

# UMONS



Faculté  
des Sciences

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

**Étudiant :** Benjamin André  
**Directrice :** Véronique Bruyère  
29 mai 2021

# Table des matières

<b>1</b>	<b>Introduction</b>	<b>5</b>
<b>2</b>	<b>Automates et apprentissage</b>	<b>6</b>
2.1	Langage . . . . .	6
2.2	Expressions régulières . . . . .	7
2.3	Automates finis . . . . .	8
2.3.1	Définitions . . . . .	8
2.3.2	Représentation graphique . . . . .	10
2.3.3	Algorithme ECLOSE . . . . .	11
2.3.4	Langage d'un automate . . . . .	12
2.3.5	Équivalence entre expression régulière et automate . . . . .	14
2.3.6	Équivalence entre un ADF et un ANF . . . . .	20
2.4	Algorithme Table Filling . . . . .	23
2.4.1	Relation $R_F$ . . . . .	23
2.4.2	Algorithme . . . . .	24
2.4.3	Appartenance et équivalence . . . . .	27
2.4.4	Minimisation . . . . .	29
2.5	Algorithme d'Angluin . . . . .	31
2.5.1	Fonctionnement . . . . .	31
2.5.2	Relation $R_L$ . . . . .	32
2.5.3	Théorème de Myhill-Nérode . . . . .	33
2.5.4	Table d'observation . . . . .	35
2.5.5	Fermeture . . . . .	35
2.5.6	Cohérence . . . . .	36
2.5.7	Algorithme et complexité . . . . .	36
2.6	Automates à files . . . . .	40
2.6.1	Définition . . . . .	41
2.6.2	Système de transitions . . . . .	42
2.6.3	Langage tracé . . . . .	43
<b>3</b>	<b>Apprentissage d'automates à files</b>	<b>45</b>
3.1	Approche . . . . .	45
3.2	Trace annotée . . . . .	46
3.2.1	Alphabet d'annotation . . . . .	46

3.2.2	Trace annotée . . . . .	47
3.2.3	Fonction d'extension de trace . . . . .	48
3.3	Appartenance . . . . .	48
3.4	Équivalence . . . . .	49
3.4.1	$L$ est-il un point fixe de $\mathcal{F}$ ? . . . . .	49
3.4.2	$L$ intersecte-t-il avec une région à risque? . . . . .	50
3.4.3	Le chemin vers l'état à risque est-il valide? . . . . .	50
3.5	Sûreté . . . . .	51
3.5.1	Définitions . . . . .	51
3.5.2	Traces annotées menant à des états à risques . . . . .	51
<b>4</b>	<b>Implémentation</b>	<b>53</b>
4.1	Langage de programmation et librairies . . . . .	53
4.2	Difficultés . . . . .	54
4.3	Algorithme d'appartenance . . . . .	55
4.4	Algorithme de construction de $\mathcal{F}(L)$ . . . . .	56
4.4.1	Si <i>action</i> est $\tau$ ou de la forme $c!a$ . . . . .	57
4.4.2	Si <i>action</i> est de la forme $c?a$ . . . . .	57
4.4.3	Exemple . . . . .	58
4.5	Algorithme de comparaison entre $\mathcal{F}(L)$ . . . . .	61
4.5.1	Exemple d'application de reverseFL . . . . .	63
4.6	Expérimentations . . . . .	64
4.6.1	Exécution avec arrêt . . . . .	64
4.6.2	Exécution sans arrêt . . . . .	66
<b>5</b>	<b>Conclusion</b>	<b>70</b>
5.1	Rappel de la problématique . . . . .	70
5.2	Résumé des notions rencontrées . . . . .	70
5.3	Apports . . . . .	70
5.4	Suites possibles . . . . .	70
<b>A</b>	<b>Théorème de Knaster-Tarski</b>	<b>72</b>
<b>B</b>	<b>Modifications à Learnlib et Automatalib</b>	<b>73</b>
B.1	Ajouts et modifications à Automatalib . . . . .	73
B.2	Ajouts et modifications à Learnlib . . . . .	74
B.3	Module de démonstration . . . . .	75

## Environnements théoriques

Proposition 2.1.1 $\epsilon$ et la concaténation . . . . .	6
Théorème 2.3.1 ADF et expression régulière . . . . .	14
Théorème 2.3.2 ADF $\implies$ expression régulière . . . . .	15
Preuve 2.3.2.1 . . . . .	15
Théorème 2.3.3 Expression régulière $\implies$ ADF . . . . .	18
Preuve 2.3.3.1 . . . . .	18
Théorème 2.3.4 ANF $\Leftrightarrow$ ADF . . . . .	21
Preuve 2.3.4.1 . . . . .	21
Proposition 2.4.1 $R_F$ . . . . .	23
Preuve 2.4.1. $R_F$ est une relation d'équivalence . . . . .	23
Proposition 2.4.2 Congruence de $R_F$ . . . . .	24
Preuve 2.4.2. Congruence de $R_F$ . . . . .	24
Théorème 2.4.3 Table de différenciation . . . . .	25
Preuve 2.4.3.1 . . . . .	26
Théorème 2.4.4 Minimalité de l'automate réduit . . . . .	30
Preuve 2.4.4.1 . . . . .	30
Preuve 2.4.4.2 . . . . .	31
Théorème 2.5.1 . . . . .	32
Lemme 2.5.2 . . . . .	33
Preuve 2.5.2. Equivalence et Congruence à droite . . . . .	33
Lemme 2.5.3 . . . . .	33
Preuve 2.5.3.1 . . . . .	33
Théorème 2.5.4 . . . . .	34
Preuve 2.5.4.1 . . . . .	34
Proposition 2.5.5 . . . . .	35
Preuve 2.5.5.1 . . . . .	35
Théorème 3.2.1 . . . . .	48
Théorème A.0.1 . . . . .	72
Preuve A.0.1.1 . . . . .	72

## Exemples

Exemple 1	Langages . . . . .	6
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	. . . . .	13
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	. . . . .	41
Exemple 16	. . . . .	43
Exemple 17	. . . . .	44
Exemple 18	. . . . .	48

## Environnements algorithmiques

Algorithme 2.3.1 Appartenance d'un mot à un langage défini par un automate . . . . .	13
Complexité 2.3.1. Lecture d'un mot par un automate . . . . .	14
Complexité 2.3.2.1 . . . . .	16
Complexité 2.3.3.1 . . . . .	19
Complexité 2.3.4. Conversion ANF vers ADF . . . . .	22
Complexité 2.3.4. Conversion ADF vers ANF . . . . .	23
Complexité 2.4.1.1 . . . . .	27
Algorithme 2.4.2 Équivalence entre deux automates . . . . .	28
Complexité 2.4.2.1 . . . . .	29
Algorithme 2.4.3 Minimisation . . . . .	29
Algorithme 2.5.1 Algorithme d'Angluin $L^*$ . . . . .	37
Algorithme 2.5.2 <i>corriger</i> ( $O$ ) . . . . .	37
Algorithme 4.3.1 Appartenance . . . . .	55
Algorithme 4.5.1 Comparaison . . . . .	62
Algorithme 4.5.2 reverseFL . . . . .	63

# Chapitre 1

## Introduction

Onzième version du document.

Le but de ce mémoire dont le titre est une traduction de "Actively learning to verify safety for FIFO automata" [6] est de comprendre et implémenter l'article en question.

Les automates à files sont des objets mathématiques permettant de représenter différents problèmes et sont particulièrement adapté à la représentation de protocoles de communications ou de programmes informatiques communiquant par un réseau.

Une des opérations qui peut leur être appliquée est la vérification de la sûreté : est-il possible que l'automate à file finisse dans une configuration que je juge comme étant indésirable ? Cela peut servir à détecter des bugs dans ces programmes de façon formelle.

Dans le cas d'un modèle fini, une exploration exhaustive est possible. Cependant, les automates à files présentent une infinité de configurations possibles en général. Dès lors, une solution plus élaborée doit être envisagée. C'est ce que présente [6] en appliquant et adaptant un algorithme de machine learning pour répondre à la question de sûreté. Le processus consiste à fournir une représentation de l'automate à apprendre, de supposer que celle-ci est finie, et d'explorer celle-ci pour répondre si l'automate à apprendre est sûr ou non.

Pour tout cela, de nombreuses notions sont nécessaires. Le chapitre 2 introduit celles existant indépendamment de l'article, touchant à la notion d'automate à files et à l'apprentissage d'automates ayant un nombre fini de configurations.

Ensuite, le chapitre 3 décrit les nouvelles techniques et adaptations proposée par [6], une définition de la sûreté ainsi que l'algorithme de machine learning complet permettant de se prononcer sur la sûreté d'un automate à files.

Le chapitre 4 décrit les conditions d'implémentation, les logiciels utilisés et fournit les résultats avant de discuter de leur signification dans le contexte de ce problème.

Finalement, le chapitre 5 reprend des éléments des différents chapitres pour nuancer les propos et proposer des pistes d'amélioration et d'exploration.

Différents algorithmes et théorèmes sont soit référencés soit proposés en annexe.

# Chapitre 2

## Automates et apprentissage

Ce chapitre, comme mentionné dans l'introduction 1, développe les différentes notions théoriques qui servent de blocs de base à l'apprentissage de la sûreté d'un automate à file avec LeVer.

En particulier, la section 2.1 définit les langages, suivies des expressions régulières à la section 2.2, donnant leur nom aux langages réguliers.

Ensuite, avant d'énoncer les automates à files, un modèle plus simple et fini est présenté : les automates finis de la section 2.3. Le Table Filling Algorithme est ensuite détaillé à la section 2.4. Celui-ci permet notamment de tester l'équivalence entre deux automates finis. Cette équivalence est un des éléments utilisés par l'algorithme d'Angluin de la section 2.5, permettant l'apprentissage actif d'automates finis.

Finalement, les automates à files sont introduits dans la section 2.6.

### 2.1 Langage

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.

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  $L^* = \{w_1 w_2 \dots w_n \mid n \in \mathbb{N}, \forall i \in \{1, 2, \dots, n\}, w_i \in L\}$ .

## 2.2 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.

## 2.3 Automates finis

Cette section décrit les automates déterministes finis (ADF) en 2.3.1 et en propose une représentation en 2.3.2. Par après, le langage correspondant est défini en 2.3.4. Les notations utilisées dans cette section s'appuient principalement sur [3], [2] et [4].

En plus des automates déterministes finis, les automates non-déterministes finis avec ou sans transition sur  $\epsilon$  sont également définis. Leur unique utilité dans le cadre de ce mémoire est de permettre de prouver l'équivalence entre un automate déterministe fini et une expression régulière en 2.3.5. Cette dernière section permet dès lors de passer d'une expression régulière à un automate et vice-versa, ce qui est utile lors de l'apprentissage d'un langage.

Pour faciliter la construction du langage et de l'algorithme d'équivalence, une fonction eclose est définie en 2.3.3.

### 2.3.1 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 ??.

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  $q \in Q$ .
- Un 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.

### 2.3.2 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 :

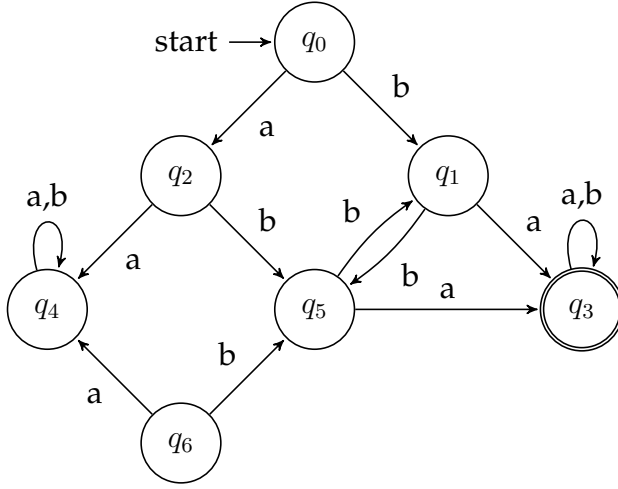
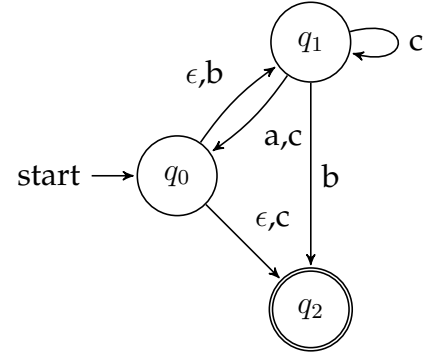


FIGURE 2.1: Graphe de l'ADF de la table 2.1

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.

### 2.3.3 Algorithme ECLOSE

L'algorithme ECLOSE 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 en 2.3.4.

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.

### 2.3.4 Langage d'un automate

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 auto-

mates représentent le même langage, ils sont dit *équivalents*. Voici les définitions ensemblistes, respectivement pour un ADF et pour un  $(\epsilon)$ -ANF

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

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 2.3.1 représente cette appartenance pour un mot.

#### Algorithme 2.3.1 (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:    $C \leftarrow \delta(C, a)$  {passage à l'ensemble des états suivants}
8:   {Si l'automate est un ADF,  $C$  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 2.3.1.1 (Lecture d'un mot par un automate)

La complexité de l'algorithme 2.3.1 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
$\mathcal{O}(1) + n(\mathcal{O}(1) + \mathcal{O}(1)) + \mathcal{O}(\log_2(n))$	$\mathcal{O}(1) + n(\mathcal{O}(1) + \mathcal{O}(n)) + \mathcal{O}(n \log_2(n))$	$\mathcal{O}(n) + n(\mathcal{O}(n) + \mathcal{O}(n)) + \mathcal{O}(n \log_2(n))$
$\mathcal{O}(n)$	$\mathcal{O}(n^2)$	$\mathcal{O}(n^2)$

## 2.3.5 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 2.3.1 (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 2.3.2 (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 2.3.2.1

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  $k$ .

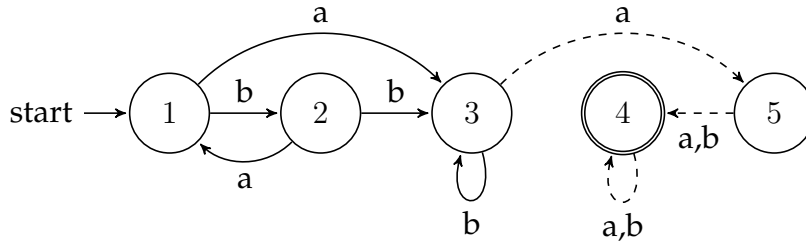


FIGURE 2.3: Exemple : automate mettant  $E_{1,3}^3$  en évidence. Les mots représentés par un chemin passant par un arc discontinu n'appartiennent pas à  $E_{1,3}^3$  : un des états intermédiaires est dénommé par un nombre supérieur à  $k = 3$ .

