Reconnaissance de motifs dynamiques par automates temporisés à mémoire

Clément Bertrand

université
PARIS-SACLAY

Reconnaissance de motifs dynamiques par automates temporisés à mémoire

Thèse de doctorat de l'Université Paris-Saclay

École doctorale n° 580, Sciences et technologies de l'information et de la communication (STIC)
Spécialité de doctorat : Informatique
Unité de recherche : Université Paris-Saclay, Univ Evry, IBISC, 91020, Évry-Courcouronnes, France.
Référent : Université d'Évry Val d'Essonne

Thèse présentée et soutenue à Évry, le 17 Décembre 2020, par

Clément BERTRAND

Composition du jury :

Franck POMMEREAU Professeur, Université Paris-Saclay	Président
Étienne ANDRÉ Professeur, Université de Lorraine	Rapporteur
Alexandre DURET-LUTZ Professeur, EPITA	Rapporteur
Emmanuel FILIOT Chargé de Recherche, Université Libre Belgique	Examinateur
Elisabeth PELZ Professeur, Université Paris-Est Créteil	Examinatrice
Tiphaine VIARD Maître de conférences, Telecom Paris	Examinatrice
Hanna KLAUDEL Professeur, Université Paris-Saclay	Directrice
Frédéric PESCHANSKI Maître de conférences, Sorbonne Université	Coencadrant

Thèse de doctorat

NNT : 2020UPASG034

Table des matières

Remerciements

Je souhaite tout d'abord exprimer toute ma gratitude à Frédéric PESCHANSKI et Hanna KLAUDEL pour m'avoir offert l'opportunité de travailler avec eux et m'avoir initié à la recherche. Vous m'avez énormément appris et je vous suis très reconnaissant pour toute l'aide que vous m'avez apportée durant ces quelques années de collaboration. Merci Frédéric pour ces discussions énergiques qui auront fait progresser nos travaux et m'auront bien amusé par la même occasion. Merci Hanna pour tout le soutien dont tu m'as fait part et surtout pour les bien trop nombreuses relectures que tu as faite et qui auront permis à ce manuscrit d'être ce qu'il est aujourd'hui.

Je tiens à remercier les membres du jury Étienne ANDRÉ, Alexandre DURET-LUTZ, Emmanuel FILIOT, Elisabeth PELZ, Franck POMMEREAU et Tiphaine VIARD. En particulier, les relectures particulièrement minutieuses d'Étienne et Alexandre et pour tous leurs commentaires pertinents qui m'ont aidé à améliorer ma thèse.

Merci bien sûr à Matthieu LATAPY qui, avec Frédéric, a initié le sujet que j'ai étudié durant ce doctorat et m'a permis de le découvrir durant le stage qui l'a précédé.

Durant ces quelques années j'ai eu la chance de pouvoir travailler simultanément dans deux laboratoires et dans deux équipes très chaleureuses qui m'ont accueilli et soutenu : COSMO à IBISC et APR au LIP6. Avant d'être mes collègues, les membres du LIP6 sont ceux qui m'ont fait découvrir et enseigner l'informatique durant mes premières années à l'université et je vous en suis particulièrement reconnaissant. C'est ensuite mes collègues d'IBISC qui m'ont fait découvrir l'enseignement de l'informatique et le monde de la recherche académique, et je vous remercie de m'en avoir offert l'opportunité.

À tous mes collègues doctorants et postdocs, les anciens qui m'ont vu arriver et accueilli parmi eux, et aussi les nouveaux qui nous ont rejoints après, je vous remercie pour tout ce bon temps que l'on aura passé ensemble. C'est grâce à l'entraide et les moments de détente tous ensemble que le doctorat est au final une période plaisante dont on garde de nombreux et agréables souvenirs (sauf pour la rédaction).

Je tiens aussi à remercier tou.te.s les secrétaires, gestionnaires, ingénieurs et régisseurs qui permettent au laboratoire de fonctionner et sans qui rien ne serait possible.

Enfin, pour m'avoir apporté une aide inconditionnelle, je souhaite remercier ma famille qui a toujours été présente pour moi et qui m'aura soutenu dans bien des occasions. Encore merci à mes parents qui m'ont donné la force de faire ce travail et qui m'ont toujours donné les meilleurs conseils pour progresser. Je remercie aussi mes amis et mes proches, et en particulier ma petite amie Jung qui aura toujours fait son maximum pour moi et qui m'a poussé à donner le meilleur de moi-même jusqu'au bout.

Chapitre 1

Introduction

Un enjeu technologique majeur est de pouvoir traiter une quantité de données toujours plus importante. Cela concerne des flux d'informations de différents types, par exemple : les réseaux de communications, comme internet, les réseaux sociaux, les transactions boursières ou encore les réseaux biologiques. De nombreux enjeux scientifiques et commerciaux tournent autour de leur étude afin d'y détecter certaines propriétés ou d'en prédire l'évolution. Cette problématique correspond à l'étude des systèmes complexes.

Les systèmes complexes sont composés d'un ensemble d'entités qui interagissent entre elles au cours du temps et dont les comportements sont régis par des règles souvent non linéaires telles que des boucles de rétroaction ou de causalité circulaire. De manière similaire à la théorie du chaos, il est théoriquement impossible de prédire l'évolution du système sans le simuler ou l'exécuter. Une manière classique de représenter l'évolution des systèmes complexes est par de très grands graphes dynamiques composés de motifs non triviaux [62, 88, 39]. Les arêtes du graphe représentent une relation d'interaction ou de connexion entre les entités (nœuds) du système. Par exemple, une arête représente : une relation d'amitié dans le cas des réseaux sociaux, une transaction commerciale dans le cadre de la bourse, ou encore une communication dans le cadre de la télécommunication. L'une des propriétés importantes de ces systèmes est leur dynamisme. Ce sont des systèmes qui évoluent au cours du temps, des connexions se forment et se rompent, des utilisateurs rejoignent le réseau ou s'en déconnectent...

Dans cette thèse nous étudions la problématique de supervision de tels systèmes. Cela signifie que l'on souhaite analyser les flux de données, durant l'exécution du système afin de vérifier dynamiquement la satisfaction de propriétés. Ce qu'on appelle ici une propriété correspond à un scénario de communications, que l'on peut se représenter par une sorte de *motif* ou sous-graphe [58, 72].

L'une des manières classiques d'étudier l'évolution d'un système complexe est de l'observer sur une succession de fenêtres de temps dans lesquelles on représente

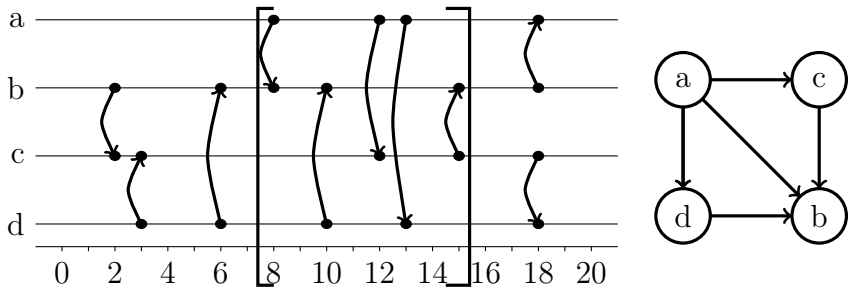


FIGURE 1.1 – Un flot de liens (gauche) et sa projection sur l'intervalle $[8, 15]$ (droite).

par un graphe statique les interactions qui y ont lieu [44, 22, 48]. L'évolution du système complexe devient une séquence ordonnée de graphes représentant l'ensemble des arêtes qui apparaissent dans un intervalle de temps donné. Un motif peut alors être représenté par une séquence de sous-graphes statiques. Détecter un motif consiste donc à trouver une correspondance entre les deux séquences. De manière naïve cela consiste à une résolution répétée du problème de l'isomorphisme de sous-graphes. Cela est cependant irréalisable dans la plupart des situations, principalement pour deux raisons :

1. les systèmes complexes sont issus de situations réelles et sont des graphes extrêmement grands ;
2. et le problème de l'isomorphisme de sous-graphes est NP-complet.

Un défaut supplémentaire de cette représentation est la perte d'information temporelle sur les arêtes qui apparaissent dans une même fenêtre de temps [60].

