mathbb{N}$  tels que  $w_j \in U_q(i, c_j)$ .

Si un état n'est pas sécurisé, il est à *risque*.

#### 3.4.2 Traces annotées menant à des états à risques

Soit la fonction  $h_c : \Phi^* \rightarrow \Sigma^*$  qui, pour un trace annotée donnée, ne retourne que les messages envoyés mais non réceptionnés sur le canal  $c$ .

$h_c$  est l'unique homomorphisme qui étend la fonction suivante de  $\Phi$  à  $\Phi^*$  :

$$h_c(\theta) = \begin{cases} m & \text{si } \theta \in \Theta \text{ et } \delta(\theta) = (p, c!m, q) \\ \epsilon & \text{sinon} \end{cases}$$

Si  $L = L(F)$  alors  $L_q$  est l'ensemble des mots correctement formatés pour  $L$ .

Si il existe un état à risque  $s$ , alors il existe une trace  $\sigma \in \Theta^*$  telle que  $s_0 \xrightarrow{\sigma} s$  où  $s_0$  est l'état initial. Si les transitions dénotant des actions d'envoi et de réception d'un même symbole sur un même canal sont enlevées par paires, il ne reste que les transitions participant au contenu final des différents canaux de  $s$ . Par définition de  $h_c$ , pour chaque contenu  $w[c_j]$  de chaque canal  $c_j$ ,  $w[c_j] = h_{c_j}(\mathcal{A}(\sigma))$ . Dès lors, pour que  $s$  soit atteignable, il faut qu'il existe une trace annotée  $\gamma \in AL(F)$  telle que  $s = (q_\gamma, [h_{c0}(\gamma), h_{c1}(\gamma), \dots, h_{ck}(\gamma)])$  où  $q_\gamma$  est l'état de contrôle désigné par le symbole de contrôle à la fin de  $\gamma$ .

Soit la fonction  $h_c^{-1} : \Sigma^* \rightarrow \Phi^*$  l'homomorphisme inverse de  $h_c$ . C'est-à-dire  $h_{c_j}^{-1}(U_q(i, c_j))$  retourne des traces annotées correspondant au contenu d'un canal. Dans ce cas particulier, un des langages réguliers servant à définir la propriété de sécurité. Comme un plusieurs traces

annotées peuvent correspondre au même contenu de canaux par  $h_c$ , un contenu de canal peut correspondre à plusieurs traces annotées via  $h_c^{-1}$ .

En calculant cette fonction pour l'ensemble des états, canaux et langages réguliers définissant la sécurité de  $F$  et en s'assurant que ces traces sont correctement formatées, on obtient un ensemble de traces menant à des états à risque-.

Cet ensemble est décrit mathématiquement par  $\mathcal{W}(L)$  :

$$\mathcal{W}(L) = \bigcup_{q \in Q} \left( \bigcup_{0 \leq i \leq n_q} \left( \bigcap_{0 \leq j \leq k} h_{c_j}^{-1}(U_q(i, c_j)) \right) \right)$$

# Chapitre 4

## LeVer

Dans les chapitres précédants, les bases théoriques sur les automates et langages ont été posées. Celles-ci ont été suivies de concepts plus étroitement liés à LeVer tels que les automates à files, les traces, traces annotées et langages associés. Un dernier élément présenté dans la section 3.4 est la sécurité d'un état ou d'un automate.

En appliquant l'algorithme d'Angluin [1] selon la méthode LeVer [7], il est possible de se prononcer sur la sécurité pour toute une classe d'automates à files : ceux pour lesquels le langage de traces annotées est régulier.

L'objectif dès lors n'est pas d'apprendre le langage de l'automate à file mais le langage de trace associé, et d'adapter la méthode pour répondre à la question de sécurité. En formulant les bonnes propriétés, il est également possible d'interrompre l'algorithme d'apprentissage avant terme si l'on peut se prononcer sur la sécurité de façon certaine.