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.

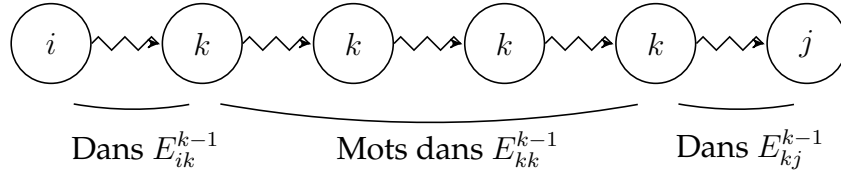


FIGURE 2.4: Un chemin de  $i$  à  $j$  peut être découpé en différent segment en fonction de  $k$

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 2.3.2.1

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 :

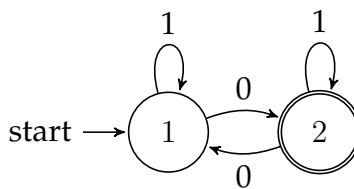


FIGURE 2.5: Un automate acceptant tout mot ayant un nombre impair de 0

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
$E_{11}^1$	$E_{11}^0 + E_{11}^0 (E_{11}^0)^* E_{11}^0$	$(1 + \epsilon) + (1 + \epsilon)(1 + \epsilon)^*(1 + \epsilon)$	$1^*$
$E_{12}^1$	$E_{12}^0 + E_{11}^0 (E_{11}^0)^* E_{12}^0$	$0 + (1 + \epsilon)(1 + \epsilon)^* 0$	$1^* 0$
$E_{21}^1$	$E_{21}^0 + E_{21}^0 (E_{11}^0)^* E_{11}^0$	$0 + 0(1 + \epsilon)^*(1 + \epsilon)$	$0 1^*$
$E_{22}^1$	$E_{22}^0 + E_{21}^0 (E_{11}^0)^* E_{12}^0$	$(1 + \epsilon) + 0(1 + \epsilon)^* 0$	$\epsilon + 1 + 0 1^* 0$

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

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

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,  $E_{12}^2 = 1^* 0(1 + 0 1^* 0)^* = L$ .

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

**Théorème 2.3.3 (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 2.3.3.1

Comme tout  $\epsilon$ -ANF a un ADF équivalent (théorème 2.3.4), 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 ?? . 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.

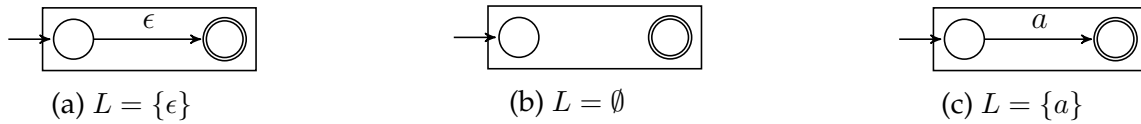


FIGURE 2.6: Blocs de base pour la construction d’un automate à partir d’une expression régulière

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 *abstraits* 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 2.3.3.1

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.

### 2.3.6 Équivalence entre un ADF et un ANF

Il est possible de construire un ADF à partir d'un ANF et réciproquement, tous deux représentant le même langage  $L$ . 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.

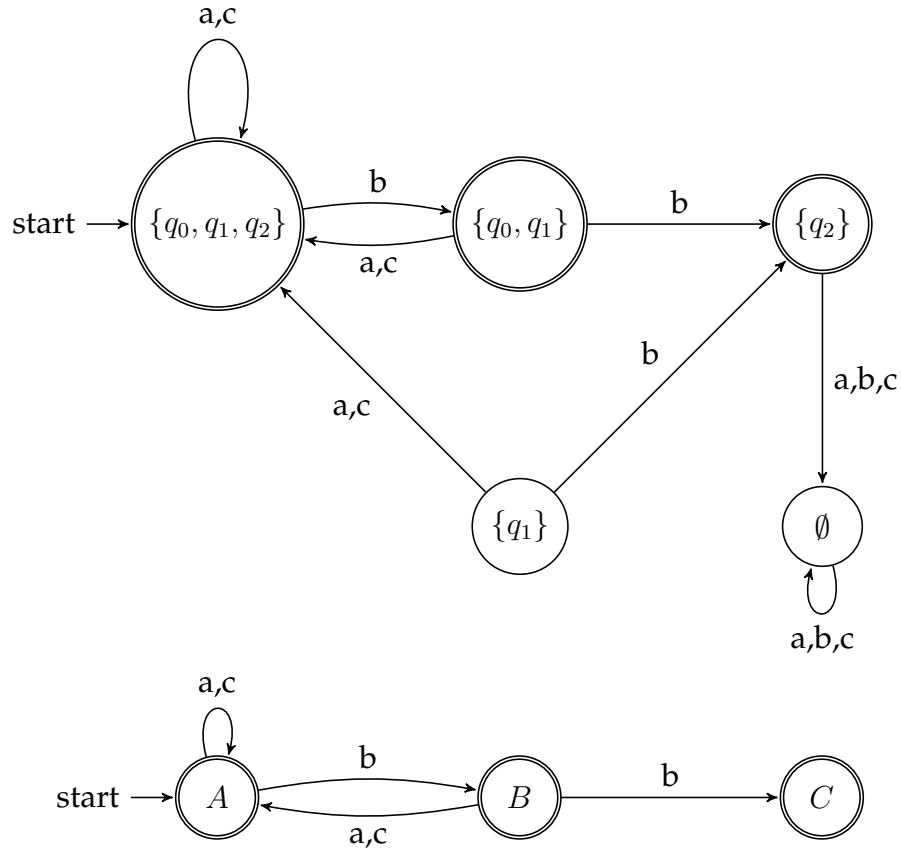


FIGURE 2.8: Automate  $D$ . De par la construction de  $Q$ , le nombre d'états de  $D$  est exponentiel. Les états inatteignables et  $\emptyset$  sont souvent omis pour clarifier la représentation.

**Théorème 2.3.4 (ANF  $\Leftrightarrow$  ADF)** *Un langage  $L$  peut être représenté par un ANF si et seulement si il peut l'être par un ADF.*

#### Preuve 2.3.4.1

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  $\Rightarrow L$  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,  $\hat{\delta}_D(q_D, w) = \delta_D(\{p_1, p_2, \dots, p_k\}, a) = \bigcup_{j=1}^m \text{ECLOSE}(p_j) = \hat{\delta}(q_0, w)$ .

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

Les langages sont donc bien égaux.

□

### Complexité 2.3.4.1 (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 [2], 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 2.3.4.2 (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  $\mathcal{O}(kn)$ .

## 2.4 Algorithme Table Filling

Le Table Filling Algorithm, détaille en 2.4.2 vise à construire un tableau permettant de visualiser une relation entre les différents états d'un ADF. La relation en question,  $R_F$ , est définie en 2.4.1

Une fois l'algorithme défini, il est utilisé dans les sous-section 2.4.3 et 2.4.4 pour tester l'équivalence entre deux automates ou en minimiser un.

### 2.4.1 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 2.4.1 ( $R_F$ )**  $R_F$  est une relation d'équivalence.

**Preuve 2.4.1.1 ( $R_F$  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  $xR_Fy$  et  $yR_Fz$  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  $xR_Fz$ .



- **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 2.4.1.1

$R_F$  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 2.4.1.2

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 2.4.2 (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 2.4.2.1 (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 2.4.2.1

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  $T, \forall q \in S, \delta(q, a) \in T$ .

## 2.4.2 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 2.4.3 (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.

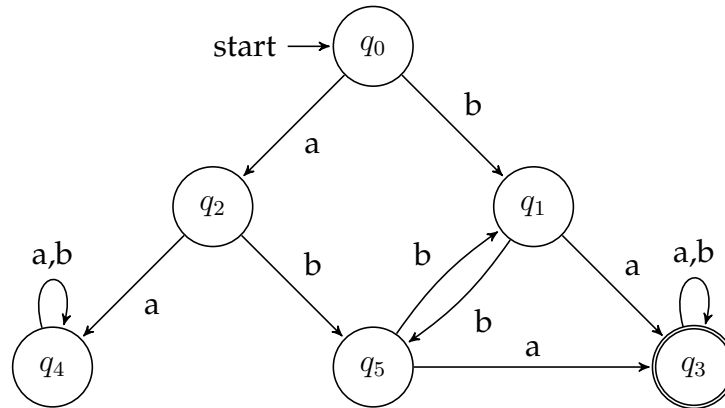


FIGURE 2.9: Automate  $A_2$

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 2.4.3.1

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 2.4.1.1

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  $\mathcal{O}(kn^2)$ .

## 2.4.3 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 2.4.3, 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.

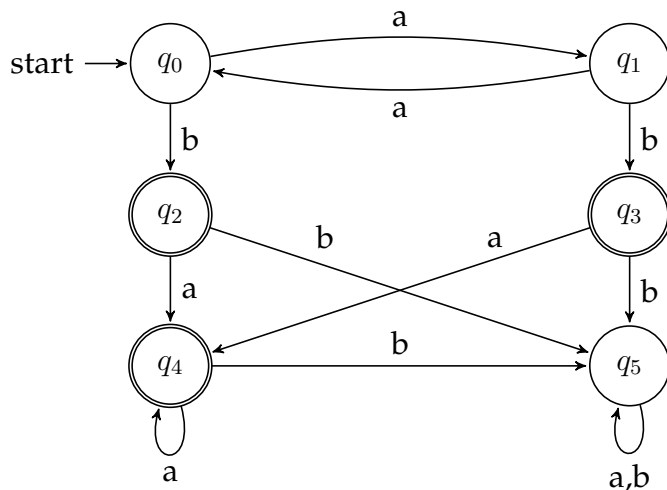


FIGURE 2.10: Automate  $A_H$ , du livre [3] (Fig3.2)

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 2.4.2 (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.

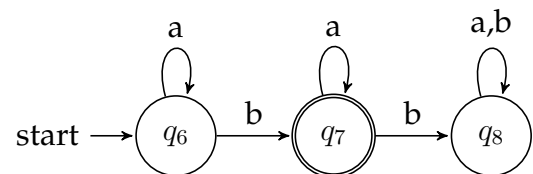


FIGURE 2.11: Automate  $A_I$ , provenant également de [3].

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 2.4.2.1

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 ??.

## 2.4.4 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 ??, construire un nouvel automate.

Décrivons en détail cette minimisation dans l'algorithme 2.4.3.

### Algorithme 2.4.3 (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 2.4.2.1.
  - (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 2.4.2.1. 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.

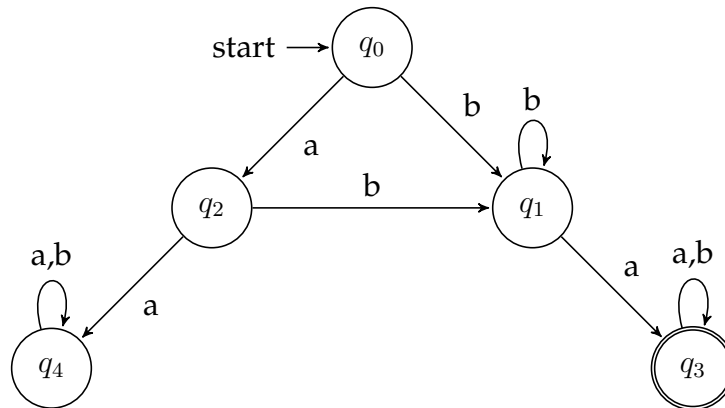


FIGURE 2.13: Automate  $A_3$

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 2.4.4 (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 2.4.4.1

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 ??.

$C$  n'a pas d'état inaccessible par construction et  $M$  n'en a pas par hypothèse. En effet,  $M$  est sensé être minimal. Avoir un état qui peut être éliminé contredirait cette hypothèse.

De plus, les états initiaux sont équivalents puisque  $L(C) = L(M)$ . Dès lors, les successeurs pour chaque symbole sont eux aussi équivalents. Des successeurs différenciables impliquerait que les états initiaux sont différenciables, ce qui est faux par hypothèse.

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  $p$ . Comme  $p$  est quelconque, cela signifie que tout état de  $C$  est équivalent à au moins un état de  $M$ .

Or,  $k > l$ . Il doit exister au moins deux états de  $C$  équivalents à un même état de  $M$  et donc équivalents entre eux. Il y a là une contradiction : par construction, les états de  $C$  sont tous

différenciables les uns des autres. La supposition de l'existence de  $M$  est absurde. Il n'existe pas d'automate équivalent à  $A$  comportant moins d'états que  $C$ . □

#### *Preuve 2.4.4.2*

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. □

## 2.5 Algorithme d'Angluin

L'algorithme d'Angluin permet l'apprentissage actif d'automates déterministes finis définis dans la section 2.3. Ceux-ci représentant un langage régulier, celui-ci est effectivement appris. C'est cet algorithme, adapté à la situation, qui se trouve au cœur de LeVer, objet de ce mémoire.

La section 2.5.1 donne une description générale de l'algorithme et de son exécution. Celui-ci repose sur différents concepts tels que :

- le table filling algorithm de la section 2.4 (permettant notamment de tester l'équivalence entre deux automates et de les minimiser au besoin)
- la relation  $R_L$  de la section 2.5.2 (portant sur des mots d'un langage) elle-même utilisée par le théorème de Myhill-Nerode
- le théorème de Myhill-Nerode de la section 2.5.3
- la table d'observation de la section 2.5.4, utilisée pour stocker la progression de l'algorithme
- la fermeture et la cohérence définies dans les sections 2.5.5 et 2.5.6

Finalement, l'algorithme est donné dans la section 2.5.7 et sa complexité et ses limites y sont discutées.

### 2.5.1 Fonctionnement

Soit un langage régulier  $L$ . L'algorithme d'Angluin[1] est un algorithme d'apprentissage actif d'automate qui permet d'apprendre un automate  $A$  représentant  $L(A) = L$ . 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. Pour cela, il s'aide d'une table d'observation (section 2.5.4)
- Le professeur a accès au langage que l'élève veut apprendre.



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  $w$ .