Le modèle des *flots de liens* [62] a été proposé pour modéliser les systèmes d'interactions complexes d'une manière simple et sans perte d'information. Un flot de liens est une séquence de triplets (t, u, v) horodatés, où chacun d'entre eux indique qu'une interaction (par exemple, un échange de messages) a lieu entre u et v à l'instant t. La Figure 1.1 (gauche) présente un exemple de flot de liens qui modélise les interactions entre les nœuds a, b, c et d. Chaque flèche est un lien entre deux nœuds, par exemple, à l'instant $t = 6$ on peut observer un lien du nœud d vers le b, ce qui correspond à $(6, d, b)$ dans le flot. La droite de la Figure représente le graphe statique composé des liens contenus dans la fenêtre de temps de l'instant 15 à 21. Cet intervalle est représenté par le cadre de la partie gauche de la Figure. Il existe plusieurs variantes du modèle des flots de liens qui ajoutent plus ou moins d'informations dans les triplets. Par exemple, il est possible de simuler les signaux booléens à l'aide des flots de liens, soit en indiquant dans le lien la durée de l'interaction, soit en ajoutant des éléments dénotant le début et la fin d'un signal.

Nous avons pour objectif de développer des techniques d'analyse qui peuvent s'exécuter directement sur le flot de liens, sans avoir besoin de créer la séquence de graphes sous-jacente. Les motifs *intéressants* dans les flots de liens sont composés à la fois d'un aspect structurel et aussi d'un aspect temporel, ce qui entraîne des difficultés quant à la description de tels motifs mais aussi au développement d'algorithmes de reconnaissance. Les principaux algorithmes de reconnaissance que l'on trouve dans la littérature ont été développés pour des motifs précis. Prenons l'exemple du motif du triangle, c'est-à-dire un sous-graphe complet composé de trois nœuds. Dans des flots de liens issus du monde réel, détecter des triangles efficacement n'est en réalité pas trivial, et l'on retrouve plusieurs algorithmes optimisés pour la recherche de ce motif en particulier [58, 72, 61]. Dans le domaine de la sécurité des réseaux de télécommunication ce motif est intéressant car il représente potentiellement une « attaque » : deux nœuds peuvent être identifiés comme « attaquants » s'organisent pour « attaquer »un troisième nœud identifié comme la « cible ». Une telle situation peut être observée sur la Figure 1.1 (droite), où les *attaquants* a et d s'en prennent à la *cible* b.

Ces travaux sont spécifiques car ils ne caractérisent qu'un seul type de structure. Dans cette thèse, notre motivation est de développer une approche suffisamment générique pour traiter non seulement des triangles, mais aussi d'autres sortes de motifs : des polygones dirigés, des chemins, des clignotements (*e.g.* un lien apparaissant périodiquement)... Nous gardons aussi à l'esprit que l'algorithme de reconnaissance doit être utilisable en pratique, et par conséquent nous devons réfléchir à ses performances. Notre point de départ est la théorie des *automates finis* (AF) et les *expressions régulières* (regex, pour *regular expressions*). En effet, lorsque l'on ignore les informations liées au temps, un flot de liens est similaire à un mot fini, chaque symbole étant un lien dirigé (un couple de nœuds). Par exemple, dans la fenêtre de temps $(8, 15)$ de la Figure 1.1 nous observons le « mot » suivant :

$$(a, b)(d, b)(a, c)(a, d)(c, b)$$

En se basant sur cette représentation, nous pouvons utiliser les AF comme un outil de reconnaissance et les regex comme un langage de spécification de haut niveau. L'exemple précédant du triangle peut être représenté avec la regex suivante :

$$(@ \to @)^* \otimes (((a \to d) + (d \to a)) \cdot ((a \to b) \otimes (d \to b)))$$

Cette expression utilise les opérateurs des expressions régulières classiques, telle que la concaténation $\cdot$, la disjonction $+$, l'étoile de Kleene * et l'opérateur de mélange $\otimes$. Le symbole @ est utilisé pour représenter un nœud quelconque, de même $(@ \to @)$ représente un lien quelconque. À partir d'une telle spécification, il est facile de construire un automate fini pour reconnaître efficacement les triangles dans un flot de liens sans informations de temps.

De fait, les langages rationnels ne permettent pas de capturer les propriétés temporelles des flots de liens. Toujours dans la théorie des automates, avec l'information de temps, les flots de liens sont similaires aux langages temporisés reconnus par les automates temporisés [2]. Étonnamment, les travaux de recherche sur la problématique de la reconnaissance de propriétés temporelles sont relativement récents, et ce malgré le succès que connaissent les automates temporisés. Un important point de départ est le formalisme des *expressions régulières temporisées* [7]. Le principe de base est de considérer les mots lus en entrée, ici les flots de liens, comme des *séquences d'événements temporisés* : une succession de symboles ou de délais correspondant au passage du temps. Ci-dessous est représentée la fenêtre de temps (8,15) de la Figure 1.1 sous forme de séquence d'événements temporisés :

$$(a, b)2(d, b)2(a, c)1(a, d)2(c, b)$$

Une expression régulière temporisée représentant le motif du triangle peut être représentée de la manière suivante :

$$\big(((a \to d) + (d \to a)) \cdot \langle (a \to b) \otimes (d \to b) \rangle_{[0,1]}\big) \otimes (@ \to @)^*$$

L'opérateur de délais $\langle S \rangle_{[x,y]}$ indique que le sous motif S doit avoir une durée totale comprise dans l'intervalle $[x, y]$. Pour le motif du triangle, cela signifie que les nœuds a et d sont considérés comme « attaquants » la cible b seulement s'ils sont liés simultanément à b dans un intervalle de 1 seconde.

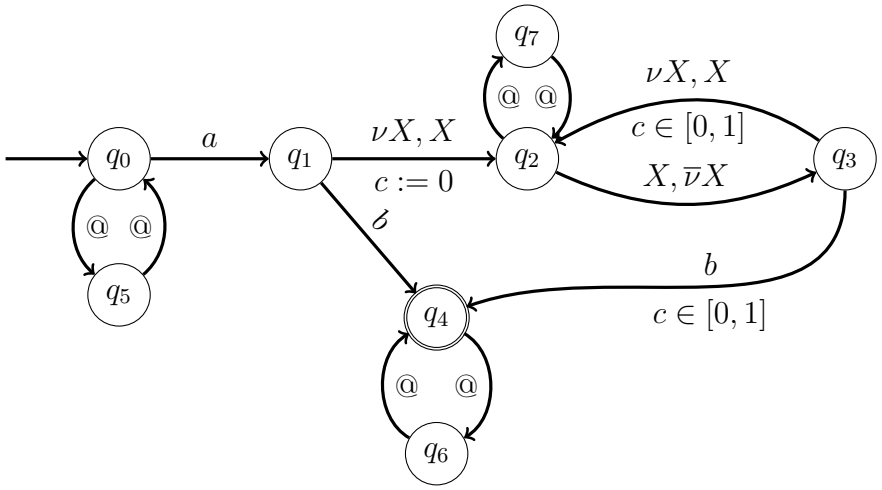
Un autre aspect fondamental des motifs dans les flots de liens que nous souhaitons capturer est la *connaissance partielle*. Dans les automates finis comme temporisés, les symboles utilisés font partie d'un alphabet fini fixé à l'avance. Dans les flots de liens, cela signifierait que les nœuds composant le graphe devraient être connus à l'avance, ce qui est une supposition trop forte. Dans les cas des scénarios d'attaques, par exemple, nous devons considérer des systèmes ouverts : on peut connaître à l'avance la cible, mais les deux attaquants sont inconnus.

Le genre de motif que nous souhaitons représenter est par exemple :

$$\big((\sharp X \to \sharp Y) \cdot \langle (X \to b) \otimes (Y \to b) \rangle_{[0,1]}\big) \otimes (@ \to @)^*$$

Dans ce motif, X et Y sont des variables qui représentent les nœuds inconnus, correspondant aux deux « attaquants ». La construction d'affectation $\sharp X$ signifie que le symbole lu en entré (ici un nœud) est associé à X et doit être frais, c'est-à-dire qu'il n'a jamais été lu précédemment. Lorsqu'un nœud correspond, il est associé à X et gardé en mémoire. L'opérateur de suppression $X!$ efface, en quelque sorte, la mémoire associée à X.

Le sous-motif $(\sharp X \to \sharp Y)$ décrit un lien entre deux nœuds frais. On remarque que comme Y est reconnu après X, les contraintes de fraîcheur imposent que la



$$(@ \to @)^* \otimes \Big((a \to b) + a \to \sharp X \cdot \langle (X! \to \sharp X)^* \cdot (X! \to b) \rangle_{[0,1]}\Big)$$

FIGURE 1.2 – Exemple d'automate et l'expression correspondante.

valeur représentée par Y soit différente de celle de X. Pour reconnaître la sous-expression $(X \to b)$, le symbole lu doit être un lien du nœud déjà associé à X en mémoire vers le nœud b. C'est potentiellement une attaque vers la cible b.

