Parse & Merge : compilation de Règles de Réécriture en transducteurs à états finis

Richard Beaufort

Multitel ASBL
Avenue Copernic 1, 7000 Mons, Belgique
beaufort@multitel.be
http://www.multitel.be/TTS

ABSTRACT

Context-dependent rewrite rules, originally developed in the context of generative phonology, are now widely used in many areas of natural language and speech processing. We present *Parse & Merge*, a new algorithm to compile such rules into finite-state transducers, which is both simpler and more efficient than existing algorithms. Further, Our algorithm allows the definition of weighted and/or optional rules, because these variations are used in most real applications.

1. INTRODUCTION

Les règles de réécriture prennent la forme générale suivante :

$$\varphi \rightarrow \psi :: \lambda _ \rho \qquad (1)$$

qui exprime que, entouré de λ et ρ, φ doit être remplacé par ψ. Les règles sont ordonnées de la plus spécifique à la plus générale de sorte qu'une règle ne puisse s'appliquer que si aucune autre règle, plus spécifique et applicable, n'a été rencontrée. En outre, tout contexte d'une règle (λ, ρ) qui apparaît comme cible d'une règle sera lui-même réécrit : en somme, toutes les règles applicables à une entrée donnée lui seront appliquées, simultanément ou de manière récursive.

Développées dans le cadre de la phonologie générative, ces règles sont maintenant largement utilisées dans de nombreux domaines du traitement du langage naturel (prétraitement, morphologie, syntaxe, phonétique…) et de la parole.

Généralement, la description d'un domaine d'application nécessite de nombreuses règles de réécriture. L'efficacité du système dépendra donc énormément de la méthode utilisée pour représenter ces règles. Les machines à états finis, telles que les automates (FSAs) et les transducteurs (FSTs), dont Johnson [1] a montré qu'elles permettent de modéliser les règles de réécriture, constituent un cadre de travail idéal. Elles autorisent une représentation compacte de l'information [4] et acceptent des opérations telles que l'union, la concaténation et l'étoile de Kleene [10] – nécessaires à l'expression des langages réguliers –, mais aussi la composition [9, 5] – une généralisation de l'intersection qui fait partie intégrante du processus de compilation des règles de réécriture.

Cet article présente un nouvel algorithme de compilation de règles de réécriture sous la forme de FSTs : *Parse & Merge*. Plus simple et moins coûteux en temps de calcul que les algorithmes existants, *Parse & Merge* autorise en outre l'expression de règles de réécriture pondérées et/ou optionnelles, étant donné que ces variantes sont employées dans la plupart des applications réelles.

2. ALGORITHMES EXISTANTS

Les principaux algorithmes proposés afin de compiler des règles de réécriture sous la forme de FSTs sont ceux de Kaplan & Kay [2], Karttunen [3] et Mohri & Sproat [8]. Nous donnons ici un rapide aperçu de ces algorithmes. Nous encourageons cependant le lecteur à lire les articles concernés, auxquels ces quelques lignes ne rendent pas justice.

Quel que soit l'algorithme, le principe général est identique. Chaque partie d'une règle ($\varphi, \psi, \lambda, \rho$), est une expression régulière définie sur l'alphabet des règles. Chaque règle est compilée séparément. Les règles sont ensuite réunies en une seule machine par composition, dans l'ordre dans lequel elles ont été énoncées. La composition présente l'avantage de respecter les contraintes décrites précédemment : dans la machine obtenue, les règles spécifiques prévalent sur les règles générales, et toutes les règles applicables à une entrée lui sont appliquées simultanément.