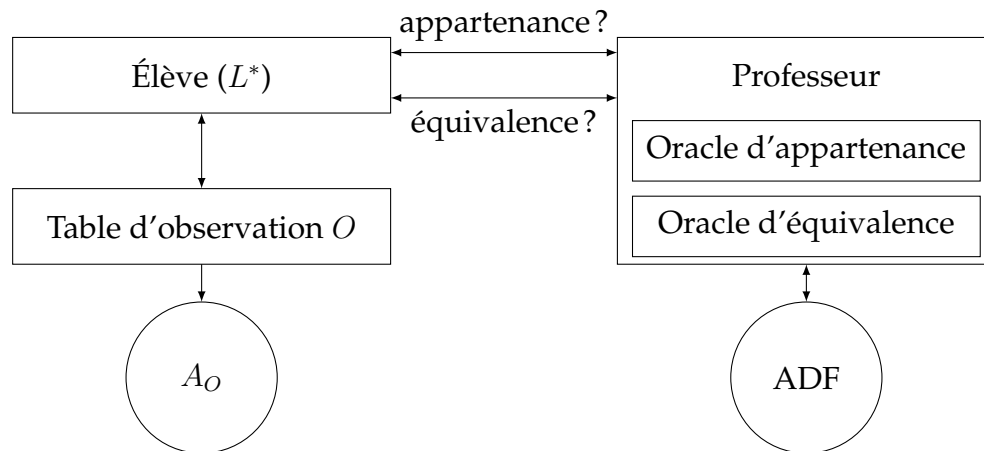


FIGURE 2.14: Vue schématique de l'algorithme d'Angluin

**Théorème 2.5.1** *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 polynimial 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 2.5.1.1

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.

## 2.5.2 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 2.5.2** 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 2.5.2.1 (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  $xaR_Lya$ .

□

### 2.5.3 Théorème de Myhill-Nérode

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  $\hat{\delta}(q_0, x) = \hat{\delta}(q_0, y)$ . □

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

**Preuve 2.5.3.1**

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  $xzR_Ayz$ .  $\hat{\delta}(q_0, xz) = \hat{\delta}(\hat{\delta}(q_0, x), z) = \hat{\delta}(\hat{\delta}(q_0, y), z) = \hat{\delta}(q_0, yz)$ .

□

**Théorème 2.5.4** *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 2.5.4.1

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  $xR_Ly$ . 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  $x \in L$ ). □

**Corollaire 2.5.4.1**

La partie (3)  $\rightarrow$  (1) de la preuve 2.5.4.1 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 [3] en lien avec le théorème 3.10.

**2.5.4 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.[5] 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 2.5.5** 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 2.5.5.1**

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 2.5.5.1**

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 2.5.4.1.

**2.5.5 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

```

**2.5.6 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.

**2.5.7 Algorithme et complexité**

Le pseudocode de l'algorithme d'Angluin est fourni par l'algorithme 2.5.1[5]. Celui-ci repose sur les oracles du professeur et l'algorithme 2.5.2, remplissant les lignes de  $T$  encore vides. Le code est suivi d'un exemple d'exécution.

**Algorithme 2.5.1 (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 2.5.2 (*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

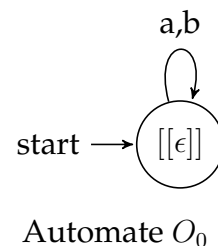
```

Considérons l'automate  $A_3$  de la figure 2.13 construit à la section ?? 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



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 :

$$\forall u, v \in R, uR_Ov \Rightarrow \forall a \in \Sigma, uaR_Ova$$

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.

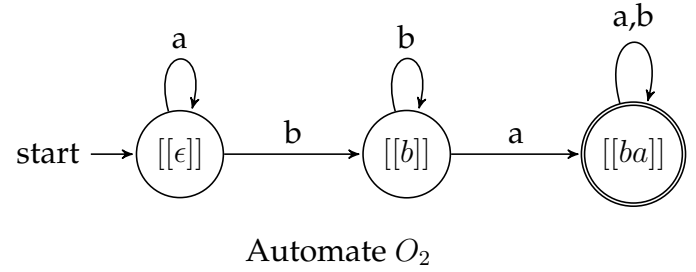
Sur cette itération, un automate initial a été proposé, et aucun état final ne pouvant être atteint avec un seul symbole, la version est minime.

## 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 ??, l'enseignant découvre qu'ils sont différents.

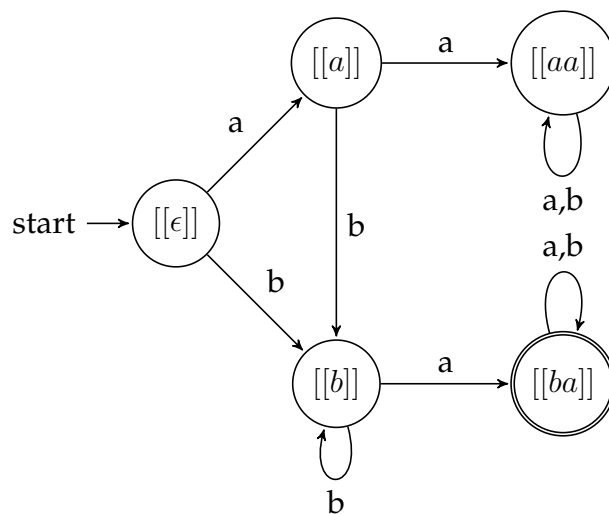
Il sort le contre-exemple  $aaba$ . Si c'est un contre-exemple et qu'il est accepté par  $O_2$ , c'est qu'il ne l'est pas (0) par  $A_4$ . Une nouvelle table  $O_3$  doit être construite.

$O_3$	$\epsilon$	$a$
$\epsilon$	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.

## 2.6 Automates à files

L'article [6] se concentre sur un automate plus général : l'automate à files. Les automates à files sont plus complexes que les ADF. Les langages définis par ceux-ci sont plus complexes que les langages réguliers mais toujours dans l'espace défini par  $\Sigma^*$ . La figure 2.15 propose une intuition des langages définis par les automates à files.

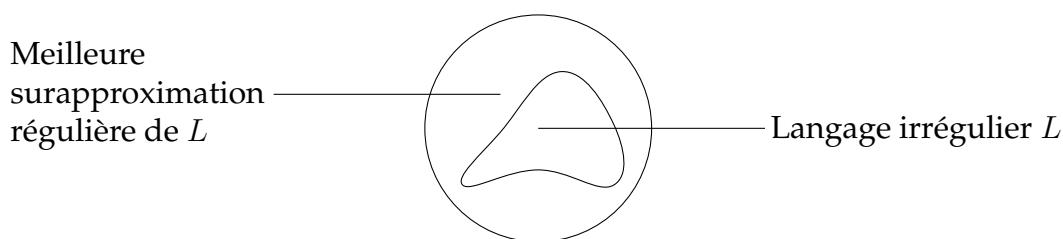


FIGURE 2.15: Relation entre langage régulier et langage irrégulier

Ainsi, de nouveaux langages peuvent être définis de façon plus précise. Leur puissance expressive est équivalente aux machines de Turing. Cela a un coût : pour certains problèmes il n'existe pas de réponse générale ou si celle-ci existe elle prend un temps exponentiel à construire.

C'est le cas de la question de sécurité des automates à files dans ce mémoire. Cependant, le chapitre ?? décrit comment une solution peut-être construite pour tout un ensemble d'automates à files, permettant de se prononcer sur un ensemble plus large que les langages régulier. L'automate à file est dès lors un outil pertinent pour élargir l'ensemble des langages pour lesquels il est possible d'étudier la sécurité.

Cette section se divise en une première partie 2.6.1 donnant une définition formelle des automates à files et un exemple ainsi qu'une seconde partie 2.6.2 sur les systèmes de transitions, représentation potentiellement infinie des différentes configurations d'un automate à files. Finalement, la section ?? donne une définition formelle de ces nouveaux langages représentés par des automates à files.

## 2.6.1 Définition

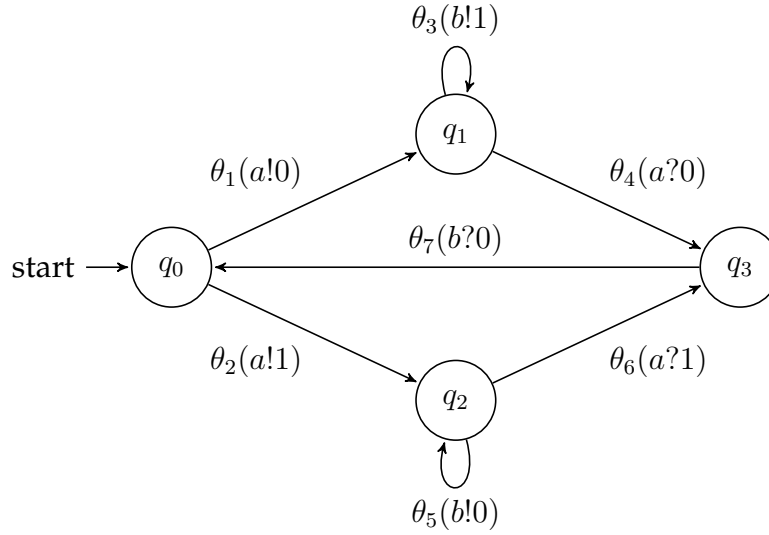
**Définition 2** Un *automate à files*  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$  est défini comme suit :

- $Q$  est un ensemble fini d'états de contrôle
- $C$  est un ensemble fini de 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  $m \in \Sigma$  est ajouté en fin de canal  $c$ .
  - $c?m$  : C'est une action de réception. Le symbole  $m \in \Sigma$  est consommé en début de canal  $c$ .
  - $\tau$  : C'est une action interne. Aucun canal n'est modifié.

□

### Exemple 15

Soit un automate à files  $F$  donné à la figure 2.16.

FIGURE 2.16: 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 telle que l'action est représentée entre parenthèses à côté du nom de transition associé

## 2.6.2 Système de transitions

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. Intuitivement, 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  $s \in S$  et un nom de transition  $\theta \in \Theta$  à un état  $s' \in S$ .

$\mathcal{T}$  respecte trois règles, correspondant chacune à un des types d'actions pouvant être associées par la fonction nommante. 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  $w = w'$

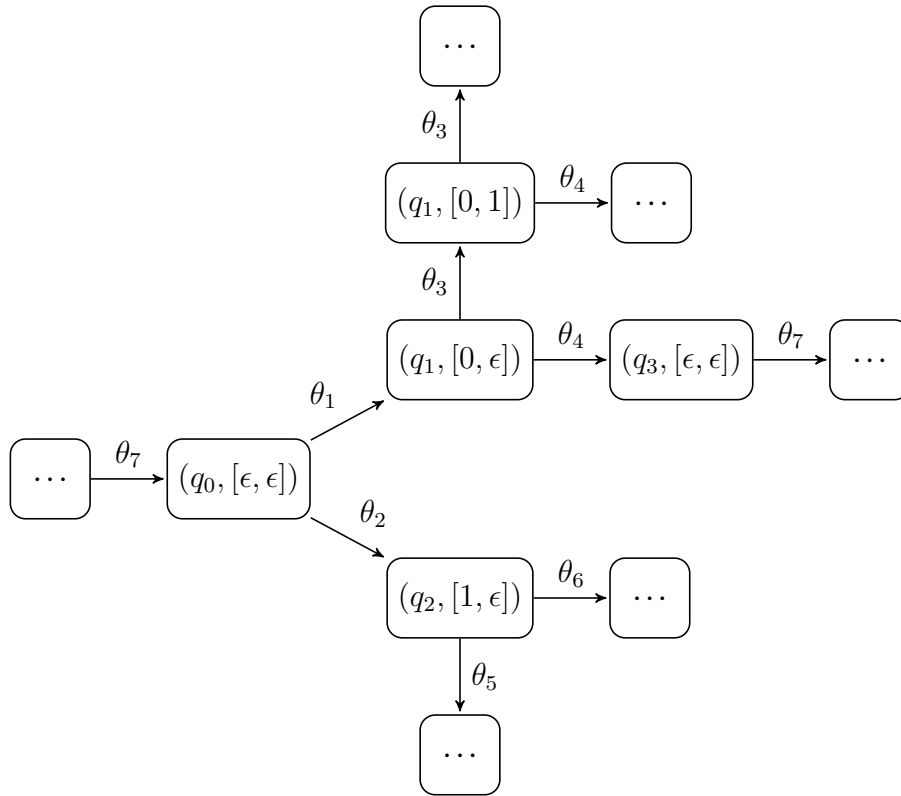
**Exemple 16**

Reprenons l'automate à files  $F$  de la figure 2.16. Un système de transitions  $\mathcal{T}$  peut lui être associé.

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  $a$ .

Comme ce système de transitions  $\mathcal{T}$  est infini et sans état initial, seule une représentation partielle est possible. La figure 2.17 offre cette représentation autour de  $s = (q_0, [\epsilon, \epsilon])$ . Noter qu'il existe une transition entrante vers  $s$  car la configuration  $s' = (q_3, [\epsilon, 0])$  est possible et que la lecture de  $\theta_7$  retourne pour  $s'$  une configuration identique à  $s$ .

FIGURE 2.17: Automate à files  $F$ **2.6.3 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.

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* et  $(p_1, w_1), (p_2, w_2), \dots, (p_{n-1}, w_{n-1})$  est un *chemin*.

De plus, comme  $\sigma$  correspond à un chemin,  $\sigma$  est une *trace valide*.

**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 traces* 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 2.16.

Pour celui-ci,  $\sigma = \theta_1\theta_4\theta_7$  n'est pas une trace. En effet, on a

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

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.

Par contre,  $\sigma = \theta_2\theta_5\theta_5\theta_6\theta_7\theta_1\theta_4\theta_7$  est une trace. En effet, on a :

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

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

# Chapitre 3

## Apprentissage d'automates à files

Ce chapitre présente le problème rencontré par [6] et la technique générale utilisée pour proposer une solution à ce problème dans la section 3.1. Ce problème s'intéresse à la propriété de sûreté des automates à files définis dans le chapitre 2. L'approche définie, un langage particulier est introduit : le langage de traces annotées de la section 3.2. Celui-ci permet de représenter certains systèmes de transitions, infinis, par des langages régulier et donc des ADF. Utilisant une version modifiée de l'algorithme d'Angluin, les oracles d'appartenance et d'équivalence sont ajustés respectivement dans les sections 3.3 et 3.4.