Pour prendre en compte une telle reconnaissance dynamique, il est nécessaire d'utiliser un alphabet infini (dénombrable) de symboles inconnus. Cela a été étudié dans le cadre des *langages de données* et dans les *automates à mémoire* [55, 46, 31]. Dans ce document nous définissons notre propre modèle, développé spécifiquement pour notre problématique : les *automates à couches mémoire* (ACM). La particularité de ce modèle est la subdivision de la mémoire en différentes couches indépendantes. Chaque couche contient des variables pouvant chacune stocker un nombre quelconque de lettres. Cependant, les variables d'une même couche ne peuvent pas stocker les mêmes lettres. Ce mécanisme permet de représenter en parallèle différentes propriétés dans la mémoire, et ainsi d'en composer des plus complexes sur les langages de données.

Pour traiter les motifs sur les flots de liens, nous introduisons un nouveau formalisme de reconnaissance : les *automates temporisés à couches mémoire* (ATCM). Ce modèle est composé à la fois des horloges et contraintes temporelles des automates temporisés, et des mécanismes de gestion de mémoire des ACM. Pour l'illustrer, la Figure 1.2 représente un automate qui détecte dans un flot de liens tous les chemins allant d'un nœud a à un nœud b dans un délai d'une seconde. Nous supposons que l'automate est défini pour l'alphabet $\Sigma = \{a, b\}$, c'est-à-dire,

seuls les nœuds a et b sont initialement connus. Les symboles νXX et $X, \overline{\nu}X$ sont les variantes pour automates des opérateurs $\sharp X$ et $X!$ présentés précédemment. Un exemple de mot reconnu par l'automate est :

$$(a, y)0.1(y, z)0.3(y, b)$$

Dans la position initiale q_0, le symbole connu a est consommé par la transition allant vers la position q_1. Le symbole inconnu y est sauvegardé en mémoire dans la variable X lors de la transition menant à la position q_2. Cela fonctionne uniquement car y est frais, c'est-à-dire qu'il n'a jamais été lu auparavant. Le délai de 0.1 seconde est consommé à la position q_2 ce qui incrémente la valeur de l'horloge c à 0.1. Le prochain symbole lu y peut soit amener à la position q_3 (car il a été précédemment associé à X) soit à q_7 (car @ accepte tous les symboles). Le principe de reconnaissance n'est pas déterministe donc deux cas sont possibles et vont être essayés :

— Si la transition $q_2 \xrightarrow{X,\overline{\nu}X} q_3$ a été choisie, plus aucun symbole n'est associé à X dans q_3. Le symbole lu suivant est le symbole inconnu z. À partir de la position q_3, seule la transition $q_3 \xrightarrow[c\in[0,1]]{\nu X,X} q_2$ est franchissable. À la position q_2, la variable X sera associée à z. Cependant, ce chemin est condamné car le prochain (et dernier) lien n'a pas pour origine z. Ainsi, à la fin de la séquence le chemin mène à la position q_2 qui n'est pas finale.

— Si la transition $q_2 \xrightarrow{@} q_7$ a été choisie, alors la valeur associée à X n'a pas été oubliée et le symbole lu z mène à la position q_2 via la transition $q_7 \xrightarrow{@} q_2$. La valeur 0.3 incrémente la valeur de l'horloge à $c = 0.4$. Le prochain symbole lu y peut de nouveau mener soit à q_7 soit à q_3 comme précédemment. Dans la position q_3 le symbole b qui est ensuite lu permet d'emprunter la transition $q_3 \xrightarrow[c\in[0,1]]{b} q_4$, qui mène à la position finale q_4 (car $b \in \Sigma$).

Une position finale est bien atteinte car la valeur de l'horloge $c = 0.4$ est inférieure à 1 seconde et respecte bien la contrainte de temps. D'un autre côté, si le second délai n'était pas 0.3 mais par exemple 1.0 alors le flot de liens n'aurait pas été accepté par l'automate car la contrainte de temps de la position q_2 aurait été brisée.

En suivant ce principe de reconnaissance, nous avons conçu un algorithme à la volée permettant de vérifier si un mot appartient au langage d'un ATCM. Ce modèle d'automate est définit comme un formalisme de reconnaissance pour les motifs exprimés sous formes d'ETCM. Nous présentons ainsi un algorithme permettant la traduction d'ETCM en un ATCM équivalent. Notre dernier objectif est alors le développement d'un outil permettant la reconnaissance à la volée des motifs décrite sous-forme d'ETCM dans des flux de données réels. Cela nécessite l'implémentation de l'algorithme traduction et de reconnaissance pour le modèle

d'automate que l'on a définit.

Dans cette thèse, nous défendons les contributions suivantes :

— La définition des *expressions temporisées à couches mémoire* (ETCM), qui sont une extension des expressions régulières permettant la spécification des langages temporisés sur alphabet infini. C'est à notre connaissance le premier langage de spécification permettant d'exprimer simultanément des propriétés temporisées et des propriétés sur un alphabet infini. De plus, la partie non-temporisée des ETCM est l'un des rares langages de spécification sur alphabet infini possédant un modèle d'automate sous-jacent. Le segment du langage concerné s'avère être plus expressif que les quelques langages similaires ayant des modèles d'automates. Aussi, nous présentons comment les ETCM permettent de modéliser formellement des motifs sur les flots de liens.

— Pour le problème de la reconnaissance dans les ETCM, nous avons développé les *automates temporisés à couches mémoire* (ATCM), un nouveau formalisme de reconnaissance inspiré des FMA [55] et des automates temporisés [2]. Notre contribution concerne l'introduction de la notion de couche mémoire. Ces couches permettent d'exprimer, de manière modulaire, des propriétés complexes sur un alphabet infini. Nous prouvons que la classe des langages reconnus par les ATCM est équivalente à la classe des langages exprimables par les ETCM, de manière similaire au théorème de Kleene pour les langages finis. Nous comparons la version non-temporisée des ATCM aux autres modèles d'automates à mémoire de la littérature afin d'en déterminer son positionnement précis. Cela contribue à la définition de plusieurs propriétés du modèle, comme la décidabilité de problèmes classiques et des clôtures pour les opérations rationnelles de l'ensemble des langages reconnaissables.

— L'implémentation des modèles et de l'algorithme dans un prototype. Il permet d'analyser à la volée un flux de données temporisées dans le cadre de la vérification dynamique de propriété spécifiée sous forme d'ETCM. Il est utilisé pour effectuer des expérimentations de détection de motifs sur des flots de liens tirés de jeux de données réels et présente des performances encourageantes.

En plus de cela, nous présentons quelques contributions techniques mineures avec :

— Le calcul des tailles des différentes constructions sur les ACM et leur comparaison à celles des constructions proposées dans la littérature. Cela nous permet d'évaluer la qualité des automates générés depuis les ETCM.

— La conception d'un algorithme de reconnaissance pour les ATCM. Il contient une contribution dans le domaine des automates temporisés avec l'adaptation de la structure des *Difference Bound Matrix* pour le calcul des zones temporelles lors de la reconnaissance.

Cette thèse débute, dans le Chapitre 2, par une étude de l'état de l'art et une présentation des différentes disciplines abordées dans cette thèse. Dans le Chapitre 3, nous définissons le langage des expressions temporisées à couches mémoire ainsi que sa sémantique. Ensuite, dans le Chapitre 4, est présenté le modèle des automates temporisés à couches mémoire ainsi que ses propriétés. Nous y prouvons un théorème, similaire au théorème de Kleene, démontrant l'équivalence entre les langages reconnaissables par les automates et ceux exprimables par les expressions temporisées à couches mémoire. Enfin, dans le Chapitre 5, nous présentons l'algorithme de reconnaissance que nous avons développé et implémenté dans un prototype. Ce dernier est utilisé dans des expérimentations sur la vérification dynamique de motifs dans les flots de liens. Pour finir, nous présentons dans le Chapitre 6 les conclusions de ces travaux et les perspectives et possibles continuations.

Chapitre 2

État de l'art

Comme à l'accoutumée, nous débutons par une présentation de travaux présents dans la littérature sur les différentes problématiques abordées. Ce chapitre commence par une présentation du domaine des systèmes complexes. C'est un sujet très vaste et étudié via de nombreuses approches : apprentissage, statistique, algorithmique... Aussi, nous prenons le parti de présenter les quelques modèles classiques représentant ces systèmes ainsi que les travaux relatifs à la spécification et la détection de motifs.

Notre approche est à l'intersection de deux disciplines : les langages temporisés et les langages de données. La seconde section présente donc les modèles permettant la spécification de propriétés temporelles et temporisées. La troisième et dernière section est consacrée à la présentation des différents formalismes de spécification de langages de données, et principalement les automates à mémoire.