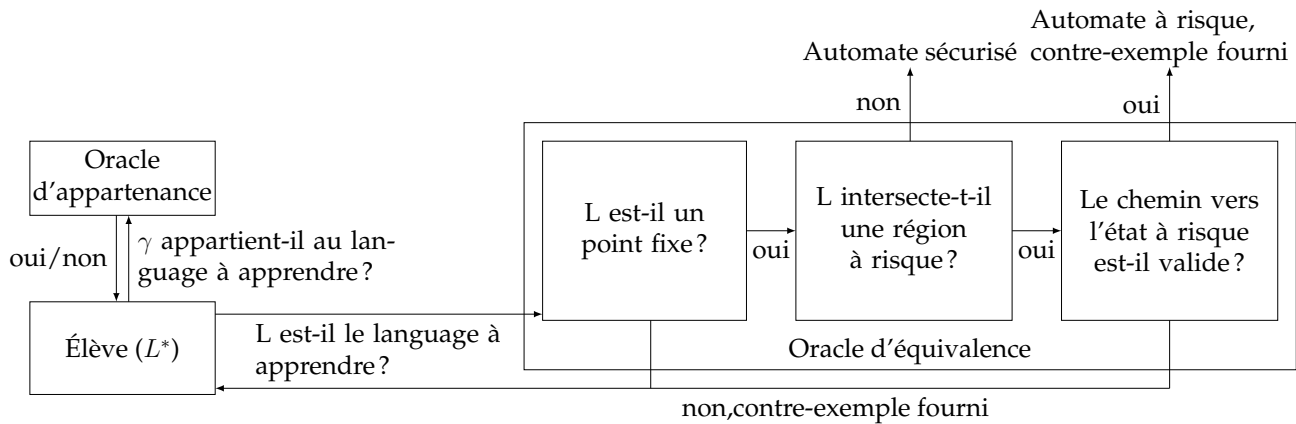


FIGURE 4.1: Vue schématique de l'algorithme d'Angluin pour LeVer[7]

Contrairement à la version proposée dans la section 2.4, ici il n'est pas possible de tester directement l'équivalence entre  $L$  et le langage à apprendre. En effet, ce dernier peut ne pas être régulier et de façon générale le professeur ne connaît justement pas l'automate permettant une telle comparaison si celle-ci existe.

Ici, le professeur passe par trois questions mais à aucun moment il sait dire si  $L$  est exacte-

ment le langage à apprendre. Il peut soit se prononcer sur la sécurité, soit dire avec certitude qu'il existe une meilleure approximation.

- *L est-il un point fixe?* Cette question fait référence à l'application de la fonction d'extension de trace de la section 3.3.4.
- *L intersecte-t-il une région à risque?* Si  $L$  est un point fixe de  $\mathcal{F}$ , il est soit le langage à apprendre soit un point fixe le contenant. Si une surapproximation n'a pas d'intersection avec une région à risque,  $L$  n'en a pas non plus.
- *Le chemin vers l'état à risque est-il valide?* Si un *chemin à risque* (menant à un état à risque) existe dans  $L$ , est-il valide? C'est-à-dire, appartient-il au langage visé et représente-t-il bien un chemin dans l'automate à files étudié?

Les prochaines sections décrivent ce nouvel oracle d'appartenance et celui d'équivalence avec les trois questions posées avant de revenir sur une vue d'ensemble. De la sorte, la section 4.3 résume quand et pourquoi cette méthode fonctionne.

## 4.1 Appartenance

Lorsqu'un mot  $\gamma$  est fourni à l'oracle d'appartenance la question est de savoir s'il appartient à  $AL(F)$ , le langage de trace annotée. Comme l'oracle ne possède pas d'automate pour représenter ce langage, il doit répondre en se basant sur  $F$ .

Ainsi, un mot  $\gamma$  appartient à  $AL(F)$  s'il représente au moins un chemin  $\sigma$  valide dans  $F$ .

Soit une fonction  $\mathcal{A}^{-1}(\gamma)$  donnant l'ensemble des chemins  $\sigma$  pour lesquels  $\mathcal{A}(\sigma) = \gamma$ . Si  $\mathcal{A}^{-1}(\gamma) \neq \emptyset$ , c'est que  $\gamma$  correspond bien à un chemin dans  $F$ ; que  $\gamma \in AL(F)$ .

Premièrement, si  $\gamma$  est incorrectement formaté,  $\mathcal{A}^{-1}(\gamma) = \emptyset$  puisque tout mot de  $AL(F)$  est conforme par définition.

En supposant que  $\gamma$  est correctement formatée, on peut remplacer les co-noms (appartenant à  $\bar{\Theta}$ ) par leur nom de transition équivalent (de  $\Theta$ ). De plus, l'état final est omis. De la sorte, on obtient un mot  $\sigma'$  qui pourrait appartenir à  $\mathcal{A}^{-1}$  si les transitions de réceptions correspondant aux états d'envois qui étaient barrés n'étaient pas omises.

Il est possible d'identifier ces transitions de réception. Dans le corps de  $\gamma$ , chaque envoi barré peut être associé à une transition de réception sur le même canal pour le même symbole. Cependant, la position à laquelle insérer de telles transitions dans le mot est inconnue.