Finalement, la sûreté est définie formellement à la section 3.5. Tous ces éléments mis ensemble permettent de compléter l'approche proposée dans la section 3.1. Les détails algorithmiques sont laissés pour le chapitre 4.

### 3.1 Approche

Les automates à files définis à la section 2.6 sont plus puissants que les ADF définis dans la section 2.3. En effet, les systèmes de transitions associés ont potentiellement une infinité de configurations. Dans ces conditions, il n'est pas possible de faire une exploration exhaustive des états pour trouver lesquels sont acceptants.

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

L'idée dès lors est de travailler non pas avec un système de transitions infini mais avec une représentation construite pour être finie dans un vaste ensemble de problèmes. Cette représentation est le langage de traces annotées défini dans la section 3.2.

En supposant que celui-ci est régulier, il est possible d'utiliser l'algorithme d'Angluin de la section 2.5 pour l'apprendre. Cependant, ce langage n'est qu'un concept et le professeur n'a accès qu'à un automate à files  $F$ . Pour cette raison, les oracles d'appartenance et d'équivalence sont adaptés pour répondre à une requête entre un langage fourni par l'élève et  $F$ .

La figure 3.1 donne une vue schématique de ce nouvel algorithme d'Angluin modifié.

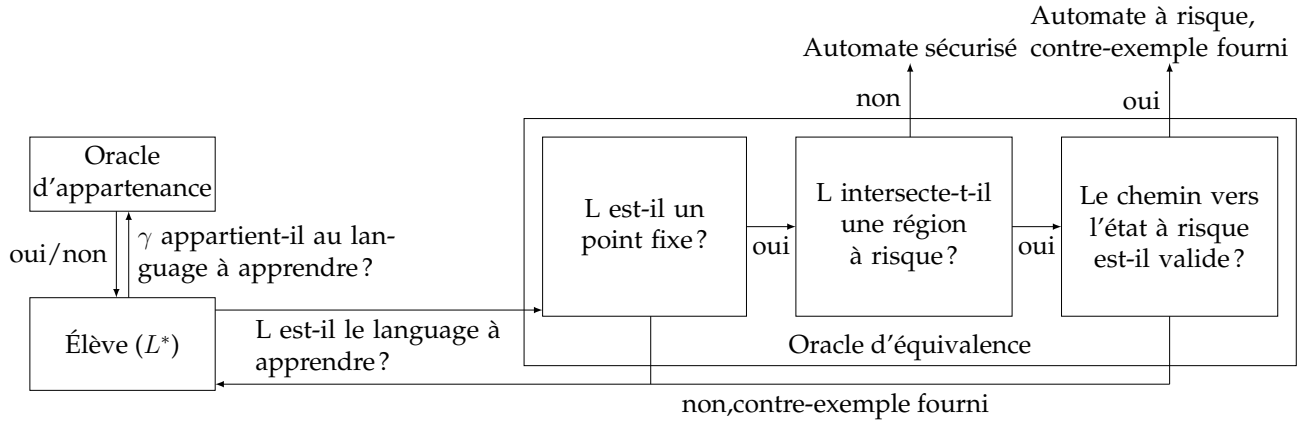


FIGURE 3.1: Vue schématique de l'algorithme d'Angluin pour LeVer[6]

*LeVer* (Learn to Verify) est le nom de cette technique. En particulier, on peut noter sur ce schéma que l'oracle d'équivalence peut non seulement répondre oui ou non mais également interrompre l'apprentissage s'il est possible de se prononcer sur la sûreté d'un automate à files  $F$ . Pour rappel,  $L(F)$  n'est pas régulier en général. Pouvoir se prononcer sur la sûreté de  $F$  étant possible, il n'est ni utile ni forcément possible de continuer à appliquer  $L^*$  pour obtenir une meilleure approximation  $L$  du langage de trace de  $F$ .

## 3.2 Trace annotée

Cette section propose un nouveau langage : les traces annotées. Il se base sur les traces et permet de retrouver celles-ci. Si ce langage de trace annotées est régulier, il peut être appris conformément à [6]. Se basant sur le langage de traces défini à la section 2.6, un nouvel alphabet est proposé en 3.2.1. Grâce à celui-ci, un algorithme est défini en 3.2.2 qui permet de construire une trace annotée à partir d'une trace. Finalement, la sous-section 3.2.3 propose une fonction d'extension de trace et en propose une propriété intéressante qui permet de faciliter la réponse à l'oracle d'équivalence par le professeur de l'algorithme d'Angluin défini dans la section 2.5.

### 3.2.1 Alphabet d'annotation

Le langage de traces 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 :

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

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

En combinant les noms de transitions, les co-noms et les symboles de contrôle, un nouvel alphabet peut être défini, l'*alphabet d'annotation* :  $\Phi = (\Theta - \Theta_r) \cup \bar{\Theta} \cup T_Q$ . Dans cet alphabet, on a  $\Theta_r = \{\theta \mid \theta \in \Theta \wedge \exists p, q \in Q, c \in C, a \in \Sigma, \text{ tels que } \delta(\theta) = (p, c?a, q)\}$ , similaire à  $\bar{\Theta}$  mais avec un nom pour chaque transition pour les actions de réception.

### 3.2.2 Trace annotée

Soit  $\mathcal{A} : L(F) \rightarrow \Phi^*$  une fonction associant une trace annotée  $\gamma \in \Phi^*$  à une trace d'automate à file  $\sigma \in \Theta^*$ . Cette *trace annotée* est un mot sur l'alphabet d'annotation  $\Phi$  tel qu'il représente au moins la trace  $\sigma$ . Il y a une perte d'information lors de cette transformation.  $\mathcal{A}$  est la *fonction d'annotation*.

Intuitivement,  $\mathcal{A}$  compacte une trace en remplaçant tout  $\theta$  correspondant à une action d'envoi par son équivalent  $\bar{\theta}$  si le symbole envoyé est consommé plus loin dans la trace, sur le même canal. Si une trace valide est annotée, il ne peut y avoir d'action de réception orpheline : le symbole a bien dû être envoyé avant d'être consommé. Dès lors, la trace annotée ne comporte que des symboles correspondant à des actions d'envoi, avec une barre ou non. De plus, un symbole appartenant à  $T_Q$  est ajouté à la fin pour signifier l'état de contrôle à atteint en suivant le chemin défini par la trace.

Cette fonction est reprise algorithmiquement en l'algorithme 2.

---

#### Algorithme 2 $\mathcal{A} : L(F) \rightarrow \Phi^*$

---

**Requis:** un automate à files  $F = (Q, C, \Sigma, q_0, \Theta, \delta)$ , une trace  $\sigma \in L(F)$

**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  $s_O \xrightarrow{\sigma} s = (q, w)$  pour un certain  $w \in (\Sigma^*)^c$
  - 11:  $\gamma \leftarrow \gamma t_q$  avec  $t_q \in T_Q$  le symbole de contrôle associé à  $q$
  - 12: **retourner**  $\gamma$
- 

Soit  $AL(F) = \{\mathcal{A}(\sigma) \mid \sigma \in L(F)\}$  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 finit par un symbole de  $T_Q$  et aucune autre occurrence d'un symbole de  $T_Q$  n'apparaît dans le mot. Soit un langage arbitraire  $L$ .  $L$  est *correctement formaté* si tous les mots  $\gamma \in L$  le sont.



**Exemple 18**

Soit l'automate  $F$  représenté par la figure ?? . 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 :  $\mathcal{A}(\sigma_1) = \theta_2\theta_8t_{(q_1,q_B)} = \gamma_1$  et  $\mathcal{A}(\sigma_2) = \bar{\theta}_1\bar{\theta}_5t_{(q_0,q_B)} = \gamma_2$ . 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  $\gamma_3 = t_{q_0,q_A}\theta_2t_{q_0,q_B} \in \Phi^*$ .  $\gamma_3$  n'est pas correctement formatée : il est impossible que cette trace annotée appartienne à  $AL(F)$ .

**3.2.3 Fonction d'extension de trace**

Cette section définit  $\mathcal{F}(L)$  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 formatée,  $source(\theta)$  et  $cible(\theta)$  donnent respectivement la source et la cible d'une transition  $\delta$ .

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

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 3.2.1** 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 ensembles c'est-à-dire qu'elle préserve l'inclusion d'ensembles. De plus,  $AL(F)$  est le plus petit point fixe de  $\mathcal{F}(L)$ .

La preuve de ce théorème est disponible en annexe de [6]. Un algorithme détaillant comment calculer  $\mathcal{F}(L)$  est fourni en annexe ??, accompagné d'un exemple.

**3.3 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  $F$ .  $\gamma \notin AL(F)$  s'il n'est pas correctement formaté.

Simuler l'automate à files  $F$ , symbole par symbole, permet de tester si  $\gamma$  représente bien une exécution valide. La section 4.3 apporte un algorithme efficace pour simuler les différentes exécutions possibles pour une trace annotée.

### 3.4 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.

#### 3.4.1 $L$ est-il un point fixe de $\mathcal{F}$ ?

??

Cette question se base sur le théorème 3.2.1 et en particulier du fait que  $AL(F)$  est un point fixe. Il n'est pas possible de prouver que  $L = AL(F)$  mais il est possible de montrer que  $L \neq AL(F)$  en montrant que  $L$  n'est pas un point fixe.

Une façon de montrer que  $L \neq AL(F)$ , est d'énoncer un seul élément dans  $L \cup AL(F) - L \cap AL(F) = AL(F) \oplus L$  l'union exclusive des deux ensembles. Pour rappel,  $AL(F)$  est un langage contenant l'ensemble des traces valides pour l'automate à files  $F$ .

En comparant  $\mathcal{F}(L)$  avec  $L$  pour vérifier si  $L$  est un point fixe de  $\mathcal{F}$ , plusieurs situations peuvent apparaître :

- $\mathcal{F}(L) - L \neq \emptyset$ . Considérons un mot  $w \in \mathcal{F}(L) - L$  et montrons que  $w \in AL(F) \oplus L$ .
  - Si  $w = t_{q_0}$ , c'est que  $t_{q_0} \notin L$ . Pourtant, par définition,  $t_{q_0} \in AL(F)$ . Dès lors,  $w \in AL(F) \oplus L$ .
  - Sinon, si  $w$  est une annotation valide, c'est que  $L$  n'est pas un point fixe de  $\mathcal{F}$ . Dès lors,  $w$  suffit à démontrer que  $AL(F) \neq L$ .
  - Sinon,  $w$  n'est pas une annotation valide.  $w$  faisant partie de  $\mathcal{F}(L)$ , cela signifie qu'il doit exister un mot  $w' \in L$  tel que  $w = \text{Post}(w')$ . Ce  $w'$  ne peut pas être une annotation valide : cela impliquerait que  $w$  l'est également ce qui est posé comme faux ici. Comme  $w'$  n'est pas une annotation valide pour  $F$ ,  $w' \notin AL(F)$ . Comme  $w' \in L$ , naturellement  $w' \in AL(F) \oplus L$ .
- $\mathcal{F}(L) \subsetneq L$ . Dans ce cas, utilisons la notion de point préfixe. Un ensemble  $Z$  est un *point préfixe* d'une fonction  $\mathcal{F}$  s'il réduit par son application :  $\mathcal{F}(\mathcal{F}(Z)) \subseteq Z$ . Selon cette définition,  $L$  est un point préfixe de  $\mathcal{F}$ . En appliquant  $\mathcal{F}$  des deux côtés, ce qui préserve l'inclusion car  $\mathcal{F}$  est monotone, on obtient  $\mathcal{F}(\mathcal{F}(L)) \subsetneq \mathcal{F}(L)$ .  
Donc,  $\mathcal{F}(L)$  est également un point préfixe. Soit  $w \in L - \mathcal{F}(L)$ . Comme  $w$  n'est pas dans l'intersection de ces deux point préfixes, il ne fait pas partie du point fixe minimum  $AL(F)$ . En effet, selon la théorie des points fixes de Knaster-Tarski (annexe A), un point fixe minimal est également l'intersection de tous les points préfixes de  $\mathcal{F}$ .
- $\mathcal{F}(L) = L$ . Cela signifie que  $L$  est un point fixe de  $\mathcal{F}$ . Cela signifie qu'il n'est pas possible de prouver que  $L \neq AL(F)$  et infirmant cette propriété. Cela ne signifie par pour autant

que  $L = AL(F)$ .

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. Pour tenter de continuer à chercher un contre-exemple ou tester la sûreté, il faut poser d'autres questions.

La transcription en algorithme de la comparaison entre  $L$  et  $\mathcal{F}(L)$  pour trouver un contre-exemple à  $L = AL(F)$  est donnée dans la section 4.5, accompagnée d'un exemple.

### 3.4.2 $L$ 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.

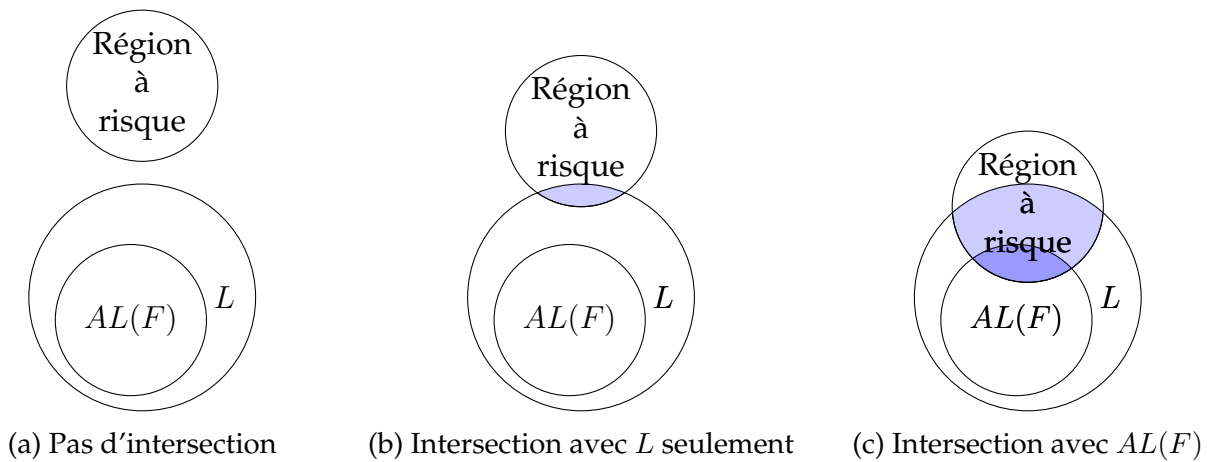


FIGURE 3.2:  $L$ ,  $AL(F)$  et la région à risque

Pour savoir si nous sommes dans le scénario (a), (b) ou (c) de la figure 3.2, 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.