2.1 La reconnaissance de motifs dans les systèmes complexes

Dans cette thèse, nous représentons l'évolution d'un système complexe comme un (très grand) graphe dynamique. C'est une approche courante qui a un succès croissant ces dernières années, dû à l'accroissement rapide de la quantité de données collectées. En effet, les historiques de connexions dans les réseaux, les échanges dans les réseaux sociaux, les réseaux de télécommunication sont autant de disciplines en expansion dont les données s'apparentent naturellement à des graphes dynamiques [54, 22, 48]. Bien d'autres domaines s'apparentent aux systèmes complexes, par exemple : les réseaux biologiques, les neurosciences, la finance...

Il existe dans la littérature plusieurs formalismes pour représenter les graphes dynamiques. Le plus classique est la représentation sous forme de séquence de graphes statiques [44, 22, 48]. Chaque élément de la séquence représente alors

un cliché (*snapshot*) de l'état du graphe dynamique dans une fenêtre de temps. Cela a l'avantage de permettre la réutilisation partielle des propriétés et méthodes développées pour la théorie des graphes « classiques ». Cependant, cette représentation a le désavantage de perdre une partie de l'information liée à la temporalité des liens apparaissant dans la même fenêtre de temps. De plus le choix de la durée des fenêtres de temps n'est pas trivial [60, 74] car c'est un compromis entre une durée suffisamment longue pour que les clichés soient pertinents et une durée suffisamment courte pour ne pas perdre la temporalité des arêtes.

Plusieurs autres représentations des graphes dynamiques sont similaires à des graphes statiques afin de pouvoir préserver les propriétés de la théorie des graphes. Des travaux les représentent comme des graphes statiques dont les arêtes sont annotées par les instants où elles sont présentes [29, 68]. Ou encore, dans les travaux [90, 77] l'information de temps est représentée par les nœuds du graphe. Chaque nœud du graphe dynamique est dupliqué pour chaque instant où une arête le relie à un autre nœud, et toutes les instances d'un même noeud sont reliées entre elles par des arêtes.

Dans cette thèse, notre attention se porte principalement sur le formalisme des flots de liens [62]. Les graphes dynamiques y sont représentés comme des séquences ordonnées de triplets (liens) (t_i, u_i, v_i) représentant l'existence d'une arête du nœud u_i au nœud v_i à l'instant t_i. Ce modèle met en avant l'aspect temporel des graphes dynamiques et se détache en partie de la théorie classique des graphes. La proximité de ce formalisme avec les données brutes, issues des historiques de communications, permet une application directe des méthodes développées sur les flots de liens.

L'étude de la topologie des systèmes complexes a permis la définition d'un ensemble de composantes primaires communes à de nombreuses propriétés et applications [62]. Parmi les composantes les plus représentatives dans le domaine, on distingue les cliques [84, 44]. Ce sont des sous-graphes fortement connexes qui s'apparentent, par exemple, aux communautés dans les réseaux sociaux. Ils permettent aussi de représenter les graphes de manière abstraite en regroupant les entités appartenant à une même communauté [44]. On retrouve aussi la notion de chemin, qui est une séquence d'arêtes reliant deux nœuds à travers le graphe. Dans les graphes dynamiques, les arêtes du chemin doivent apparaître de manière causale et permettent la représentation de communications [91, 78]. D'autres notions centrales sont les *clusters*, les notions de densité et de voisinage pour représenter les nœuds les plus significatifs d'un graphe, etc.

C'est en partie pour détecter ces composantes que les premiers travaux de recherche de motifs dans les flots de liens ont été développés. Plusieurs travaux traitent du comptage approximatif de motifs dans les flots de liens [61, 49]. Ces estimations permettent de détecter des composantes des graphes. Par exemple, les triangles sont les plus petites cliques et la présence d'une grande quantité de tri-

angles permet de détecter la présence d'importantes cliques. C'est à partir des travaux [72, 58] que la notion de motif dans les flots de liens est formellement définie. Plusieurs algorithmes précis de comptage de motifs y sont proposés. Cependant, la complexité du problème pousse au développement d'algorithmes optimisés spécifiques à certains motifs (l'étoile, le triangle...) [72, 65] qui permettent un comptage efficace. De telles méthodes sont cependant difficiles à étendre à toutes sortes de motifs, où les optimisations et heuristiques proposées peuvent ne plus s'appliquer. Le problème du comptage de motif est similaire au problème de l'isomorphisme de sous-graphe qui est NP-complet [30] pour les graphes statiques. Ainsi peu d'algorithmes efficaces existent effectivement pour la détection de tous types de motifs [72, 66], et ils souffrent en général de performance limitée comparée aux algorithmes *ad hoc*. Dans ces travaux, aucun formalisme ne permet la spécification de motifs temporels de manière abstraite. Ils se basent principalement sur la définition des motifs donnée dans [72]. Les motifs y sont représentés par une séquence de liens apparaissant dans une fenêtre de temps δ, donc comme un sous-graphe précis de durée δ. Cela permet uniquement de spécifier un motif de taille et de structure fixe, et sans possibilité d'exprimer des contraintes de temps internes au motif.

C'est le constat de l'absence d'un formalisme générique de spécification de motifs et les similitudes entre les flots de liens et les langages temporisés et les langages de données qui a initié nos travaux et les motive. Nous avons donc souhaité créer un langage de spécification permettant de décrire de manière abstraite les motifs temporels, mais aussi de représenter une classe de motifs respectant tous une spécification commune. Cela permet par exemple de spécifier un motif représentant l'ensemble des chemins allant d'un nœud a à un nœud b. Les sections suivantes présentent la synthèse des travaux sur la spécification de langages temporisés et de langages de données qui nous servirons de base pour la création de notre propre formalisme. Nous présentons également les travaux de la littérature qui étudient la problématique des formalismes et algorithmes de reconnaissance de motifs.

2.2 Les motifs temporisés

Les systèmes temps réel et les propriétés liées au temps ont été largement étudiés dans la littérature. En effet, dans les systèmes embarqués, concurrents ou distribués, le passage du temps est une donnée importante à prendre en compte. Les motifs que l'on souhaite représenter sont dynamiques, ils représentent une évolution spécifique du graphe au cours du temps. Il y a deux notions importantes que l'on va souhaiter représenter dans un tel motif : des contraintes temporelles et des contraintes temporisées. Les temporelles représentent la causalité des interactions, ce qui fait qu'un événement est potentiellement causé par un autre événement. Cela indique que l'ordre des occurrences des événements peut être important.

C'est une notion de temps implicite, relative aux événements. Les contraintes temporisées étudient le temps comme une information explicite, discrète ou continue. Elles prennent en compte le temps réel, énumèrent les instants, et permettent de contraindre les durées.

2.2.1 Les langages de spécification

On peut distinguer deux types d'approches pour exprimer des propriétés temporelles et/ou temporisées. Il y a d'abord les *logiques temporelles*, une branche des *logiques modales*, qui permettent de décrire des propriétés sur le temps. De façon complémentaire, il y a des approches inspirées de la théorie des langages, les expressions régulières et les automates finis.

Commençons par les logiques temporelles, avec la Logique Temporelle Linéaire (LTL) [9], une logique monadique du second ordre qui permet de définir des propriétés sur la causalité des événements. Ces propriétés décrivent les présences successives ou simultanées de différents prédicats booléens apparaissant sur un temps discret. Les formules LTL sont aussi expressives que les automates de Büchi. Elles ne permettent pas d'exprimer des propriétés temporisées, il n'est, par exemple, pas possible d'exprimer des délais ou durées.

La logique temporelle métrique (*Metric Temporal Logic*, MTL) [59], est une généralisation de LTL permettant de préciser des contraintes sur les délais et durées. Tous les opérateurs LTL y sont généralisés avec la possibilité de préciser un intervalle de réels positifs précisant la période de temps concernée. Les formules MTL permettent de décrire des propriétés temporelles et temporisées sur du temps réel. Une version contrainte de cette logique, MiTL (*Metric interval Temporal Logic*) [67], est équivalente en expressivité aux automates temporisés.

Enfin la logique temporelle pour les signaux (*Signal Temporal Logic*, STL) [38], étend la logique MTL avec la possibilité de décrire des prédicats non booléens. Dans LTL et MTL les événements sont des prédicats dont la valeur peut changer au cours du temps. Alors que STL permet de décrire des propriétés sur des signaux, qui sont des valeurs réelles évoluant de manière continue au cours du temps.