Pour ce faire, il est possible de tester toutes les différentes positions (qui est un ensemble fini), et tester celles-ci sur l'automate  $F$ . Dès lors, toute trace  $\sigma$  correctement formatée pour laquelle  $\mathcal{A}(\sigma) = \gamma$  appartient à  $\mathcal{A}^{-1}(\gamma)$ , rendant l'ensemble non-vidé. Une seule trace est suffisante pour garantir que  $\gamma \in AL(F)$ .

### Exemple 19

Considérons l'automate 3.2a. Pour rappel, celui-ci comprend trois transitions ( $\delta(\theta_1) = (q_0, a!1, q_1)$ ,  $\delta(\theta_2) = (q_1, a?1, q_2)$ , et  $\delta(\theta_3) = (q_2, a!0, q_0)$ ).

Une trace annotée  $\gamma = \bar{\theta}_1\bar{\theta}_3\theta_1q_1$  appartient bien à  $AL(A)$ . En effet,  $\mathcal{A}^{-1}(\gamma) = \{\theta_1\theta_2\theta_3\theta_1\}$ .

Par contre, la trace  $\gamma = \theta_1\bar{\theta}_1\theta_3q_0$  ne convient pas. Aucun chemin  $\sigma$  ne permet  $\mathcal{A}(\sigma) = \gamma$ . Il est facile de s'en convaincre : le chemin passe deux fois par  $\theta_1$  sans passer par  $\theta_3$  qui est ici inévitable pour retourner à l'état de contrôle  $q_0$ .

Par soucis d'efficacité, cet algorithme peut être effectué en exécutant l'automate par étape au lieu de générer toutes les combinaisons possibles avant de les trier. Cela permet de couper des branches dans l'arbre d'exécution. Dans le pire des cas, aucune branche n'est cependant coupée et aucune économie n'en résulte. Le pseudocode de l'annexe A donne le détail d'une telle optimisation.

## 4.2 Équivalence

Lorsqu'un langage régulier  $L$ , représenté par un automate  $A_O$  tel que  $L(A_O) = L$ , est donné à l'oracle d'équivalence, il doit se prononcer sur  $L = AL(F)$ . Cependant, il ne possède pas d'automate pour  $AL(F)$  qui est justement le langage recherché. Il doit alors se prononcer sur l'égalité en se basant uniquement sur  $F$ .

Comme expliqué dans l'introduction du chapitre, c'est impossible de façon générale. Cependant, en supposant que  $AL(F)$  est régulier, il est possible de contourner le problème pour répondre à la question de sécurité.

Contrairement à la version originale de l'algorithme d'Angluin qui a deux possibilités (équivalence ou contre-exemple), celle-ci en a trois. L'oracle peut répondre soit que  $AL(F)$  est sécurisé, soit qu'il ne l'est pas, soit que  $L$  est différent de  $AL(F)$  avec un contre-exemple.

### $L$ est-il un point fixe de $\mathcal{F}$ ?

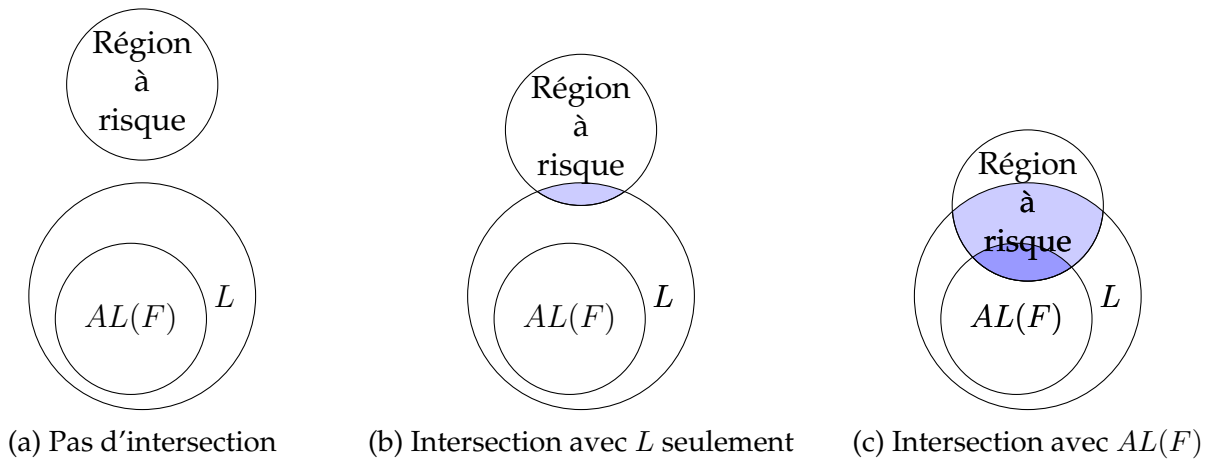
Cette question se base sur la propriété de  $\mathcal{F}$  disant que  $\mathcal{F}(AL(F)) = AL(F)$ ; que c'est un point fixe. Si  $L$  n'est pas un point fixe, il n'a alors aucune chance d'être  $AL(F)$ . Les trois cas de la section 3.3.4 donnent les différents détails.