### 3.4.3 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  $AL(F)$ . Cela correspondrait à prendre un mot de la zone bleue claire dans (c).

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).

### 3.5 Sûreté

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 la section 2.3, 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 3.5.2 propose une technique permettant de calculer l'ensemble des états indésirables à partir d'une trace annotée.

#### 3.5.1 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 tous les états d'un automate sont sécurisé, l'automate est également *sécurisé*.

Si un état n'est pas sécurisé, il est à *risque*. S'il existe au moins un état à risque dans un automate, celui-ci est à *risque*.

#### 3.5.2 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

## Implémentation

Ce chapitre décrit les choix qui ont été faits pour implémenter l'algorithme développé dans le chapitre 3. La section 4.1 présente les choix envisagés de langages de programmation et bibliothèques avant de nommer le candidat retenu de façon argumentée. Ensuite, la section 4.2 décrit les difficultés rencontrées ainsi que les solutions et compromis apportés à chacune d'elles. Trois d'entre elles étaient de l'ordre algorithmique. Les algorithmes proposés sont expliqués dans les sections 4.3, 4.4 et 4.5. Chacun est accompagné d'un exemple. Finalement, quelques expérimentations sont citées et appréciées dans la section 4.6.

Un résumé des changements effectués dans les bibliothèques utilisées est disponible dans l'annexe B.

### 4.1 Langage de programmation et bibliothèques

Implémenter ces différents types d'automates, les opérations associées, ainsi que les différents algorithmes demande une quantité considérable de travail.

Utiliser des bibliothèques préexistantes permet de réutiliser du code et de limiter le travail restant, plus spécifique à ce document.

Une bibliothèque en python, automata-lib ainsi que les bibliothèques java Automatalib et Learnlib ont été considérées.

automata-lib supporte tous les types d'automates nécessaires, avec des méthodes de bases telles que l'exécution pour un mot, la validité pour les automates sans files ainsi qu'une méthode de construction simple. Cependant, il manque beaucoup de fonctionnalités. Les opérations booléennes entre automates ne sont pas implémentées. La détermination d'un NFA ou l'algorithme d'Angluin ne sont pas proposés non plus.

Pour ces raisons, le couple Automatalib-Learnlib a été préféré. Ces bibliothèques sont plus complexes et ne supportent pas les automates à files. Cependant, toutes les autres opérations citées précédemment sont présentes. Cela permet de limiter le travail à l'implémentation des automates à files et des méthodes et algorithmes couverts dans les chapitres 3 et 4.

## 4.2 Difficultés

Ce travail a rencontrés plusieurs difficultés. En voici la liste avec les contre-mesures proposées si c'est pertinent.

- Les librairies utilisées sont complexes. Automatalib et Learnlib sont conçues de façon très générique. Cependant, ajouter le concept de file au automates n'était pas prévu. La solution apportée est de développer un pan dans chaque librairie dédié plus spécifiquement aux automates à files. Éventuellement, des méthodes ont été dupliquées dans ce contexte.
- La documentation était insuffisante ou introuvable. La documentation des librairies sur leur site respectif est succinte et sommaire. En général, le code n'est commenté que pour les classes n'ayant pas elles-même de super-classe. De plus, il manque des explications sur les relations entre les classes. Pour travailler dans ces conditions, il a fallu explorer une grande partie du projet, tester certaines fonctionnalités. Pour simplifier la lecture du code ajouté, des commentaires plus fréquent y ont été ajoutés. L'annexe B sert aussi de référence aux relations entre classes.
- Certaines opérations ne sont proposées que pour les ANF alors que l'algorithme d'Angluin est défini pour les ADFs. Une fonctionnalité a été ajoutée pour la traduction triviale d'un ANF à un ADF équivalent.
- Les librairies utilisent des dépendances complexes. Chaque librairie est découpée en de nombreux sous-modules interconnectés. Les cycles de dépendances sont interdits entre autres pour l'interprétation des annotations et la génération de code. La structure en sous-modules n'a pas été modifiée mais une attention particulière a été apportée aux dépendances et aux fichiers de configuration.
- L'article de Vardhan et al.[6] manque de précision sur l'implémentation de la méthode *LeVer*. Entre autre en comparant des ensembles potentiellement infinis (les langages) sans expliquer comment c'est fait en pratique avec les automates. Des algorithmes ont dû être reconstruits pour permettre l'implémentation de ces opérations sur des ordinateurs. Parmi ceux-ci :
  - L'algorithme d'appartenance de la section 4.3.
  - L'algorithme permettant de calcul de  $\mathcal{F}(L)$  pour un langage de trace annotées  $L$  donné, à la section 4.4
  - L'algorithme permettant de comparer  $\mathcal{F}(L)$  et  $L$  pour trouver un contre-exemple prouvant que  $L$  n'est pas un point-fixe de  $\mathcal{F}(L)$ , à la section 4.5.

L'article ne détaille pas non plus d'algorithme permettant de calculer l'ensemble des configuration à risque  $\mathcal{W}(L)$ . Cet ensemble est plus facile à calculer si les langages réguliers décrivant les configurations à risques ne s'intéressent pas au contenu des canaux mais uniquement aux états traversés dans l'automate à file. C'est cette approche qui est utilisée dans ce travail.

Une solution idéale à toutes ces difficultés demanderait une réécriture profonde des librairies ainsi qu'un grand travail de documentation. Cela améliorerait grandement la qualité du code mais est au-delà des limites de ce travail.

### 4.3 Algorithme d'appartenance

L'oracle d'appartenance est l'un des deux oracles utilisés par l'algorithme d'Angluin découvert dans la section 2.5. La section 3.3 mentionne le comportement de cet oracle. Cependant, elle manque de précisions permettant d'exécuter l'automate à files de façon efficace.

L'algorithme d'appartenance 4.3.1 répond à cette problématique.

#### Algorithme 4.3.1 (Appartenance)

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

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

```

1:  $a, \gamma' \leftarrow$  premier symbole de  $\gamma$  et le reste de la trace annotée
2: si  $a \in T_Q$  alors
3:   si  $a$  correspond à  $q_a$  et  $|\gamma'| = 0$  alors
4:     retourner vrai
5:   fin si
6: sinon si  $a \in \bar{\Theta}$  alors
7:    $a' \leftarrow a$  sans la barre
8:    $(p, c!m, q) \leftarrow \delta(a')$  {comme  $a \in \bar{\Theta}$ , il s'agit forcément d'une action d'envoi}
9:   si  $p = q_a$  alors
10:    pour chaque  $\theta_r$  tel que  $\delta(\theta_r) = (r, c?m, s)c$  faire
11:      pour chaque position  $i$  dans  $\gamma'$  sauf après le symbole terminal  $t \in T_Q$  faire
12:         $\gamma'' \leftarrow \gamma'$  où  $\theta_r$  a été inséré à la position  $i$ 
13:        si  $Appartenance(\gamma'', F, q)$  alors
14:          retourner vrai
15:        fin si
16:      fin pour
17:    fin pour
18:  fin si
19: sinon
20:    $(p, action, q) \leftarrow \delta(a)$  {action est obligatoirement de type envoi sinon la trace ne serait pas correctement formatée}
21:   si  $p = q_a$  alors
22:     retourner  $Appartenance(\gamma', F, q)$ 
23:   fin si
24: fin si
25: retourner faux

```

**Exemple d'application de l'algorithme d'appartenance** Soit l'automate à files  $F$  dans la figure 2.16. Soit le mot correctement formaté  $\gamma = \bar{\theta}_2 \bar{\theta}_5 t_{q_0} \in \Phi^*$ .

Voici le déroulement de  $appartenance(\gamma, F, q_0)$ .

$q_a = q_0$ ,  $a = \bar{\theta}_2$  et  $\gamma' = \bar{\theta}_5 t_{q_0}$ .  $a \in \bar{\Theta}$  et  $source(a) = q_0$ . Les dérivés avec  $\theta_r$  sont produits. Ici le seul  $\theta_r$  compatible est  $\theta_6$

—  $Appartenance(\theta_6 \bar{\theta}_5 t_{q_0}, F, q_2)$



1. Ici,  $source(\theta_6) = q_2$ .
2. Un appel récursif est fait sur  $Appartenance(\bar{\theta}_5 t_{q_0}, F, q_3)$
3.  $source(\bar{\theta}_5) \neq q_3 = dest(\theta_6)$ . Cet appel récursif retourne faux.
- $Appartenance(\bar{\theta}_5 \theta_6 t_{q_0}, F, q_2)$ 
  1.  $a = \bar{\theta}_5 \in \bar{\Theta}$  et  $\gamma' = \theta_6 t_{q_0}$
  2. Les dérivés avec  $\theta_r$  sont produits. Le seul  $\theta_r$  compatible avec  $\bar{\theta}_5$  est  $\theta_7$ .
  3.  $dest(\theta_5) = q_2$ 
    - $Appartenance(\theta_7 \theta_6 t_{q_0}, F, q_2)$ 
      - (a)  $source(\theta_7) = q_3 \neq q_2$
      - (b) Cet appel récursif retourne faux.
    - $Appartenance(\theta_6 \theta_7 t_{q_0}, F, q_2)$ 
      - (a)  $source(\theta_6) = q_2$
      - (b)  $dest(\theta_6) = q_3$
      - (c) Appel récursif sur  $Appartenance(\theta_7 t_{q_0}, F, q_3)$
      - (d)  $source(\theta_7) = q_3$  et  $dest(\theta_7) = q_0$
      - (e) Cet appel lance lui même un appel récursif sur  $Appartenance(t_{q_0}, F, q_0)$  qui est vrai

## 4.4 Algorithme de construction de $\mathcal{F}(L)$

Soit un langage  $L$  et un automate  $A$  tel que  $L = L(A)$ . Calculer  $\mathcal{F}(L)$  revient, par définition, à calculer  $\bigcup_{\theta \in \Theta} Post(L, \theta) \cup \{t_{q_0}\}$ .

Pour rappel, voici la définition de  $Post(L, \theta)$  telle que donnée dans la section 3.2.3 :

$Post(L, \theta) = \bigcup_{\gamma \in L} Post(\gamma, \theta)$  avec

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

Des automates  $A_\theta$  peuvent être construits pour  $Post(L, \theta)$  pour chaque  $\theta$  à partir de  $A$ . L'union de ces automates  $A' = \bigcup_{\theta \in \Theta} A_\theta$  accepte  $\bigcup_{\theta \in \Theta} Post(L, \theta)$ . L'union de ce nouvel automate avec un automate pour le langage  $L = \{t_{q_0}\}$  accepte  $\mathcal{F}(L)$ .

La suite de cette section s'intéresse donc au calcul de  $L' = Post(L, \theta)$  où  $L$  est accepté par  $A$  et  $L'$  par un ADF  $A'$  construit à partir de  $A$ .

L'automate  $A'$  peut être indéterministe. Ce n'est pas un problème car, comme l'explique la section 2.3.6, il existe un ADF équivalent. Dans le cadre de l'apprentissage avec Angluin, c'est cet ADF qui devient  $A'$  et est utilisé pour les autres opérations.

Soit  $\theta \in \Theta$  tel que  $\delta(\theta) = (p, action, q)$ . On explique ci-après les différents cas à traiter pour calculer  $A_\theta$ .

#### 4.4.1 Si *action* est $\tau$ ou de la forme $c!a$

Dans ce cas, construire un automate pour  $L' = L \cap \Sigma^* t_p$  avec  $t_p$  correspondant au  $p$  de  $\delta(\theta)$  et  $\Sigma^* t_p$  étant représenté par l'ANF de la figure 4.1.

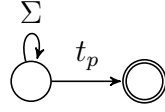
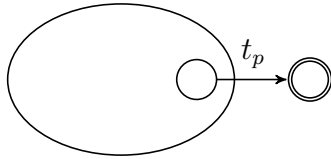


FIGURE 4.1: ANF pour  $L = \Sigma^* t_p$

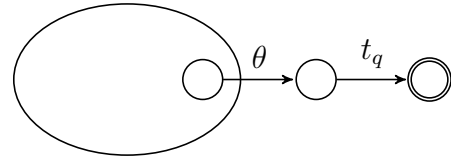
Dans cet automate  $L'$ , seuls les états et transitions menant à l'état final par  $t_p$  sont considérés.

Construire un automate pour  $L \cap \Sigma^* t_p = L'$ . Si  $L'$  est vide, c'est qu'il n'y a pas de mot finissant concerné par  $\theta$  et donc que  $Post(L, \theta) = \emptyset$ .

Sinon, si  $L' \neq \emptyset$ , supposons d'abord que *action* est de la forme  $c!a$  ou  $\tau$ . Il faut alors appliquer une transformation à l'automate correspondant à  $Post(L', \theta)$ . Si *action* est de forme  $c!a$ , la transition sur  $\theta$  peut être ajoutée avant les transitions sur  $t_p$  et un nouveau  $t_q$  signalant le changement d'état dans l'automate à files  $F$ .



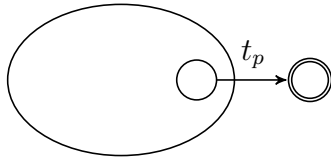
(a) Automate de  $L'$



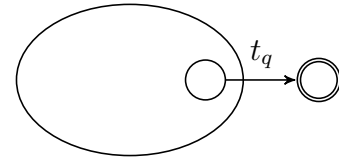
(b) Automate de  $Post(L', \theta)$

FIGURE 4.2: Application de  $Post(L', \theta)$  sur un automate si l'action est de forme  $c!a$

À présent, supposons que l'action de  $\theta$  est de forme  $\tau$ ,  $\theta \notin \Phi$ . Alors,  $t_p$  est remplacé par  $t_q$ . La transition est alors sous-entendue.