De façon complémentaire aux logiques temporelles, il existe dans la théorie des langages des formalismes permettant d'exprimer des contraintes de temps inspirés des expressions régulières. Les *expressions régulières temporisées* (*Timed Regular Expressions*, TRE) [7] sont une extension des expressions régulières permettant d'exprimer des langages temporisés. Les mots temporisés sont des séquences ordonnées d'événements, où un événement est un couple composé d'une lettre de l'alphabet fini Σ et de l'instant $x \in \mathbb{Q}_+$ où l'événement a lieu. Par exemple, le mot

$\binom{e_1}{t_1}\binom{e_2}{t_2}\binom{e_3}{t_3}$, est composé de trois événements e_1, e_2 et e_3 apparaissant respectivement aux instants t_1, t_2 et t_3 où $0 \leq t_1 \leq t_2 \leq t_3$.

Les TRE introduisent principalement un opérateur de délai, noté $\langle S \rangle_I$, représentant le langage des mots de l'expression S qui ont une durée incluse dans l'intervalle I. Ce modèle a été conçu comme un langage de spécification pour les automates temporisés. Cependant, pour que les TRE puissent représenter exactement l'ensemble des langages reconnaissables par ces automates, il est nécessaire que l'opération d'intersection y soit primitive et ils doivent définir un opérateur de renommage peu pratique [51].

Même si l'opérateur de délai offre un niveau d'abstraction agréable pour exprimer des contraintes de temps, les TRE ont le désavantage de nécessiter des opérateurs de renommage et d'intersection primitifs pour exprimer des contraintes de temps *croisées*. Ainsi, une variante des TRE a été développée afin d'éliminer ces opérateurs de la syntaxe : les expressions régulières temporisées équilibrées (*Balanced timed regular expressions*, BTRE) [8]. Ce langage d'expression généralise l'opérateur de délais avec l'ajout d'une couleur i en exposant des parenthèse : $\langle^i S\rangle^i_I$, qui permettent de spécifier des contraintes de temps croisées. La grammaire de ces expressions n'est alors plus inductive car elle autorise des expressions mal parenthésées, ce qui nécessite l'utilisation d'une sémantique en deux étapes afin de retrouver la correspondance entre les parenthèses et donc les contraintes de temps.

Certains travaux présentent des langages de spécifications dont la sémantique est définie directement avec des automates temporisés sous-jacents. Le langage des *constructions modulaires d'automates temporisés* (*Modular constructions of timed automata*) [25] étendent les expressions régulières avec l'ajout d'annotations permettant d'exprimer explicitement des contraintes et réinitialisations sur les horloges. La syntaxe de ces expressions a peu d'abstration par rapport aux automates temporisés car elle contient explicitement les horloges et gardes. Cependant, elle a les avantages, par rapport aux TRE, de ne pas introduire des opérateurs autres que les rationnels, et par rapport aux BTRE, d'avoir une syntaxe inductive.

Un autre langage dont la sémantique est définie sur les automates temporisés est l'*algèbre de processus temporisé* (*timed process algebra*) [37]. C'est un langage de spécification inspiré des algèbres de processus qui définit un ensemble de propriétés basiques, avec les constructions d'automates correspondantes, qui peuvent être composées ensemble pour exprimer des propriétés plus complexes. L'algèbre de processus temporisé est une alternative aux approches basées sur les logiques modales ou les expressions régulières. Elle offre un haut niveau d'abstraction sur les propriétés exprimables, mais au prix de la complétude du langage par rapport au modèle d'automate sous-jacent.

Pour la détection de motif nous allons nous intéresser plus particulièrement aux modèles possédant un formalisme de reconnaissance.

2.2.2 Les formalismes de reconnaissance

La théorie des automates englobe un ensemble de formalismes basés sur une notion de place/transition et dont le principe fondamental est celui de la reconnaissance, c'est-à-dire l'appartenance d'un mot à un langage. Parmi ces formalismes, les *automates temporisés*, et leurs dérivés, s'intéressent à la reconnaissance de mots temporisés, où chaque lettre est associée à l'instant où elle apparaît. Ce modèle est une extension des automates finis dotés d'un ensemble d'horloges mesurant le temps et dont les valeurs évoluent au même rythme. Les horloges peuvent être réinitialisées lors du franchissement d'une transition, c'est-à-dire qu'on peut modifier la valeur de l'horloge en lui affectant la valeur 0. Les transitions et places peuvent être annotées avec des contraintes sur les valeurs des horloges. Les contraintes des places sont similaires à des invariants interdisant au système d'être dans la place si son invariant n'est pas satisfait.

La version classique du modèle [2] d'automate temporisé permet de définir des contraintes de temps de la forme suivante :

$$\Gamma := c_1 \sim n \mid c_1 - c_2 \sim n \mid \Gamma \wedge \Gamma \mid \Gamma \vee \Gamma$$

où c_1 et c_2 sont des noms d'horloges, $n \in \mathbb{Q}$, et $\sim \in \{<, >, =, \neq, \leq, \geq\}$.

Les automates temporisés ont été largement étudiés dans le cadre de la modélisation et vérification de systèmes. Ainsi, dans la littérature, une grande partie des travaux traitent du problème du langage vide (*emptiness checking*), avec les techniques de *model checking*. Pour les automates temporisés, la complexité est PSPACE-complet via la construction de l'automate des régions [2] équivalant à l'automate initial. L'automate des régions est un automate fini non-déterministe permettant de représenter de manière finie l'ensemble infini des états atteignables d'un automate temporisé. De nombreux outils traitent de cette problématique, le plus connu étant l'outil UPPAAL [15].

Différentes variantes des automates temporisés ont été étudiées afin d'en accroître l'expressivité. Les *stopwatch automata* [1] sont une variante permettant de mettre en pause l'évolution des horloges. Plus généralement, les automates hybrides (*hybrid automata*) [50] permettent de définir les vitesses d'évolutions des horloges sous forme d'équation différentielle. Les automates temporisés avec mise à jour (*updatable Timed Automata*) [24] permettent d'effectuer des affectations aux horloges lors du franchissement des transitions. Ces différentes variantes permettent d'accroître l'expressivité du modèle, mais en accroissent aussi la complexité. Le *model checking* est indécidable pour ces trois variantes. Cependant, il redevient décidable pour certaines versions contraintes de ces modèles lorsqu'il est possible de générer l'automate des régions équivalent, mais le gain d'expressivité est alors peu significatif.

Les expressions régulières temporisées peuvent aussi être vues comme un formalisme de reconnaissance. En effet, les analyses syntaxiques avec dérivés (*parsing derivatives*) [27] sont des principes de reconnaissance développés directement sur les expressions régulières (non-temporisées). Lors de la reconnaissance d'un mot, les lettres sont lues une à une et l'expression est dérivée de manière à représenter le langage du reste du mot pouvant encore être reconnu, ou $\emptyset$ si la reconnaissance n'est pas possible. Ces principes ont été étendus à un sous-ensembles des expressions régulières temporisées pour la reconnaissance de langages temporisés pour les signaux [82].

2.2.3 L'algorithmique de reconnaissance

Les algorithmes de reconnaissance et de supervision de propriétés temporisées sont relativement récents par rapport aux travaux classiques sur les automates temporisés. En effet, les automates temporisés sont plutôt utilisés pour modéliser et vérifier des systèmes, ce qui s'apparente plutôt au *model checking*. Cette technique peut tout de même être utilisée pour résoudre le problème de la reconnaissance, en résolvant l'*emptiness checking* sur la conjonction de l'automate représentant le mot et celui du langage. Cependant, il existe la plupart du temps des algorithmes plus efficaces pour résoudre le problème de la reconnaissance.

Avant que des algorithmes de reconnaissance n'aient été développés pour les automates temporisés non déterministes, plusieurs travaux ont traité de la supervision de systèmes via la vérification dynamique de propriétés temporelles spécifiées par des formules MTL et STL [13, 14]. Les méthodes et algorithmes de supervision employés n'utilisent pas d'automates temporisés comme formalisme de reconnaissance, la génération d'un automate temporisé déterministe à partir d'une formule MTL étant trop coûteux [71]. Les premiers algorithmes utilisaient tout de même des automates finis alternants bidirectionnels (*alternating two-way automata*) pour vérifier la satisfaction des sous-formules non temporisées, d'algorithmes de programmation dynamiques pour vérifier la satisfaction des contraintes de temps [52]. L'utilisation de formules STL pour spécifier les propriétés permet une analyse qualitative, et d'obtenir une valeur de robustesse décrivant à quel point le système satisfait la formule [34]. La possibilité d'effectuer des analyses qualitatives, couplées aux analyses classiques, offre un moyen de contourner le fait que le *model checking* n'est pas décidable pour les formules STL [13].