La compilation d'une règle varie fortement d'une méthode à l'autre, mais présente toujours les constantes suivantes :

- Une règle nécessite toujours le calcul de plusieurs machines intermédiaires, dont le but est d'identifier clairement la présence du contexte gauche (λ) ou droit (ρ) de la règle, de manière à ce que le remplacement ($\varphi \rightarrow \psi$) ne soit réalisé que dans le contexte souhaité. L'identification des contextes est obtenue par insertion de marqueurs spécifiques dans les machines concernées, marqueurs qui sont ensuite supprimés à l'aide d'une machine supplémentaire. La complexité de l'insertion des marqueurs diffère fortement d'un algorithme à l'autre, mais le processus reste coûteux parce que les machines intermédiaires travaillent toutes sur la totalité de l'alphabet.
- Les différentes machines intermédiaires sont combinées par composition pour obtenir une seule machine qui représente la règle, dans laquelle les marqueurs ont disparu.

Pour la compilation d'une seule règle, toutes ces méthodes nécessitent donc de nombreuses opérations coû-

teuses, dont un certain nombre de compositions.

3. ALGORITHME « PARSE & MERGE »

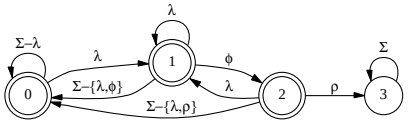
3.1. Principe

Notre algorithme diffère uniquement dans la manière de compiler le FST qui correspond à une règle donnée. Pour les besoins de l'exposé, nous supposons que λ, φ et ρ ne sont pas nuls, et que la règle $\varphi \rightarrow \psi :: \lambda _ \rho$ peut être appliquée n'importe où dans la séquence de symboles : $\varphi \rightarrow \psi :: .^{\star}\lambda _ \rho.^{\star}$.

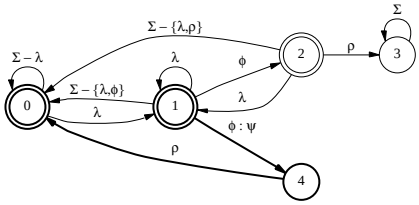
Nous partons du complément de la partie supérieure de la règle, $.^{\star}\lambda\varphi\rho.^{\star}$. Le complément d'un langage A défini sur un alphabet Σ représente « tout ce qui, dans $\Sigma^{\star}$, n'est pas A ». Par construction, le complément de $.^{\star}\lambda\varphi\rho.^{\star}$ accepte donc tout sauf $.^{\star}\lambda\varphi\rho.^{\star}$ et présente toujours λ (cf. Fig. 1,a).

L'idée à la base de l'algorithme est d'*insérer directement la règle dans le complément*, de sorte que le résultat représente la totalité du langage, et réalise le remplacement décrit par la règle.

Pour ce faire, le principe de base de l'algorithme consiste à repérer les états du complément où termine le contexte gauche λ, et à insérer, à partir de ces états, le remplacement ($\varphi \rightarrow \psi$) suivi du contexte droit ρ. Ce faisant, l'insertion doit veiller à ce que la règle s'intègre au reste du langage : pour cette raison, ρ doit rejoindre l'état initial du complément (cf. Fig. 1,b).



a.



b.

FIG. 1: A partir d'un FSA représentant le complément de $\lambda\varphi\rho$ (a), création d'un FST qui intègre la règle $\varphi \rightarrow \psi :: \lambda_\rho$ (b). Les doubles cercles représentent les états finaux.

Cependant, il arrive que dans certaines règles la fin (ou la totalité) du contexte droit ρ corresponde au début (ou à la totalité) du contexte gauche λ. C'est le cas, par exemple, de la règle « A{1,2} → B : : FD _ F ». Dans ce cas, l'algorithme doit veiller à ce que ρ rejoigne les états auxquels ρ termine dans le complément (cf. Fig. 2). De la sorte, toutes les instances de φ entourées de λ_ρ seront effectivement remplacées par ψ.

L'algorithme comprend donc les étapes suivantes :

1. Construire un FST, REPLACE, qui modélise le remplacement $\varphi \rightarrow \psi$ en présence du contexte droit ρ.
2. Construire le COMPLEMENT de $\lambda\varphi\rho$.
3. Intégrer REPLACE au COMPLEMENT entre les états où termine λ et les états où termine ρ dans le COMPLEMENT, à l'aide de 2 nouvelles opérations définies sur les machines à états finis : *Parse* et *Merge*.