(a) Automate de  $L'$



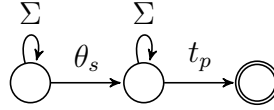
(b) Automate de  $Post(L', \theta)$

FIGURE 4.3: Application de  $Post(L', \theta)$  sur un automate si l'action est  $\tau$

#### 4.4.2 Si *action* est de la forme $c?a$

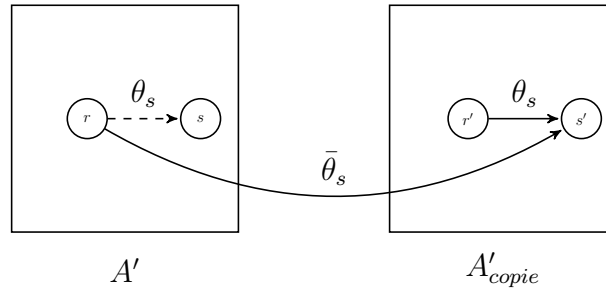
Pour qu'une réception  $c?a$  d'un symbole soit possible, il faut qu'il existe un envoi qui lui soit associé. Cet envoi est noté  $\theta_s$  tel que  $\delta(\theta_s) = (r, c!a, s)$ . De plus, il faut, comme dans la section précédente, tester si  $Post(L, \theta)$  est vide ou non grâce au  $t_p$  correspondant au  $p$  de  $\delta(\theta)$ .

En utilisant  $\theta_s$ , on construit  $L' = L \cap \Sigma^* \theta_s \Sigma^* t_p$ . Un automate acceptant ce langage est donné à la figure 4.4.

FIGURE 4.4: ANF pour  $L = \Sigma^*\theta_s\Sigma^*t_p$ 

Si  $L' = \emptyset$ , il n'existe pas de  $\theta_s$  dans les chemins qui nous concernent. Il n'y aura donc pas de possibilité de consommer un symbole :  $Post(L, \theta) = \emptyset$ .

Sinon si  $L'$  est non vide, il faut alors, à partir de l'automate  $A'$  acceptant  $L'$ , construire un automate  $A_\theta$  acceptant le langage  $Post(L', \theta)$ .

FIGURE 4.5: Première étape de construction de l'automate  $P_A$  pour  $Post(L'', \theta)$ 

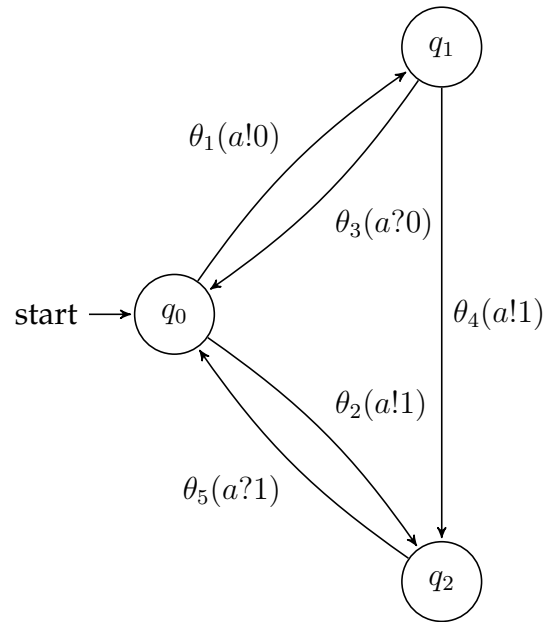
La figure 4.5 décrit comment l'automate  $A_\theta$  est construit à partir de  $A'$ . Premièrement, une copie de  $A'$  est construite :  $A'_{copie}$ . Dans  $A'$ , les états finaux sont considérés comme des états non finaux. Dans  $A'_{copie}$ , l'état initial est considéré comme un état non initial. Finalement, dans  $A'$ , toute transition  $\theta_s$  allant de  $r$  à  $s$  est remplacée par une transition sur  $\bar{\theta}_s$  allant de  $r$  à  $s'$ .

De cette façon, pour qu'un mot soit valide, il faut qu'il commence dans  $A'$ . La première occurrence de  $\theta_s$  a été systématiquement remplacée pour  $\bar{\theta}_s$  menant à  $A'_{copie}$ . Dans  $A'_{copie}$ , les autres transitions ne sont pas modifiées, laissant intact le reste du chemin. Cependant, la figure 4.5 a elle seule ne suffit pas à construire  $A_\theta$ . Pour rester en accord avec la définition de  $Post(L, \theta)$ , il faut adapter le dernier symbole, ici  $t_p$  en  $t_q$  (figure 4.6).

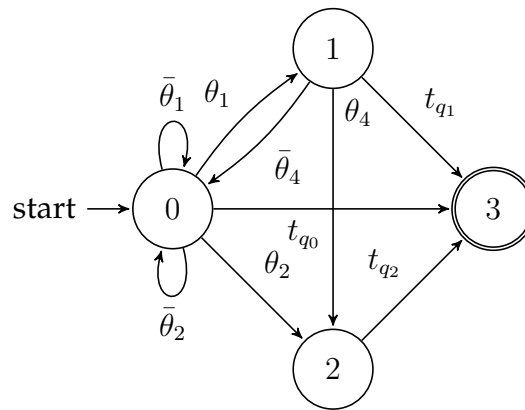
FIGURE 4.6: Deuxième étape de construction de l'automate  $A_\theta$  pour  $Post(L', \theta)$ 

### 4.4.3 Exemple

Dans cette section, on illustre la construction de  $\mathcal{F}(L)$  donnée dans la section précédente. Cette exemple se base sur l'automate à files  $F$  de la figure 4.7 ainsi que l'ADF  $A$  de la figure 4.8 acceptant le langage de traces annotées  $L$ .

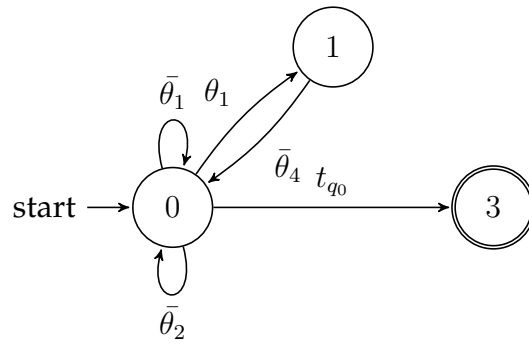
FIGURE 4.7: Automate à files  $F$ 

$F$  et  $L(F)$  candidat à  $AL(F)$     Regex pour  $AL(F)$  :  $(\bar{\theta}_1|\bar{\theta}_2)^*(\theta_1|\theta_2)?$

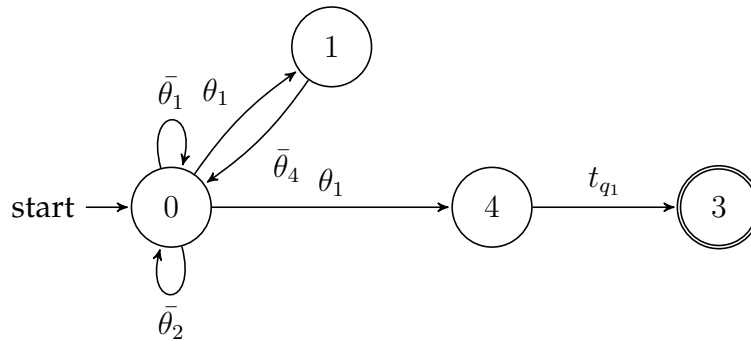
FIGURE 4.8: Automate  $A$  acceptant  $L$ 

Pour simplifier la lecture de cet exemple, les états de  $A$  pour lesquels il existe une transition  $t_p$  vers un état final sont appelés *états pré-finaux* sur  $t_p$ .

- Soit  $\theta_1$  avec  $\delta(\theta_1) = (q_0, a!0, q_1)$  menant au calcul de  $A_{\theta_1}$ . Les états pré-finaux sur  $q_0$  sont  $\{0\}$ . Un automate est construit pour  $L' = L \cap \Sigma^* t_{q_0}$  : 4.9.

FIGURE 4.9: Un ANF représentant  $L'$ 

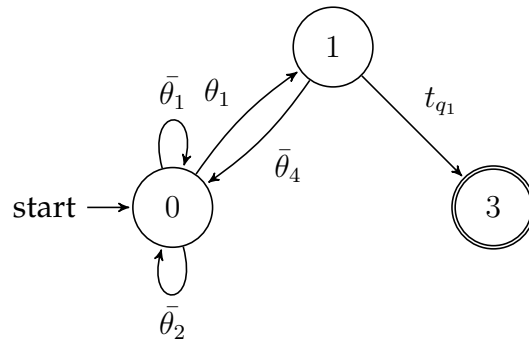
En suivant la règle pour les  $\theta$  dont l'action est de type  $c!a$ , ce qui est le cas pour  $\theta_1$ , un nouvel état est créé entre 0 et 3 car il y a une transition  $t_{q_0}$  à cet endroit. Cela donne l'ANF de la figure 4.10

FIGURE 4.10: L'ANF  $A_{\theta_1}$  acceptant  $Post(L, \theta_1)$ 

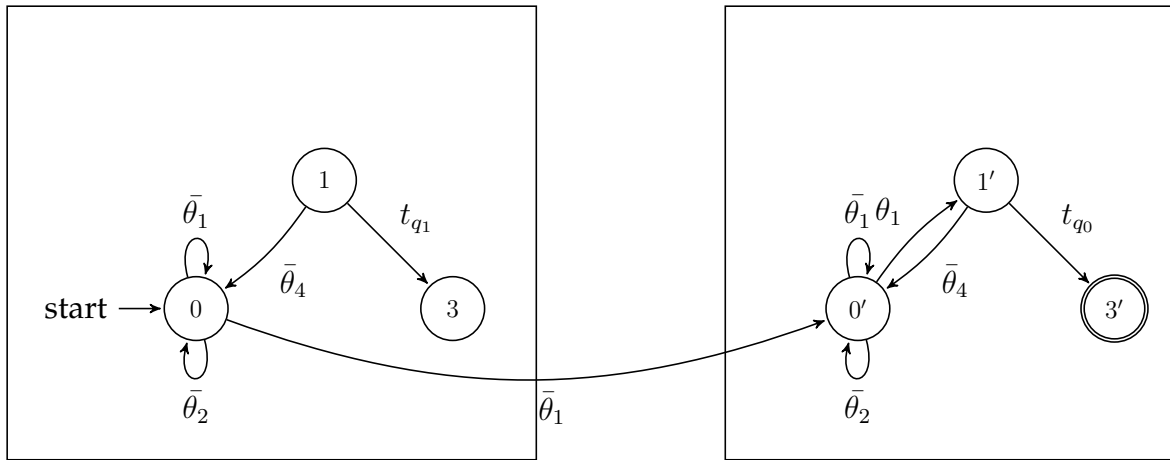
Cet automate 4.10 n'est pas déterministe mais il existe un équivalent déterministe qui peut être utilisé pour l'algorithme d'Angluin.

- Soit  $\theta_3$  avec  $\delta(\theta_3) = (q_1, a?0, q_3)$  menant au calcul de  $A_{\theta_3}$ . Les états pré-finaux sur  $q_1$  sont  $\{1\}$ . Contrairement à  $\theta_1$ ,  $\theta_3$  est une action de réception. Ici le seul symbole  $\theta_s$  d'action  $c!a$  correspondant à  $\theta_3$  ( $a!0$ ) est  $\theta_1$ .

Dès lors, l'ANF pour  $L' = L \cap \Sigma^*.\theta_3.\Sigma^*.t_{q_1}$  est calculé.

FIGURE 4.11: Un ANF représentant  $L'$ 

Cet automate sert à construire  $Post(L, \theta_3)$  comme décrit pour les  $\theta$  dont l'action est de forme  $c?a$  : 4.12.

FIGURE 4.12: Un ANF représentant  $Post(L, \theta_3)$ 

En répétant ces opérations pour les symboles  $\{\theta_2, \theta_4, \theta_5\}$ , il est possible de calculer l'automate  $A'$  acceptant  $L' = \bigcup_{\theta \in \Theta} Post(L, \theta)$ . L'union de cet automate  $A'$  avec un automate acceptant  $\{t_{q_0}\}$  résulte en un automate  $A''$  acceptant  $\mathcal{F}(L)$ .

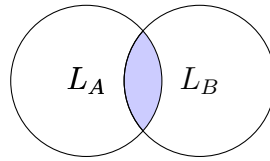
## 4.5 Algorithme de comparaison entre $\mathcal{F}(L)$

Supposons être en possession de  $L$  et de  $\mathcal{F}(L)$ . Nous décrivons ici comment implémenter la comparaison de  $\mathcal{F}(L)$  avec  $L$  selon les critères énoncés dans la section ??.

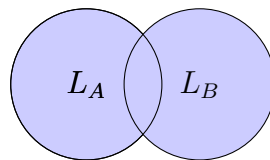
L'algorithme de comparaison 4.5.1 suppose l'existence d'opérations de base. Dans le cadre d'une implémentation, celles-ci doivent également être disponibles.

Soient deux ADFs  $A$  et  $B$  ainsi que les langages qui sont représentés par ceux-ci, respectivement  $L_A = L(A)$  et  $L_B = L(B)$ . Voici une liste des opérations nécessaires.

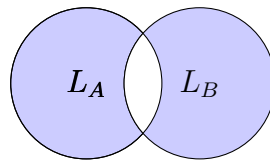
- La conjonction :  $L_A \cap L_B$

FIGURE 4.13:  $L_A \cap L_B$  en bleu

- L'équivalence :  $L_A$  est-il égal à  $L_B$  ?
- La disjonction :  $L_A \cup L_B$

FIGURE 4.14:  $L_A \cup L_B$  en bleu

- La disjonction exclusive  $((A \cup B) \setminus (A \cap B))$ , notée  $A \oplus B$

FIGURE 4.15:  $L_A \oplus L_B$  en bleu

- La différence avec le vide :  $L_A = \emptyset$  ?
- La sélection d'un mot :  $w \in L_A$

La comparaison de  $\mathcal{F}(L)$  avec  $L$  est plus complexe qu'une simple équivalence car le contre-exemple à retourner dépend de l'automate à files  $F$ . En effet, le contre-exemple recherché n'est pas un contre-exemple à l'égalité  $L = \mathcal{F}(L)$  mais à l'égalité  $L = AL(F)$ . (Voir section ??)