Les premiers travaux traitant du problème de la reconnaissance lié aux automates temporisés ont été directement effectués sur les expressions régulières temporisées [81], privées de l'opérateur de renommage [51]. L'article [81] présente un algorithme statique (*offline*) analysant un ensemble de signaux booléens pour y trouver tous les intervalles de temps respectant une propriété décrite sous forme de TRE. C'est un algorithme de type « diviser pour régner » qui va récursivement

calculer les intervalles de temps où les sous-expressions sont satisfaites, puis en détermine tous les intervalles où l'expression est satisfaite.

L'analyse syntaxique avec dérivation a ensuite été développée sur les TRE [82] comme un algorithme de reconnaissance à la volée (*online*). C'est un algorithme permettant la reconnaissance de propriétés, pour les signaux booléens, exprimées sous forme de TRE (privées de l'opérateur de renommage). Au cours de l'analyse, l'expression est dérivée par rapport aux valeurs courantes des différents signaux. L'expression dérivée est composée de deux parties :

— une partie gauche représentant la partie déjà reconnue de l'expression et accumulant les informations liées aux temps afin de vérifier si les contraintes sont satisfaites,
— une partie droite étant une TRE représentant tous les suffixes qui peuvent être reconnus pour satisfaire la propriété.

Expérimentalement, l'algorithme à la volée est plus lent que l'algorithme statique par un facteur 100, mais utilise une quantité de mémoire bien moindre et constante par rapport à la taille des données traitées. Ces algorithmes sur les TRE ont été implémentés dans l'outil MONTRE [80].

Des algorithmes de reconnaissances ont été développés sur les automates temporisés, plus expressifs que les TRE privées de l'opérateur de renommage [85, 86]. Les premiers algorithmes de reconnaissances, statiques et à la volée, sur les automates temporisés ont été présentés dans [85]. Des algorithmes de reconnaissance « naïfs », un statique et son équivalent à la volée, sont présentés permettant d'effectuer la reconnaissance directement sur l'automate temporisé. Ils sont qualifiés de naïfs car ils recherchent les sous-mots reconnaissables par l'automate avec une stratégie « force brute ».

Les algorithmes de Boyer-Moore [26, 89] et de Franek–Jennings–Smyth [43] ont été adaptés au problème de la reconnaissance de langage temporisé [85, 86]. Ces deux algorithmes sont classiquement utilisés pour la reconnaissance de motifs dans les chaînes de caractères. L'idée est d'exploiter les séquences d'évolutions (*run*) de l'automate fini représentant le motif afin de détecter à l'avance des lettres que l'on peut ignorer pour accélérer la reconnaissance. Ils sont adaptés aux automates temporisés [2] via l'automate des régions qui permet de définir des *run*.

Une des contributions présentées dans cette thèse est un algorithme de reconnaissance pour les automates temporisés ne nécessitant pas la création de l'automate des régions. Cela consiste en l'adaptation des *Difference Bound Matrix* (DBM) [35] pour représenter les zones d'horloges dans un algorithme de reconnaissance. Les DBM sont une structure de données permettant de représenter les zones d'horloges et classiquement utilisés pour le *model checking* d'automates temporisés. Cependant, des travaux similaires, sur l'adaptation des DBM à la reconnaissance, ont été publiés simultanément à nos travaux [19, 10].

Le problème de reconnaissance de motifs temporisés paramétrés (*parametric timed pattern matching*) est une généralisation du problème de la reconnaissance de propriété temporisée. Dans ce problème, les propriétés temporisées peuvent être partiellement spécifiées à l'aide de variables non-initialisées, par exemple : indiquer que la durée du mot est strictement inférieure à x. L'objectif du problème consiste à trouver, pour un mot temporisé donné, l'ensemble de ses segments qui satisfont la propriété pour chaque valeur possible des variables. Le problème a été posé pour les propriétés exprimées sous forme d'automates temporisés paramétrés [3] dans [4] et des algorithmes statiques et à la volée [86, 4] ont été proposés pour le résoudre.

2.3 Les langages de données

La représentation de systèmes ouverts et dynamiques nécessite la capacité de spécifier et analyser des événements préalablement inconnus. La théorie des langages représente ce problème sous la forme de la spécification de langages de données. Ce sont des généralisations des langages rationnels définis sur un alphabet infini de lettres, appelées ici données. Ils utilisent souvent deux alphabets à la fois, un fini représentant des lettres connues à l'avance, et un infini représentant celles qui sont inconnues.

Dans la littérature, il existe deux représentations populaires des langages de données. La première est une généralisation des mots temporisés [23]. Les langages sont systématiquement définis sur l'alphabet fini d'événements Σ et l'alphabet infini de données $\mathcal{U}$. Un mot du langage est de la forme $(\Sigma \times \mathcal{U})^*$, donc une séquence de paires associant un événement à une donnée. Par exemple $\binom{e_1}{u_1}\binom{e_2}{u_2}\binom{e_3}{u_3}$ est un tel mot, où $e_1, e_2, e_3 \in \Sigma$ et $u_1, u_2, u_3 \in \mathcal{U}$.

La seconde représentation est une généralisation des langages rationnels définis sur un alphabet infini de lettres $\mathcal{U}$. Un langage de données est alors un sous ensemble de $\mathcal{U}^*$, c'est-à-dire un ensemble de séquence de lettres issues de l'alphabet infini. Il est courant de définir un alphabet fini $\Sigma \subset \mathcal{U}$ pour représenter des lettres connues à l'avance.

2.3.1 Les automates à mémoire

De nombreux modèles d'automates ont été développés pour spécifier des langages de données, communément appelés les *automates à mémoire*. Ces modèles ont été développés dans l'optique de rechercher un juste milieu entre expressivité et complexité. Dans la littérature, deux visions des automates à mémoire ont été développées : l'une consiste à créer des modèles équivalents à différentes sous-classes de logiques du premier et second ordre sur les langages de données [70], la seconde est le développement de modèles d'automates dérivés des automates finis. Nous nous

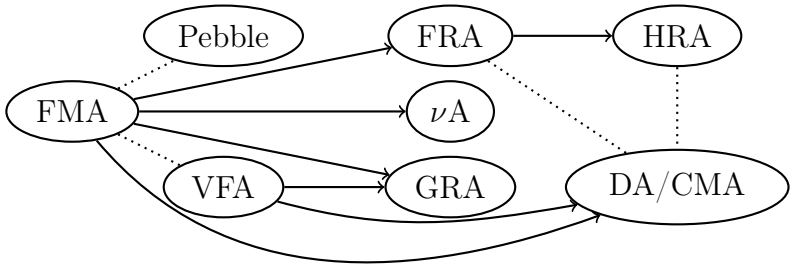


FIGURE 2.1 – Comparaison d'expressivité des automates à mémoire. Une flèche indique l'inclusion stricte et des pointillés indiquent que l'expressivité est incomparable.

sommes principalement intéressés à cette seconde catégorie car elle aboutit à des modèles d'automates relativement plus simples et qui permettent le développement de langages de spécification similaires aux expressions régulières. Le diagramme de la Figure 2.1 représente les relations d'inclusions entre les ensembles de langages reconnaissables par les différents classes d'automates à mémoire que nous avons étudiées.

Automates à registres

Le modèle le plus ancien permettant de représenter les langages de données sont les automates à registres (*Register Automata* ou *Finite Memory Automata*, FMA) [55]. Ce modèle est une variante des automates finis disposant d'un ensemble fini de registres $\mathcal{R}$ pouvant chacun stocker une lettre de l'alphabet infini $\mathcal{U}$. Ils sont composés d'un ensemble fini de places Q et de transitions Δ ainsi qu'une place initiale $q_0 \in Q$ et de places finales $F \subseteq Q$. Chaque transition $\delta \in \Delta$ est annotée avec l'identifiant d'un registre $r \in \mathcal{R}$ qui sera consulté pour autoriser le franchissement de δ. De même, chaque place $q \in Q$ de l'automate est associée à un sous-ensemble de $\mathcal{R}$ par une fonction $\rho : Q \to \mathcal{R}$ indiquant les valeurs de quels registres peuvent être modifiées dans la place q.

Un FMA peut représenter une infinité d'états de la forme (q, v) où $q \in Q$ et $v : \mathcal{R} \to \mathcal{U} \cup \{\sharp\}$ est une valuation mémoire, indiquant pour chaque registre la lettre qui y est associée, ou $\sharp$ si le registre est vide. Les valuations sont subordonnés à une contrainte d'injectivité interdisant à deux registres de stocker la même valeur. Étant donné l'état (q, v), l'automate va changer d'état lors de la lecture d'une lettre $u \in \mathcal{U}$. Les états atteignables sont tous les états (q', v') tels qu'il existe une transition $\delta = (q, r, q') \in \Delta$ franchissable. La transition est franchissable soit si $v(r) = u$ et $v = v'$, c'est-à-dire que la lettre est associée au registre annotant la transition ; soit si le registre est modifiable ($r \in \rho(q)$) et que la lettre est affectée

au registre : $v'(r) = u$ et $\forall r' \in \mathcal{R}\backslash\{r\}, v(r') = v'(r') \neq u$.