Ceux-ci mentionnent trois possibilités :

- On peut directement retourner un contre-exemple
- $L = AL(F)$  et la sécurité peut être calculée
- $L$  est une surapproximation de  $AL(F)$ . A ce moment, il faut se poser les deux autres questions pour pouvoir continuer à raffiner la compréhension de la relation entre  $L$  et  $AL(F)$ .

### $L$ intersecte-t-il avec une région à risque?

En visualisant  $L$  comme étant un super-ensemble de  $AL(F)$  et la région à risque comme un ensemble, il est possible de se représenter les différentes situations envisageables.

FIGURE 4.2:  $L$ ,  $AL(F)$  et la région à risque

Pour savoir si nous sommes dans le scénario (a), (b) ou (c) de la figure 4.2, deux tests sont à effectuer.

Premièrement, vérifier si  $\mathcal{W}(L)$  est vide ou non. S'il est vide, nous sommes dans le scénario (a) et il est possible d'annoncer avec certitude que  $F$  est sécurisé.

### Le chemin vers l'état à risque est-il valide ?

Si la réponse à la question précédente est oui, sommes-nous dans le scénario (b) ou (c) ? Considérons un des éléments de  $\mathcal{W}(L)$ . Demandons à l'oracle d'appartenance si ce mot appartient également à  $AL(F)$ .

Si ce n'est pas le cas, c'est que  $L \neq AL(F)$  et que l'algorithme d'Angluin peut continuer grâce au contre-exemple fourni. Ici, il peut s'agir soit d'un scénario (b) soit d'un scénario (c) puisqu'un autre mot de  $\mathcal{W}(L)$  pourrait appartenir à  $AL(F)$ . Améliorer l'approximation d' $AL(F)$  permet justement de mieux discriminer ces deux scénarios sans devoir consulter la totalité de  $\mathcal{W}(L)$ .

Si par contre le mot étudié appartient aussi à  $AL(F)$ , on a un mot étant à la fois dans  $AL(F)$  et dans la région à risque.  $F$  est déclaré à risque et le mot est retourné comme contre-exemple. Il s'agit du scénario (c).

## 4.3 Vue d'ensemble

Ce chapitre a décrit l'adaptation de l'algorithme d'Angluin à l'étude de la sécurité d'automates à file. Grâce à  $\mathcal{W}(L)$  et  $\mathcal{F}(L)$ , des oracles ont pu être construits pour couvrir certains cas. De par les garanties de l'algorithme d'Angluin[1] et la construction utilisée [7], certaines propriétés peuvent être prouvées, énoncées dans le théorème 4.3.1.

**Théorème 4.3.1** *Pour vérifier la propriété de safety des automates FIFO, l'algorithme LeVer respecte les propriétés suivantes :*

1. Si l'algorithme retourne une réponse, celle-ci est correcte

2. *Si  $AL(F)$  est régulier, la procédure s'arrête.*
3. *Le nombre de test d'appartenance et d'équivalence dépend principalement de l'algorithme d'Angluin. Le temps total est borné en temps polynomial du nombre d'états de l'automate minimal pour  $AL(F)$  et linéaire en le temps pris pour une requête d'appartenance à  $AL(F)$*

Ce théorème rappelle la limite de la méthode. Il existe toutes une classe d'automates à files pour lesquels  $AL(F)$  n'est pas régulier. À ce moment, aucune garantie ne peut-être fournie quand à l'exécution.

La preuve est disponible en annexe du travail de Vardhan et al.[7].