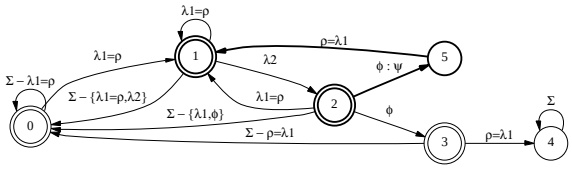


FIG. 2: FST, dérivé du complément, qui intègre la règle $\varphi \rightarrow \psi :: \lambda 1 \lambda 2_\rho$, avec $\rho = \lambda 1$.

Dans les sections suivantes, nous rappelons les calculs préliminaires nécessaires, avant de détailler la construction de REPLACE et du COMPLEMENT. Nous présentons ensuite les opérations *Parse* et *Merge*, et la manière dont elles sont employées pour intégrer REPLACE dans le COMPLEMENT.

3.2. Calculs préliminaires

Chaque partie de la règle (φ, ψ, λ, ρ) est compilée sous la forme d'un automate (Φ, Ψ, Λ, P) qui modélise strictement le langage de l'expression concernée : seuls les symboles de l'expression y sont donc combinés selon les opérateurs utilisés [11].

3.3. Calcul de REPLACE

REPLACE est défini comme suit :

$$\text{REPLACE} = (\Phi \times \Psi) \cdot \text{P} \tag{2}$$

Le produit cartésien $\Phi \times \Psi$ est obtenu par composition :

$$\Phi \times \Psi = (\Phi_{\rightarrow\epsilon}) \circ (\Psi_{\epsilon\rightarrow}) \tag{3}$$

où $\Phi_{\rightarrow\epsilon}$ est le transducteur qui projette φ en entrée sur ϵ en sortie (cf. Fig. 3,a), et $\Psi_{\epsilon\rightarrow}$ est le transducteur qui projette ϵ en entrée sur ψ en sortie (cf. Fig. 3,b).

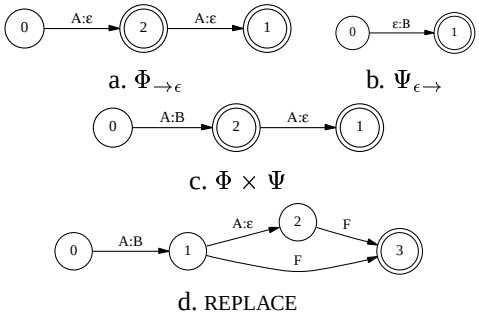


FIG. 3: Calcul de REPLACE. La règle représentée est A{1,2} → B : : (F|E)D ? _ F.

3.4. Calcul du COMPLEMENT

Le calcul du COMPLEMENT varie selon que la règle présente ou non des contraintes de début et/ou de fin :

– Pas de contrainte : $\lambda\varphi\rho$ peut survenir n'importe où dans la séquence de symboles :

$$\text{COMPLEMENT} = \overline{\Sigma^{\star}\lambda\varphi\rho\Sigma^{\star}} \tag{4}$$

– Contrainte de fin : $\lambda\varphi\rho$ doit terminer la séquence de symboles :

$$\text{COMPLEMENT} = \overline{\Sigma^\star\lambda\varphi\rho} \quad (5)$$

– Contrainte de début : $\lambda\varphi\rho$ doit se trouver au début de la séquence de symboles :

$$\text{COMPLEMENT} = \overline{\lambda\varphi\rho\Sigma^\star} \quad (6)$$

– Contrainte de début et de fin. Dans ce cas, $\lambda\varphi\rho$ doit commencer et terminer l'expression :