Les FMA ont été développés dans l'optique de garder la plus grande proximité possible avec les automates finis afin de préserver un maximum de leurs propriétés et de limiter la complexité des problèmes d'analyse des FMA. La classe de langages reconnus par les FMA est appelée la classe des langages *quasi-réguliers*. Cette classe est close pour la concaténation, l'étoile de Kleene, la conjonction et la disjonction. De même il est possible de détecter si le langage représenté par un FMA est vide (*emptiness problem*) en interprétant l'automate comme si c'était un automate fini dont l'alphabet fini est l'ensemble de registres de l'automate. Ce problème est démontré NP-complet dans [75].

Cependant, plusieurs des propriétés des automates finis n'ont pas pu être préservées. La classe de langages définie par les FMA n'est pas close pour la complémentation et il n'est pas possible de déterminiser un FMA. De même, le problème de l'inclusion de langage et le problème de l'universalité d'un langage ne sont pas décidables [69, 70]. Enfin, l'expressivité d'un FMA est limitée par son nombre de registres, un automate à n registres est capable d'exprimer des langages contenant au plus n lettres différentes. En effet, une fois les n registres remplis, toute nouvelle lettre qui sera stockée écrasera l'une des précédentes. Aucune mécanique de l'automate ne permet de savoir si la nouvelle lettre stockée est différente d'une lettre qui a été vue et oubliée. Le langage des mots dont toutes les lettres sont différentes n'est pas exprimable par un FMA. Les principaux modèles d'automates à mémoire étudiés ici sont des extensions des FMA dont le but est d'en accroître l'expressivité tout en essayant d'en préserver les propriétés.

Les automates à prédiction

Les automates à prédictions (*Finite memory automata with non-deterministic assignement*, *Guessing Register Automata*, GRA) [56] sont des automates à mémoire mettant l'accent sur le non-déterminisme afin d'obtenir un modèle plus « simple » d'utilisation.

Les automates à prédiction (GRA) sont une variante des FMA et disposent d'un ensemble fini de registres numérotés de 1 à n pouvant chacun stocker une valeur de l'alphabet infini $\mathcal{U}$. Un GRA est représenté par un n-uplet de la forme $A_{GR} = (Q, q_0, F, \Delta, \sigma_0)$. L'automate est composé d'un l'ensemble fini de places Q où $q_0 \in Q$ est la place initiale et $F \subseteq Q$ est l'ensemble de place finales. Comme pour les FMA, une même lettre ne peut jamais être stockée simultanément dans différents registres d'un GRA. Un état d'un GRA est un couple (q, σ) où $q \in Q$ et $\sigma : [1, n] \to \mathcal{U}$ est une valuation mémoire. La valuation mémoire initiale de A_{GR} est σ_0. L'ensemble de transitions Δ contient deux types de transitions de la forme :

— (q, i, q') où $i \in [1, n]$ est une transition reliant l'état (q, σ) à (q', σ), lors de la lecture de la lettre e si $\sigma(i) = e$.

— $(q, i_\bullet, q')$ est une ε-transition reliant l'état (q, σ) à (q', σ') où la lettre associée au registre i est modifiée de manière non-déterministe et donc : $\forall j \neq i, \sigma(j) = \sigma'(j) \neq \sigma'(i)$.

La mécanique d'affectation présentée par les transitions de la forme $(q, i_\bullet, q')$ est appelée prédiction du fait du non-déterminisme qu'elle implique. La valeur associée sera réellement *déterminée* lors d'une future transition de Δ annotée avec ce registre.

Les GRA sont strictement plus expressifs que les FMA, qu'ils peuvent intégralement simuler. Ils sont aussi présentés comme « quasi-réguliers » car les propriétés, et preuves correspondantes, des FMA se prolongent naturellement aux GRA. Cependant, ce modèle échoue à dépasser la limite des FMA à représenter des langages contenant un nombre arbitraire de lettres différentes, car ils sont limités par la quantité de registres qu'ils possèdent.

La classe des automates finis à variables (*Variable Finite Automata*, VFA) [47] est une sous-classe des GRA conçu dans l'optique de créer le modèle d'automate à mémoire aussi proche des automates finis que possible. La mémoire de ce modèle est composée de n registres/variables dont les valeurs sont toutes *prédites* (affectation non déterministe des GRA) uniquement au début de la reconnaissance, plus un unique registre qui peut être réassigné lors de la lecture d'une lettre qui n'est assignée à aucun des autres registres. Bien que la classe des langages reconnaissables par les VFA soit limitée, l'utilisation du système de prédiction rend les VFA incomparables aux FMA. Il existe un sous-ensemble des VFA déterminisable mais le problème reste indécidable pour les VFA en général. Cette limite s'étend naturellement aux GRA qui sont une généralisation des VFA.

Les automates à registres frais

L'une des limitations des FMA est la limite de stockage des registres. Une fois chaque registre rempli, toute nouvelle lettre écrasera une valeur précédemment stockée et provoquera son oubli. Ainsi il n'est pas possible de savoir si une valeur est réellement nouvelle ou a été précédemment oubliée.

Le modèle des automates à registres frais, *Fresh Register Automata* (FRA) [79] est une extension des FMA définissant les notions de fraîcheur des données. Durant la reconnaissance d'un mot, une lettre/donnée est localement fraîche si elle n'est stockée dans aucun registre, et est globalement fraîche si elle n'a jamais été lue. Un FRA est un n-uplet $F = (Q, q_0, F, \Delta, \sigma_0, \mathcal{C}, \mathcal{U})$ défini sur deux alphabets, un alphabet fini de constantes $\mathcal{C}$ et un alphabet infini de données $\mathcal{U}$, et représente des langages sur $(\mathcal{C} \cup \mathcal{U})^*$. La mémoire des FRA est composée de n registres pouvant chacun stocker une unique lettre, ainsi que d'un historique H mémorisant toutes les lettres qui ont été stockées dans un registre depuis le début d'un *run*. Ainsi, l'état d'un FRA est présenté comme un triplet (q, σ, H) où q est la place courante,

$\sigma : i \to \mathcal{U}$ est la valuation mémoire et $H \subset U$ est l'ensemble des lettres qui ont été lues depuis l'état initial. L'état initial d'un FRA est le triplet $(q_0, \sigma_0, \emptyset)$ où q_0 est la place initiale et σ_0 la valuation mémoire initiale. L'ensemble Δ contient 4 types de transitions, définissant la dynamique de l'automate. Chaque transition δ permet de passer d'un état (q, σ) à un l'état (q', σ', H') lors de la lecture d'une lettre e, tel que :

- Si $\delta = (q, c, q')$ alors $e = c \in \mathcal{C}$ et $\sigma = \sigma'$ et $H = H'$,
- Si $\delta = (q, i, q')$ alors $e \in \mathcal{U}$ et $\sigma = \sigma'$ et $H = H'$,
- Si $\delta = (q, i^\bullet, q')$ alors e est une lettre localement fraîche, c'est-à-dire stockée dans aucun registre de σ, et $\sigma' = \sigma[i \to e]$ et $H' = H \cup \{e\}$,
- Si $\delta = (q, i^*, q')$ alors e est une lettre globalement fraîche, c'est-à-dire $e \notin H$ et elle n'est associé à aucun registre dans σ_0, et $\sigma' = \sigma[i \to e]$ et $H' = H \cup \{e\}$,

Le modèle des FRA est strictement plus expressif qu'un FMA tout en restant très similaire, les problèmes qui étaient décidables pour les FMA le restent pour les FRA, par exemple l'*emptiness problem*. Cependant, le gain d'expressivité reste très faible car il permet uniquement de faire la différence entre les données localement fraîches et celles globalement fraîches. Seules les données stockées dans des registres peuvent être concrètement manipulées. De plus, ce modèle n'est plus clos pour les opérations de concaténation et l'étoile de Kleene.

Automates à historiques

L'utilisation d'un historique, comme celui des FRA, a été généralisée dans les automates à historiques (*History Register Automata*, HRA) [46]. Ce modèle est une variante des FMA utilisant simultanément des registres et des historiques. Chaque historique peut stocker un sous-ensemble fini de lettres de $\mathcal{U}$ et être consulté lors du franchissement des transitions. À la différence des registres, une même donnée peut être stockée simultanément dans différents historiques, il n'y a pas de contrainte d'injectivité. Il est prouvé que les historiques peuvent totalement simuler le fonctionnement des registres.