#### Algorithme 4.5.1 (Comparaison)

**Requis:** Un langage de trace annotée  $L$  pour un automate à files  $F$ ,  $\mathcal{F}(L)$

**Promet:** L'égalité entre  $L$  et  $\mathcal{F}(L)$  ou un contre-exemple  $\gamma \in \Phi^*$  à l'égalité  $L = AL(F)$

- 1: **si**  $L = \mathcal{F}(L)$  **alors**
- 2:     **retourner**  $L$  est un point fixe de  $\mathcal{F}$ .
- 3: **sinon**
- 4:      $X = L \oplus \mathcal{F}(L)$  { $X$  est non-vide car  $L \neq \mathcal{F}(L)$ }
- 5:     Prenons  $\gamma \in X$
- 6:     **si**  $\gamma \in L$  **alors**
- 7:         **retourner**  $\gamma$

```

8:  sinon
9:    si  $\gamma$  est une annotation valide alors
10:      retourner  $\gamma$ 
11:    sinon
12:      retourner  $\text{reverseFL}(\mathcal{F}(L), F, \gamma)$ 
13:    fin si
14:  fin si
15: fin si

```

A l'exception du mot  $t_{q_0}$ ,  $\mathcal{F}(L)$  contient des mots  $\gamma$  tels que  $\exists \gamma' \in L \exists \theta \in \Theta, \gamma = \text{Post}(\gamma', \theta)$ .

Sous l'hypothèse que  $\gamma$  n'est pas  $t_{q_0}$ , ce qui en ferait une annotation valide, rendant inutile l'appel à  $\text{reverseFL}$ ,  $\text{reverseFL}$  trouve ce  $\gamma'$  et  $\theta$ .

#### Algorithme 4.5.2 (reverseFL)

**Requis:** — Un langage obtenu par la fonction  $\mathcal{F} : \mathcal{F}(L)$

— L'automate à files concerné par  $L : F$

— Une trace annotée  $\gamma \in \Phi^*$

**Promet:** une trace annotée  $\gamma' \in L$  tel que  $\exists \theta, \gamma = \text{Post}(\gamma', \theta)$  s'il existe

```

1: si  $\gamma$  est une trace correctement annotée alors
2:    $\gamma' t_q \leftarrow \gamma$  avec  $\gamma' \in \Phi^*, t_q \in T_Q$ 
3:   si  $\exists \theta_\tau$  tel que  $\text{dest}(\theta_\tau) = t_q$  et l'action de  $\theta_\tau$  est  $\tau$  alors
4:     si  $\gamma' = \epsilon$  ou  $\text{dest}(\gamma') = \text{source}(\theta_\tau)$  alors
5:       retourner  $\gamma'.\text{source}(\theta_\tau)$ 
6:     fin si
7:   fin si
8:   si  $\gamma' = \gamma''.\phi$  avec  $\gamma'' \in \Phi^*$  et  $\phi \in \Phi$  alors
9:     si  $\phi \in \Theta$  et  $\text{dest}(\phi) = q$  alors
10:      retourner  $\gamma''.t_{\text{source}(\phi)}$ 
11:     fin si
12:   pour chaque symbole  $\bar{\theta} \in \gamma''.\phi$  de droite à gauche tels que  $\bar{\theta} \in \bar{\Theta}$  faire
13:      $u_1.\bar{\theta}.u_2.t_q \leftarrow \gamma$  avec  $u_1, u_2 \in \Phi^*$ 
14:      $(r, c!m, s) \leftarrow \delta(\bar{\theta})$ 
15:     pour chaque symbole  $\theta_r \in \Theta$  faire
16:       si  $\delta(\theta_r) = (p, c?m, q)$  avec le  $q$  de  $t_q$  alors
17:         retourner  $u_1.\theta_r.u_2.t_p$ 
18:       fin si
19:     fin pour
20:   fin pour
21: fin si
22: fin si

```

#### 4.5.1 Exemple d'application de reverseFL

Soit l'automate à files  $F$  dans la figure 2.16. Supposons être en possession d'un ADF  $A$  tel que  $L(A) = AL(F)$ . Comme  $AL(F) = \mathcal{F}(AL(F))$ ,  $\text{reverseFL}$  peut être employé avec  $A$ . Soit la



trace annotée  $\gamma = \bar{\theta}_2\bar{\theta}_5\bar{\theta}_5\bar{\theta}_1\theta_4t_{q_0} \in AL(F)$ .

Voici le déroulement de  $reverseFL(A, F, \gamma)$ .

1.  $\gamma$  finit par  $\theta_4 \in \Theta$  mais  $dest(\theta_4) \neq q_0$
2.  $\bar{\theta}_0 \in \bar{\Theta}$  pour lequel il existe  $\theta_3$  avec  $\delta(\theta_3) = (q_1, a?0, q_3)$ .  $q_3 \neq q_0$
3.  $\bar{\theta}_4 \in \bar{\Theta}$  pour lequel il existe  $\theta_7$  avec  $\delta(\theta_7) = (q_3, b?0, q_0)$ .  $q_0$  est correct. L'algorithme retourne  $\gamma' = (\theta_2\bar{\theta}_5).\theta_5.(\bar{\theta}_1).t_{q_3}$

Le résultat peut être vérifié simplement.  $Post(\gamma', \theta_7) = \gamma$ .

## 4.6 Expérimentations

Plusieurs expérimentations sont nécessaires pour se convaincre du bon fonctionnement global de la vérification de la sécurité d'automates à files par apprentissage actif. Trois cas sont retenus.

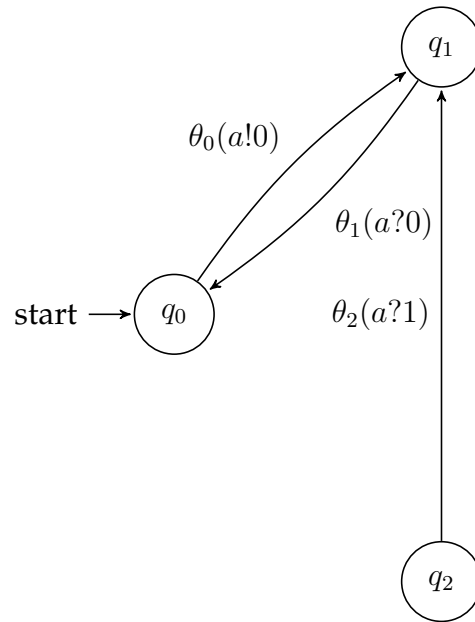
- L'algorithme s'arrête en déclarant l'automate sécurisé.
- L'algorithme s'arrête en déclarant l'automate comme étant à risque.
- L'algorithme ne s'arrête pas.

Le programme a été exécuté sur différents exemples  $F$  et le résultat a été confirmé par rapport à l'automate étudié. L'évolution de l'ADF  $A$  candidat pour accepter  $AL(F)$  a été exportée sous forme d'images. Dans ces images, représentant des ADFs, une notation a été établie.

- Une transition sur  $ti$  dans l'image représente une transition sur  $\theta_i$  dans  $A$ .
- Une transition sur  $bi$  dans l'image représente une transition sur  $\bar{\theta}_i$  dans  $A$ .
- Une transition sur  $si$  dans l'image représente une transition sur  $t_{q_i}$  dans  $A$ .

### 4.6.1 Exécution avec arrêt

L'automate à files  $F$  de la figure 4.16 permet d'étudier les deux premiers cas.

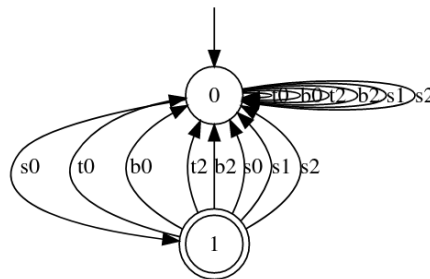
FIGURE 4.16: Automate à files  $F$ 

La question de la sécurité de cet automate à file  $F$  est posée pour deux ensembles  $\mathcal{W}$  :

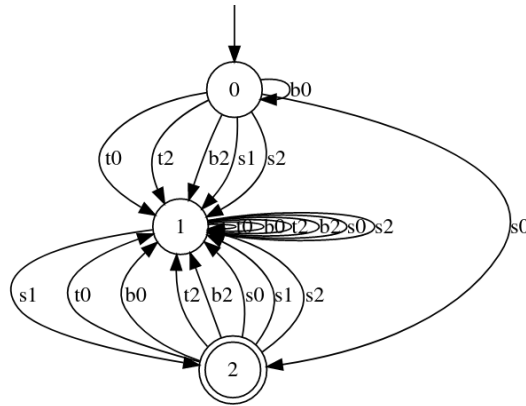
- $\mathcal{W}_1$  : Toute configuration de l'automate à file où l'état courant est  $q_1$ .
- $\mathcal{W}_2$  : Toute configuration de l'automate à file où l'état courant est  $q_2$ .

Voici les différentes itérations de l'algorithme avant l'arrêt du programme.

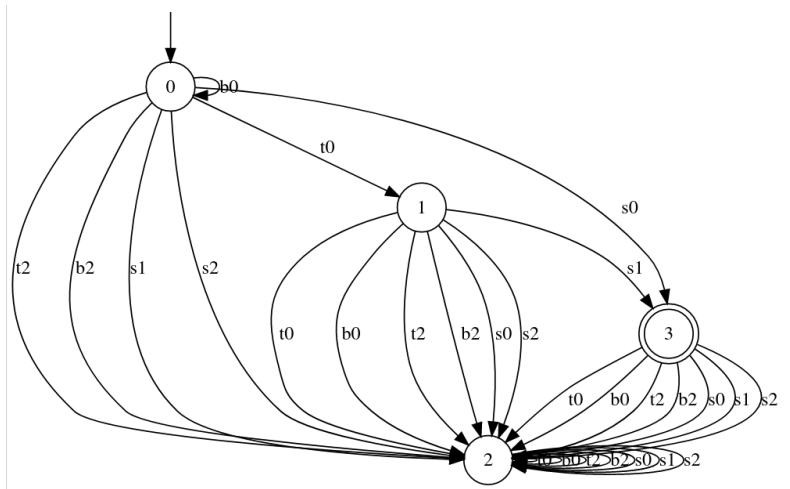
1. L'automate  $A_1$  acceptant le langage  $L = \{t_{q_0}\}$  est proposé (figure 4.17). Celui-ci est refusé et un contre-exemple est fourni :  $\gamma = \theta_0 t_{q_1}$ . En effet,  $A_1$  n'accepte pas  $\gamma$  alors qu'il existe bien une exécution valide de  $F$  pour ce mot.

FIGURE 4.17: Automate  $A_1$  lors du premier appel à l'oracle d'équivalence

2. L'automate  $A_2$  de la figure 4.18 est proposé. Il est refusé et le contre-exemple  $\gamma = \theta_1 t_{q_1}$  est retourné. En effet, il est accepté par  $A_2$  mais ne peut pas être exécuté par  $F$ .

FIGURE 4.18: Automate  $A_2$  lors du second appel à l'oracle d'équivalence

3. L'automate  $A_3$  de la figure 4.19 est proposé. Aucun contre-exemple n'est trouvé contre  $L(A) = AL(F)$ . Alors, les ensembles  $\mathcal{W}$  sont évalués.

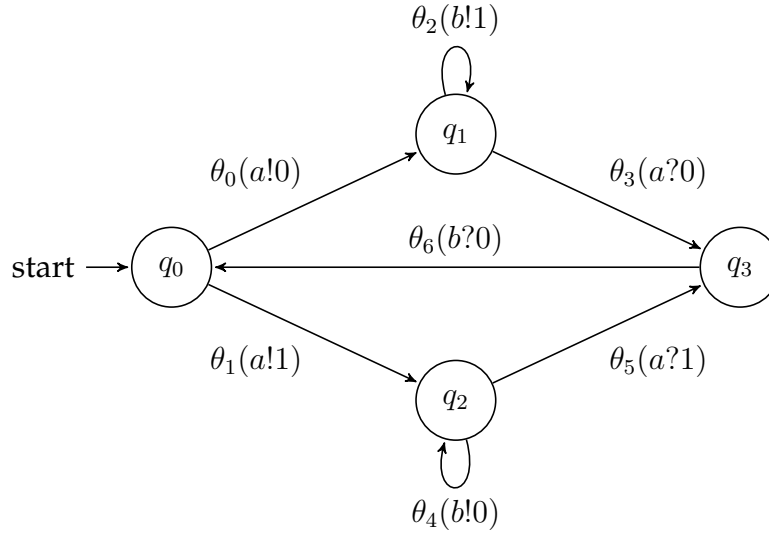
FIGURE 4.19: Automate  $A_3$  lors du troisième appel à l'oracle d'équivalence

D'après  $A_3$ , il est possible de suivre le chemin  $\gamma = \theta_1 t_{q_1}$ . Ce chemin passe par  $q_1$ .  $F$  est alors déclaré comme à risque pour  $\mathcal{W}_1$ .

Par contre, aucun chemin valide ne passe par  $q_2$ .  $F$  est alors déclaré comme sécurisé pour  $\mathcal{W}_2$ .

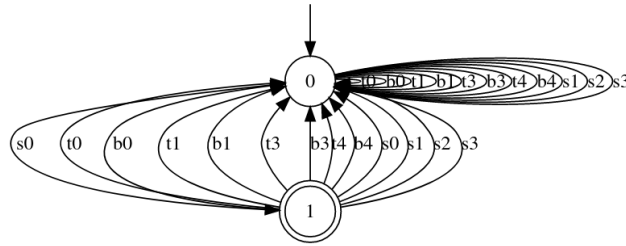
#### 4.6.2 Exécution sans arrêt

Pour cette troisième possibilité, l'algorithme qui ne s'arrête pas, l'automate à files  $F$  utilisé est donné à la figure 4.20. Au nom des transitions prêt, indexées à partir de zéro, il s'agit du même automate à file que celui de la figure 2.16.

FIGURE 4.20: Automate à files  $F$ 

Voici le suivi de l'exécution du programme jusqu'à ce qu'un motif récurrent émerge, suggérant que l'exécution ne s'arrête pas.

1. L'automate  $A_1$  acceptant le langage  $L = \{t_{q_0}\}$  est proposé (figure 4.17). Celui-ci est refusé et un contre-exemple est fourni :  $\gamma = \theta_0 t_{q_1}$ . En effet,  $A_1$  n'accepte pas  $\gamma$  alors qu'il existe bien une exécution valide de  $F$  pour ce mot. C'est la même situation que dans la sous-section précédente.