$$\text{COMPLEMENT} = \overline{\lambda\varphi\rho} \quad (7)$$

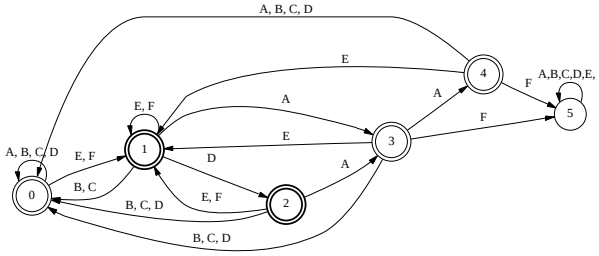
3.5. Intégration : Parse & Merge

Parse(A, B). Réalisée sur deux automates A et B, cette opération est une variante de l'intersection : il s'agit d'un parcours *en parallèle* de A et B, à partir de leurs états initiaux, qui se contente de stocker dans un vecteur V tous les états de A qui sont atteints *en même temps* que les états *finaux* de B (cf. Fig. 4). C'est cette opération qui permet d'identifier, dans le COMPLEMENT, les états où terminent λ (stockés dans V_λ) et ceux où terminent ρ (stockés dans V_ρ). Néanmoins, le calcul de ces vecteurs dépend des contraintes exprimées sur la règle :

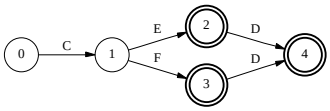
– V_λ est toujours calculé, à moins que λ ne soit nul. Dans ce cas, V_λ contient uniquemement l'état initial du COMPLEMENT.
– V_ρ n'est calculé que s'il n'y a pas de contrainte. S'il y a contrainte, $V_\rho = \emptyset$. Sinon, si ρ est nul, V_ρ contient uniquemement l'état initial du COMPLEMENT.
– S'il y a une contrainte de début (V_ρ est nul), mais pas de contrainte de fin, REPLACE est augmenté :

$$\text{REPLACE} = \text{REPLACE} \cdot \Sigma^\star \quad (8)$$

afin d'autoriser n'importe qu'elle suite de symboles après l'application de la règle.



Machine A

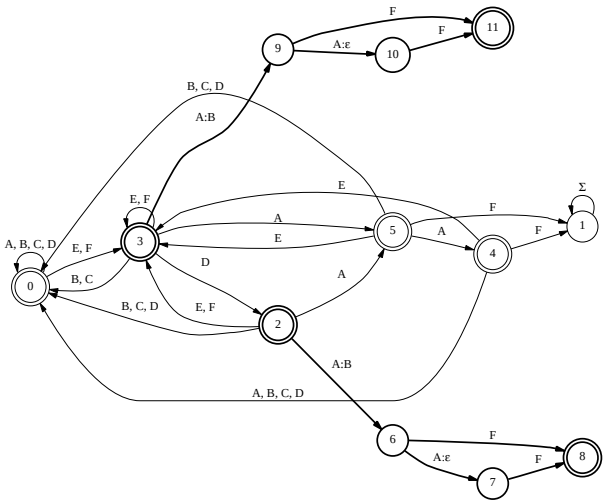


Machine B

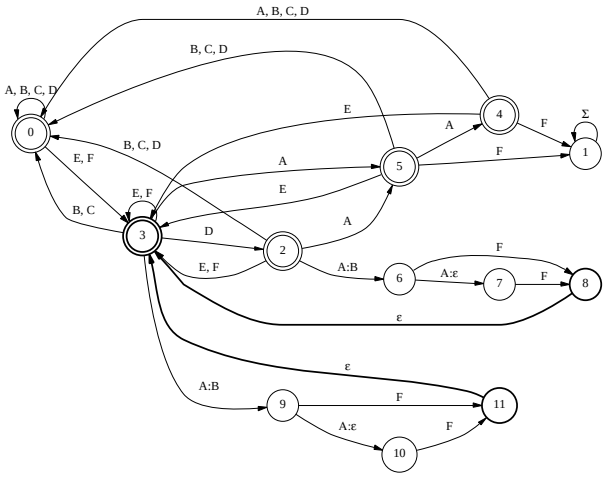
FIG. 4: Parse(A,B). Les états stockés de A (1,2) sont atteints en même temps que les états finaux de B (2,3,4).