# Chapitre 5

## Implémentation

### 5.1 Choix

### 5.2 Résultats

## **Chapitre 6**

## **Conclusion**

# Bibliographie

- [1] D. ANGLUIN, *Learning regular sets from queries and counterexamples*, Information and Computation, 75 (1987), pp. 87 – 106.
- [2] B. BOLLIG, A. FINKEL, AND A. SURESH, *Bounded Reachability Problems Are Decidable in FIFO Machines*, in 31st International Conference on Concurrency Theory (CONCUR 2020), I. Konnov and L. Kovács, eds., vol. 171 of Leibniz International Proceedings in Informatics (LIPIcs), Dagstuhl, Germany, 2020, Schloss Dagstuhl–Leibniz-Zentrum für Informatik, pp. 49 :1–49 :17.
- [3] J. E. HOPCROFT, *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, Addison Wesley, nov 2000.
- [4] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to automata theory, languages and computation. adison-wesley*, Reading, Mass, (1979).
- [5] D. C. KOZEN, *Automata and computability*, Springer Science & Business Media, 1997.
- [6] D. NEIDER, *Applications of automata learning in verification and synthesis*, PhD thesis, Hochschulbibliothek der Rheinisch-Westfälischen Technischen Hochschule Aachen, 2014.
- [7] A. VARDHAN, K. SEN, M. VISWANATHAN, AND G. AGHA, *Actively learning to verify safety for fifo automata*, in International Conference on Foundations of Software Technology and Theoretical Computer Science, Springer, 2004, pp. 494–505.

# Annexe A

## Algorithme d'appartenance

### Algorithme A.0.1 (Appartenance à $AL(F)$ )

**Requis:** une trace annotée  $\gamma \in \Phi^*$  et un automate à files  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$

**Promet:** si  $\gamma \in AL(F)$  ou non

```
1: si  $\gamma$  est mal formatée alors
2:   retourner faux
3: fin si
4: candidats  $\leftarrow \{(q_0, \epsilon^c)\}$ 
5: pour chaque symbole  $\phi \in \gamma$  faire
6:   si  $\phi \in T_Q$  alors
7:     {Fin du mot}
8:   pour chaque  $(q, w) \in$  candidats faire
9:     si  $\phi$  est un  $t_q$  correspondant à  $q$  alors
10:      retourner vrai
11:   fin si
12: fin pour
13: retourner faux
14: sinon
15:   {Exécution de l'automate sur une étape}
16:   nouveau  $\leftarrow \emptyset$ 
17:   extension  $\leftarrow \emptyset$ 
18:   pour chaque  $(q, w) \in$  candidats faire
19:      $\phi' \leftarrow \phi$  sans l'éventuelle barre
20:     {Permet de rejeter les chemins incorrects sans effectuer l'exécution complète}
21:     ajouter  $\delta(\phi', (q, w))$  à extension si  $\delta$  est définie pour ces valeurs
22:   fin pour
23:   {Complète avec les actions de réception possibles}
24:   tant que extension est non vide faire
25:     {Créer les différents cas de consommation des canaux}
26:     prendre un élément  $(q, w) \in$  extension
27:     pour chaque canal non-nul  $c$  de  $w$  commençant par un symbole  $s$  faire
28:       si  $\exists \theta$  correspondant à l'action  $c?s$  sortant de  $q$  alors
```

```
29:      {Comme une extension est elle-même étendue, on a récursivement les cas avec plusieurs
        réceptions bout à bout jusqu'à vider un canal}
30:      ajouter  $\delta(\theta, (q, w))$  à extension
31:      fin si
32:      fin pour
33:      ajouter  $(q, w)$  à nouveaux
34:      retirer  $(q, w)$  d'extension
35:      fin tant que
36:      candidats  $\leftarrow$  nouveaux
37:      fin si
38:      fin pour
```

### Note d'aide à la lecture

Cet algorithme a 3 étapes principales :

- La progression : un nouveau symbole est lu et les états qui le peuvent passent au nouvel état en suivant une transition. Les canaux sont mis à jour.
- L'extension : chaque candidat génère tous les candidats atteignables en consommant ce qu'il reste dans les canaux jusqu'à ce qu'ils soient vides ou qu'aucune transition ne permette de continuer.
- La vérification : une fois que l'algorithme a progressé jusqu'à la fin de  $\gamma$ , s'il reste des candidats, il vérifie si l'état de contrôle final correspond bien au symbole en fin de  $\gamma$ .

Intuitivement, si il y a  $n$  états de contrôle et  $m$  symboles, cet algorithme me semble être dans le pire des cas en  $\mathcal{O}(mn^m)$ . **TODO : une vérification formelle s'il est validé.** Il est linéaire en le nombre d'états explorés. Par contre comme un symbole est consommé à chaque étape, il s'arrête toujours.