FIGURE 4.21: Automate  $A_1$  lors du première appel à l'oracle d'équivalence

2. L'automate  $A_2$  est proposé et refusé. Le contre-exemple  $\gamma = \bar{\theta}_0 t_{q_3}$  est retourné. En effet, le chemin  $w = \theta_0. \theta_3$  est valide dans  $F$ , mène bien à  $q_3$  et  $\mathcal{A}(w) = \gamma$ .  $\gamma$  aurait dû être accepté par  $A_2$ .
3. L'automate  $A_3$  est proposé et refusé. Le contre-exemple  $\gamma = \theta_3 t_{q_3}$  est retourné.  $\gamma$  ne correspond à aucune exécution valide de  $F$ , ce mot ne devrait pas être accepté par  $A_3$ .
4. L'automate  $A_4$  est proposé et refusé. Le contre-exemple  $\gamma = \bar{\theta}_1 \theta_3 t_{q_3}$  est retourné, étant accepté par  $A_4$  à tort.
5. L'automate  $A_5$  est proposé et refusé. Le contre-exemple  $\gamma = \bar{\theta}_1 \bar{\theta}_4 \theta_4 t_{q_0}$  est retourné, le mot  $\gamma$  n'étant pas accepté par  $A_5$  alors qu'il devrait l'être.

6. L'automate  $A_6$  de la figure 4.22 est proposé. Il a été simplifié depuis l'automate original par soucis de lisibilité.

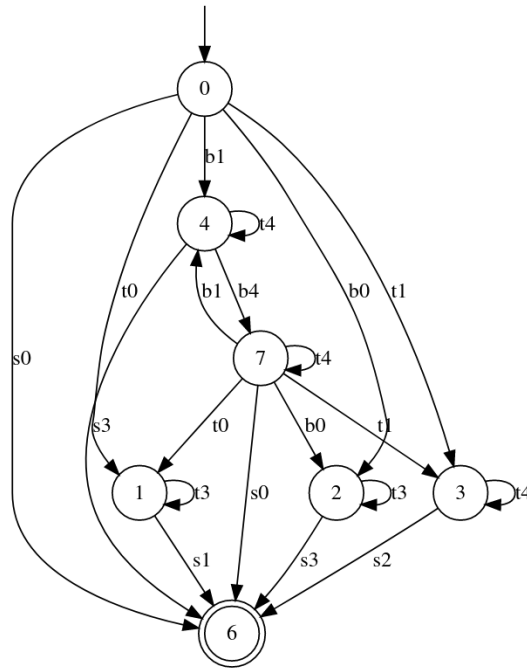
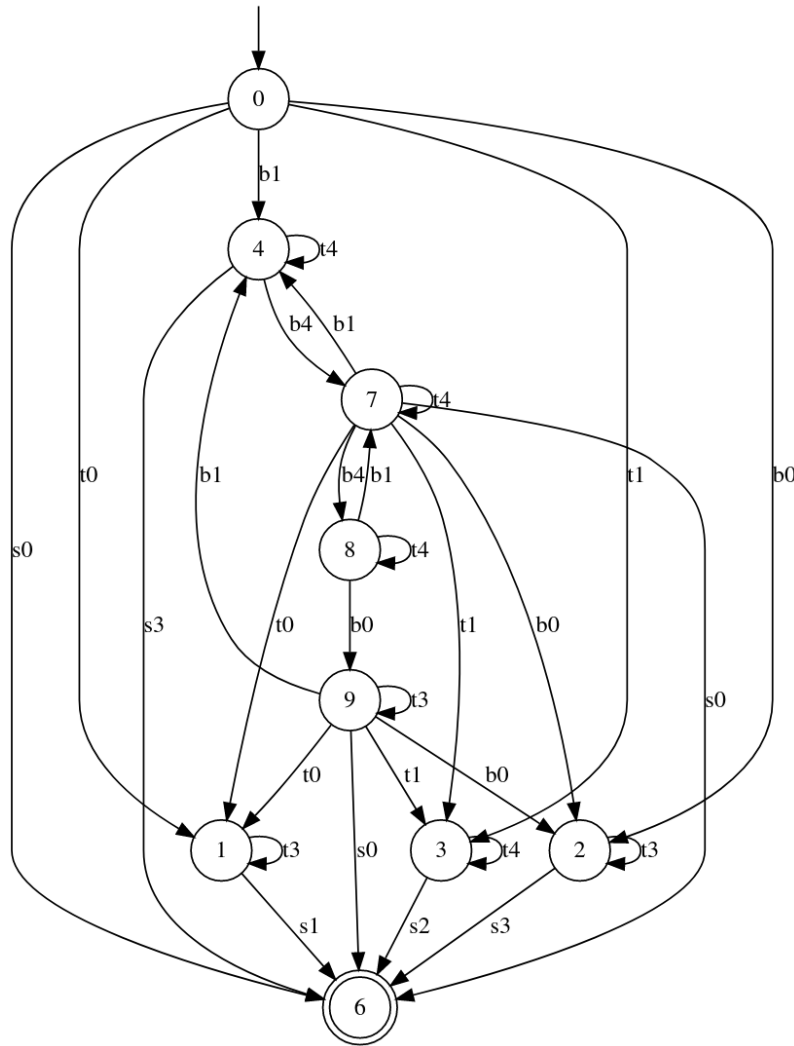


FIGURE 4.22: Automate  $A_6$  lors du sixième appel à l'oracle d'équivalence

On peut déjà y remarquer une transition sur  $\bar{\theta}_4$  entre les états 4 et 7. Cependant, ce n'est pas suffisant. Un contre-exemple  $\gamma = \bar{\theta}_1\bar{\theta}_4\bar{\theta}_4\bar{\theta}_0t_{q_0}$  est fourni. Le mot  $\gamma$  n'est pas accepté par  $A_6$  alors qu'il devrait l'être. Auquel cas, une exécution valide dans  $F$  est :  $w = \theta_1\theta_4\theta_4\theta_5\theta_6\theta_0\theta_3\theta_6$ .  $w$  est bien une exécution valide dans  $F$  menant à  $q_0$ . De plus,  $\mathcal{A}(w) = \gamma$ .

7. L'automate  $A_7$  de la figure 4.23 est proposé. Il est également simplifié pour plus de lisibilité. Néanmoins, il reste plus complexe que  $A_6$ . En particulier, on peut noter l'apparition des états 8 et 9, servant à compter une utilisation de  $\theta_4$  supplémentaire. Un contre-exemple  $\gamma = \bar{\theta}_1\bar{\theta}_4\bar{\theta}_4\bar{\theta}_4\bar{\theta}_0\bar{\theta}_0t_{q_0}$  est fourni. Il est semblable au contre-exemple précédent. En réalité, il contient une utilisation de  $\theta_4$  en plus.

FIGURE 4.23: Automate  $A_7$  lors du septième appel à l'oracle d'équivalence

À partir de là, l'automate se complexifie d'itération en itération, cherchant en vain à trouver un langage régulier permettant de garder le compte du nombre de transitions  $\theta_4$  suivies. C'est un comportement cohérent au vu de l'automate  $F$ . Cet automate simple fait partie de la classe des automates à files pour lesquels cette méthode est insuffisante.

# **Chapitre 5**

## **Conclusion**

**5.1 Rappel de la problématique**

**5.2 Résumé des notions rencontrées**

**5.3 Apports**

**5.4 Suites possibles**

# Bibliographie

- [1] D. ANGLUIN, *Learning regular sets from queries and counterexamples*, Information and Computation, 75 (1987), pp. 87 – 106.
- [2] J. E. HOPCROFT, *Introduction to Automata Theory, Languages, and Computation (2nd Edition)*, Addison Wesley, nov 2000.
- [3] J. E. HOPCROFT AND J. D. ULLMAN, *Introduction to automata theory, languages and computation. adison-wesley*, Reading, Mass, (1979).
- [4] D. C. KOZEN, *Automata and computability*, Springer Science & Business Media, 1997.
- [5] D. NEIDER, *Applications of automata learning in verification and synthesis*, PhD thesis, Hochschulbibliothek der Rheinisch-Westfälischen Technischen Hochschule Aachen, 2014.
- [6] 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

## Théorème de Knaster-Tarski

**Théorème A.0.1** *Soit une fonction monotone  $F$  portant sur des ensembles et  $P$  l'ensemble de points préfixes de  $F$ . Alors  $\mu = \bigcap_{p \in P} p$  est le point fixe minimal de  $F$ .*

Il s'agit en réalité d'une reformulation adaptée aux ensembles et outils utilisés.

### *Preuve A.0.1.1*

Soit une fonction monotone  $F$  portant sur des ensembles et  $P$  l'ensemble de points préfixes de  $F$ .

Posons  $\mu = \bigcap_{p \in P} p$ .

Par définition,  $\forall p \in P, \mu \subseteq p$ . Comme la fonction  $F$  est monotone, elle peut être appliquée aux deux ensembles :  $\forall p \in P, F(\mu) \subseteq F(p)$ .

Comme  $F(p) \subseteq p$  par définition, on obtient par transitivité de l'inclusion que  $\forall p \in P, F(\mu) \subseteq p$ . Si  $F(\mu)$  est inclus à tout ensemble  $p$ , c'est qu'aucun élément de  $F(\mu)$  ne fait pas partie d'un  $p$  en particulier : cela revient à la définition de l'intersection. Donc,  $F(\mu) \subseteq \bigcap_{p \in P} p = \mu$ . Cela prouve que  $\mu$  est un point préfixe ( $F(\mu) \subseteq \mu$ ). Comme  $F$  est monotone,  $F(F(\mu)) \subseteq F(\mu)$  et  $F(\mu)$  est également un point préfixe.

Or, comme  $\mu \subseteq p$  est vrai pour tout  $p$  et en particulier pour  $p = F(\mu)$  puisque  $F(\mu)$  est un point préfixe et appartient alors à  $P$ , on obtient  $\mu \subseteq (F(\mu))$ . Dès lors,  $\mu = F(\mu)$ ;  $\mu$  est le point fixe minimal (la minimalité venant du fait que  $\mu$  est déjà un point préfixe minimal).

□

# Annexe B

## Modifications à Learnlib et Automatalib

Cette annexe est une liste exhaustive des modifications apportées aux bibliothèques Automatalib et Learnlib pour leur permettre de supporter l'apprentissage automatique de la sécurité d'automates à files.

### B.1 Ajouts et modifications à Automatalib

Nouvelles classes avec leurs méthodes :

- FIFOA
- MutableCA (CA pour Channel Automata)
- MutableFIFOA
- SupportsGrowingChannelAlphabet
- ChannelNamesHolder
- TransitionAction
  - 1. Action (enum des trois types d'action possibles)
- CAGraphView (aide à la visualisation d'un automate utilisant des canaux)
- FIFOAVisualizationHelper (aide à la visualisation)
- PhiChar (les symboles  $\phi \in \Phi$  et leur correspondance dans l'automate à files  $F$  concerné)
  - Type (isTheta, isBarred, representsState)
  - getI (numéro du  $\theta$  ou  $t_q$  représenté)
- UniversalChannelCompact
  - Méthodes de bases de UniversalCompact
  - getAnnotationAlphabet (retourne  $\Phi$ )
  - getAdditionsAlphabet (retourne tous les  $\theta$  pour le calcul de  $\mathcal{F}(L)$ )
- AbstractFastTransition
- FastFIFOA
- FastFIFOAState
- FastFIFOATransition
- FastFIFOATransitionAction
- CompactFIFOA
  - Méthodes de base

- isCorrectAnnotatedTrace (la trace  $\gamma$  est-elle correctement annotée pour  $F$ ?)
- validateTrace (la trace  $\gamma$  est-elle valide pour  $F$ ?)
- FIFOBuilderImpl : utilitaire de création d'automates à files
- FIFOAs
  - applyFL (algorithme  $\mathcal{F}$ )
  - sigmaStarTP (automate pour  $L = \Sigma^*.t_p$ )
  - sigmaStarThetaSSigmaStarTP (automate pour  $L = \Sigma^*.\theta_s.\Sigma^*.t_p$ )
  - aPlusAPrime (utilitaire pour le cas de l'action de type  $c?a$  lors du calcul de  $\mathcal{F}(L)$ )
  - tq0 (automate pour  $L = \{t_{q_0}\}$ )
  - fuseDFAs (regroupe une liste d'ADFs)
  - reverseFL
  - reversehcj (limitée ici à  $\mathcal{W}$  défini par des langages réguliers définissant tout passage par un état de contrôle comme à risque)

Classes modifiées avec leur méthodes :

- Word : insert (pour l'ajout de  $\theta$  dans l'oracle d'appartenance)
- WordTest : insertTest
- AbstractCompact : méthodes pour récupérer un numéro d'état, de symbole ou transition
- AbstractCompactSimpleDeterministic : méthodes pour obtenir les transitions ou des sous-ensembles en fonction de l'état d'origine ou de destination
- CompactDFA : getLeadingTransitions (obtenir les transitions menant à un état  $p$ )
- CompactNFA : getLeadingTransitions (obtenir les transitions menant à un état  $p$ )
- Visualisation : méthodes relatives à la visualisation d'automates utilisant des canaux
- AutomatonBuilder : méthode de création d'automates à files
- NFAs : indeterminize (construire un ANF depuis un ADF)

## B.2 Ajouts et modifications à Learnlib

Nouvelles classes avec leurs méthodes :

- SafeException
- UnsafeException
- FIFOAEquivalenceOracle
- LeverExperiment
- ALFEQOracle
  - findCounterExemple (utilisant un automate à files)
  - fixpointCounterExemple (tester  $L = AL(F)$ )
  - getUnsafePath (trouver un chemin à risque grace à  $\mathcal{W}$ )
  - isPathValid (la trace est-elle correctement annotée et valide?)
- FIFOTraceSimulatorOracle
  - answerQuery (tester  $\gamma \in AL(F)$ )

Classes modifiées avec leur méthodes :

- Experiment : Experiment using a FIFOAEquivalenceOracle

## B.3 Module de démonstration

Classes du paquet de démonstration :

- FIFOA : exemple complet de l'apprentissage automatique de la propriété de sécurité d'un automate à files.
- TestAutomatons : ADFs et automates à files utilisés pour des tests unitaires
- AlphabetTest : test unitaire de l'utilisation de  $\Phi$
- FIFOATest
  - test unitaire sur les traces (sont-elles correctes ? valides ?)
  - test unitaires sur les méthodes utilitaires
- FLTest
  - tests sur les automates pour  $L = \text{Sigma}^*.t_p$
  - tests sur les automates pour  $L = \Sigma^*.\theta_s.\Sigma^*.t_p$
  - tests pour aplusAPrime
  - tests pour reverseFL