Merge(COMPLEMENT, REPLACE, V_λ, V_ρ) intègre REPLACE dans le COMPLEMENT (cf. Fig. 5) à partir de chaque état de V_λ. Parmi les nouveaux états créés et propres au langage de REPLACE (cf. Fig. 5,a), certains sont finaux. Ces états deviennent non-finaux et sont reliés, à l'aide de transitions ϵ, aux états de V_ρ, pour autant que $V_\rho \neq \emptyset$ (cf. Fig. 5,b).

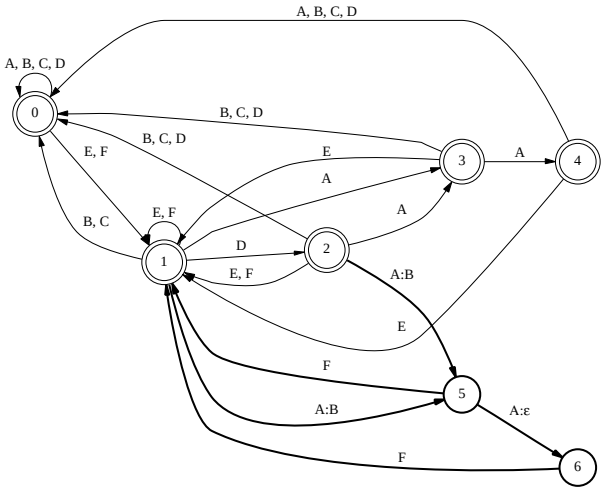
Après l'intégration, la machine est nettoyée de ses ϵ éventuels [6], déterminisée et minimisée [7] (cf. Fig. 5,c).



a. Intégration de REPLACE à partir de V_λ = {2,3}.



b. Liaison de REPLACE aux états de V_ρ = {3} à l'aide de transitions ϵ.



c. Après suppression des ϵ, déterminisation et minimisation.

FIG. 5: Règle A{1,2} → B : : (F|E)D ? _ F, définie sur l'alphabet {A, B, C, D, E, F}.

4. EXTENSIONS

4.1. Pondération des règles

Les règles de réécritures pondérées prennent la forme :

$$\varphi \rightarrow \psi :: \lambda _ \rho / \omega \tag{9}$$

qui exprime que le poids ω est attribué au remplacement $(\varphi \rightarrow \psi)$ dans le contexte $\lambda _ \rho$. Dans notre algorithme, le poids ω est intégré lors du calcul de REPLACE, qui devient :

$$\text{REPLACE} = (\Phi \times \Omega \times \Psi) \cdot \text{P} \tag{10}$$

Comme précédemment, le produit cartésien $\Phi \times \Omega \times \Psi$ est obtenu par composition :

$$\Phi \times \Omega \times \Psi = (\Phi_{\rightarrow \epsilon}) \circ (\Omega_{\epsilon}) \circ (\Psi_{\epsilon \rightarrow}) \tag{11}$$

où Ω_{ϵ} est le transducteur qui projette ϵ en entrée sur le poids ω en sortie (cf. Fig. 6).

a. Ω_{ϵ} b. REPLACE

FIG. 6: Calcul de REPLACE. La règle représentée est A{1,2} → B : : (F|E)D ? _ F / 0.25.

4.2. Règles optionnelles

Les règles de réécritures optionnelles prennent la forme :

$$\varphi \rightsquigarrow \psi :: \lambda _ \rho / \omega \tag{12}$$

qui exprime que le remplacement $(\varphi \rightsquigarrow \psi)$ est facultatif. Dans notre algorithme, le calcul de REPLACE devient :

$$\text{REPLACE} = \left[(\Phi \times \Omega \times \Psi) \cup \Phi\right] \cdot \text{P} \tag{13}$$

De la sorte, il y a deux chemins pour φ dans REPLACE : l'un réalise la règle, l'autre laisse φ inchangé (cf. Fig. 7).

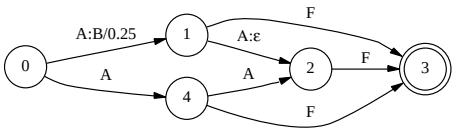


FIG. 7: Calcul de REPLACE. La règle représentée est A{1,2} ⇝ B : : (F|E)D ? _ F / 0.25.

5. EVALUATION

Le moins coûteux des algorithmes antérieurs est celui de Mohri & Sproat (M&S) [8]. Une règle y est cependant le résultat de la combinaison de 5 machines travaillant toutes sur la totalité de l'alphabet : r (ρ), f (φ), $replace$ $(\varphi \rightarrow \psi)$, $l1$ et $l2$ (λ). En outre, l'algorithme demande 4 insertions de marqueurs, 2 opérations *reverse* – qui impliquent elles-mêmes 2 déterminisations –, et 4 compositions. Au total, M&S construit 5 machines complexes et exécute 12 opérations coûteuses.

L'algorithme *Parse & Merge* ne calcule que 2 machines : une simple, REPLACE, et une complexe, le COMPLEMENT. Les opérations nécessaires sont 2 *Parse* – opération comparable à une intersection et réalisée à partir de petites machines (Λ et P) –, 1 *Merge* peu coûteux, 1 suppression des ϵ, 1 déterminisation et 1 minimisation. Au total, cet algorithme construit 1 machine simple, 1 machine complexe, exécute 3 opérations simples et 3 opérations coûteuses. Il demande donc moins de temps de calcul que M&S.

6. CONCLUSION

Nous avons présenté un nouvel algorithme de compilation de règles de réécriture sous la forme de transducteurs. Simple et peu coûteux en temps de calcul, cet algorithme se base sur le calcul du complément de la partie supérieure de la règle, dans lequel la règle elle-même est insérée à l'aide de 2 nouvelles opérations, *Parse* et *Merge*. L'algorithme a été étendu afin d'autoriser la définition de règles pondérées et/ou optionnelles.

RÉFÉRENCES

[1] C. Johnson. *Formal aspects of phonological description*. Mouton, The Hague, 1972.

[2] R. Kaplan and M. Kay. Regular Models of Phonological Rule Systems. *Computational Linguistics*, 20(3) :331–378, 1994.

[3] L. Karttunen. The replace operator. In *Meeting of the Association for Computational Linguistics*, pages 16–23, 1995.

[4] M. Mohri. Compact representations by finite-state transducers. In *Meeting of the Association for Computational Linguistics*, pages 204–209, 1994.

[5] M. Mohri, F. Pereira, and M. Riley. Weighted automata in text and speech processing. In *ECAI'96 Workshop*, pages 46–50, 1996.

[6] M. Mohri, F. Pereira, and M. Riley. Generic ϵ-Removal Algorithm for Weighted Automata. *Lecture Notes in Computer Science*, 2088 :230–242, 2001.

[7] M. Mohri and M. Riley. Weighted determinization and minimization for large vocabulary speech recognition. In *Proceedings of Eurospeech'97*, pages 131–134, 1997.

[8] M. Mohri and R. Sproat. An efficient compiler for weighted rewrite rules. In *Meeting of the Association for Computational Linguistics*, pages 231–238, 1996.

[9] F. Pereira and M. Riley. Speech Recognition by Composition of Weighted Finite Automata. In *Finite-State Language Processing*, pages 431–453, 1997.

[10] E. Roche and Y. Schabes, editors. *Finite-State Language Processing*. MIT Press, Cambridge, Massachusetts, 1997.

[11] K. Thompson. Regular expression search algorithm. *Communications of the ACM*, 11(6) :419–422, 1968.