Pour en simplifier la présentation, nous considérons des HRA utilisant uniquement des historiques et aucun registre, car ces derniers n'apportent pas d'expressivité supplémentaire. Les HRA sont des n-uplets $A_{HR} = (Q, q_0, F, \Delta, H, \sigma_0)$ composé d'un ensemble de places Q, d'une place initiale $q_0 \in Q$ et de places finales $F \subseteq Q$. L'automate est défini sur un ensemble d'historiques H pouvant chacun stocker un sous-ensemble de lettres de l'alphabet infini $\mathcal{U}$, où deux historiques peuvent stocker les mêmes lettres. Les états d'un HRA sont définis comme des paires (q, σ) où $q \in Q$ et σ est une valuation des historiques $\sigma : H \to 2^{\mathcal{U}}$. L'état initial d'un HRA est la paire (q_0, σ_0) composée de la place initiale q_0 et la valuation initiale σ_0. L'ensemble Δ est composé des transitions de la forme :

— (q, H_r, H_w, q') où $H_r, H_w \subseteq H$, permettant d'aller d'un état (q, σ) à l'état (q', σ') lors de la lecture d'une lettre $e \in \mathcal{U}$ si e est associée exactement à chaque historique de H_r dans σ et que e est exactement associée à chaque historique de H_w dans σ'. Cela permet de déplacer la lettre lue d'un ensemble d'historiques à un autre, voir de l'oublier si $H_w = \emptyset$. Si $H_r = \emptyset$ alors la lettre doit être fraîche.
— $(q, H_\emptyset, q')$ où $H_\emptyset \subseteq H$, est une ε-transition permettant d'aller d'un état (q, σ) à (q', σ') sans lire de lettre, si $\sigma' = \sigma[H_\emptyset \to \emptyset]$, c'est-à-dire que tous les historiques de $H_\emptyset$ ont été réinitialisés dans l'état d'arrivée.

Les HRA sont bien plus expressifs que les FRA et sont clos pour la concaténation et l'étoile de Kleene en plus de l'union et l'intersection de langages. Une étude approfondie de la complexité de l'*emptiness problem* a été effetuée sur les HRA et différentes sous classes du modèle [46], mais dans le cas général il est décidable et appartient à la classe de complexité *Ackermann*-complet.

Il existe un autre modèle utilisant des historiques : les ν-automates [31]. Ce modèle est une variante des FMA où les registres ont été remplacés par des historiques. À la différence des HRA, une donnée ne peut pas être stockée simultanément dans différents historiques, ce qui est équivalent à la contrainte d'injectivité des FMA. Il n'est cependant plus possible de déplacer les valeurs entre les historiques. Ce modèle a principalement été développé pour représenter l'utilisation de ressources dynamiques, représentées par des historiques, dans les systèmes concurrents. Un algorithme de minimisation du nombre d'historiques a été développé [32] et nous avons développé un algorithme de reconnaissance pour ce modèle [18].

Les autres modèles

D'autres modèles d'automate n'essayent pas d'étendre les FMA ou les automates finis, mais visent une expressivité similaire à celle de la logique monadique du second ordre, ou à certaines sous-classes [69, 70, 23]. En effet, les langages exprimés par les FMA et nombre de leurs variantes sont présentés comme orthogonaux aux propriétés de ces logiques [70].

Les automates à galets (*Pebbles Automata*, PA) [70], sont une classe d'automates à mémoire strictement plus expressive que la logique du première ordre et incluse dans la logique monadique du second ordre. La mémoire de ces automates est représentée par un nombre fini de galets qui peuvent être positionnés sur des lettres du mot lu afin de les comparer. À la différence des FMA, les galets permettent de mémoriser à la fois la valeur de la lettre mais aussi sa position dans le mot. En plus de mémoire, les galets sont équivalents à des têtes de lectures gérées par une mécanique de pile. Ainsi, lors de la reconnaissance, il peut être nécessaire de lire plusieurs fois les mêmes lettres. Les langages exprimés par les PA sont clos pour l'union, intersection, concaténation et étoile de Kleene. Cependant

le problème de la détection du langage vide est indécidable pour les PA.

Le modèle des automates à données (*Data Automata*) [23] est composé à la fois d'un transducteur et d'un automate de classe afin de représenter des langage de données de la forme $(\Sigma \times \mathcal{U})^*$, où σ est un alphabet fini et $\mathcal{U}$ l'alphabet infini de données. Un transducteur est un automate fini dont chaque transition lit une lettre Σ d'un mot et en réécrit une dans un autre alphabet Γ. L'automate de classe est un automate fini non-déterministe défini sur l'alphabet Γ. Ici, on appelle une classe d'un mot w de langage de donnée la projection de ce mot par rapport à une donnée d, c'est-à-dire, le mot composé des lettres associées à d dans w. Un mot $w \in (\Sigma \times \mathcal{U})^*$ est accepté par un automate à données s'il est accepté par le transducteur et que toutes les classes du mot écrit par le transducteur sont acceptées par l'automate de classes.

Les automates de données sont présentés comme aussi expressifs que la logique monadique du second ordre existentielle sur les langages de données, $\exists$*MSO on data language* [23]. Le problème du langage vide est décidable pour les automates à données et ils sont strictement plus expressifs que les automates à registres. Un modèle d'automates appelé *Class Memory Automata* (CMA), présenté comme aussi expressif que les automates à données [20, 21], est d'une expressivité incomparable à celle des HRA. Les CMA sont strictement plus expressifs que la sous classe des HRA ne disposant pas de la ré-initialisation [46]. Il est prouvé que le problème du langage vide pour cette sous-classe des HRA est de complexité ExpSpace-complet, ce qui donne une borne inférieure à la complexité du problème sur les CMA et donc les automates à données.

Ces modèles d'automates représentent des formalismes de reconnaissances pour les langages sur alphabet infini, et nous allons maintenant présenter les formalismes de spécification.

2.3.2 Les langages de spécification

L'expression de motifs contenant des données d'un alphabet infini est un problème qui a été popularisé il y a plusieurs décennies lors de la conception des premiers systèmes de bases de données [5]. Il était alors nécessaire de créer des langages de requêtes permettant l'accès aux données respectant certaines propriétés ou conditions. Les langages de requêtes sur les bases de données orientées graphe sont ceux où il est le plus clairement nécessaire de pouvoir décrire des langages de données similaires à des expressions régulières [11, 64]. La popularisation du XML à créé un regain d'intérêt dans la vérification de systèmes contenant des données, avec notamment le développement des automates à registres et des logiques sur les langages de données [33, 23, 55]. Avec les différentes classes d'automates à mémoire qui ont été formalisées depuis, plusieurs langages de spécification dérivés des expressions régulières ont été conçus pour spécifier des automates à mémoire par

rapport aux langages qu'ils peuvent reconnaître.

Le modèle des GRA a été développé simultanément avec un langage de spécification aussi expressif : les expressions régulières pour alphabet infini (*Infinite Alphabet Regular Expressions*, IARE) [56]. Ce langage de spécification est une extension des expressions régulières où l'alphabet fini utilisé est un ensemble de registres. Deux nouveaux opérateurs y sont ajoutés, $\cdot_x$ et $*_x$, qui sont des généralisations de la concaténation et de l'étoile de Kleene où le registre x qui y est annoté voit sa valeur changer de façon non-déterministe. Ce langage possède une notion de portée des données stockées dans les registres et possède toutes les propriétés des GRA.

Les expressions régulières avec mémoire, *regular epression with memory* (REM) [63] sont un langage de spécification pour les langages reconnaissables par les FMA. À la différence des IARE, les langages de données exprimés par les REM ont la forme de paires $\binom{l_1}{d_1}\binom{l_2}{d_2}\binom{l_3}{d_3}\dots$ associant des lettres de l'alphabet fini $l_1, l_2, l_3 \dots \in \Sigma$ à des données de l'alphabet infini $d_1, d_2, d_3 \cdots \in \mathcal{U}$. Les REM sont une extension des expressions régulières disposant d'un ensemble de registres $\mathcal{R}$ permettant de stocker les données. La syntaxe des REM est la même que celles des expressions régulières définies sur un alphabet Σ à laquelle s'ajoutent les expressions atomiques de la forme $a[c] \downarrow I$ où $a \in \Sigma$, $I \subseteq \mathcal{R}$ et c est une contrainte. Les contraintes sont des expressions booléennes précisant quels registres stockent une donnée égale ou différente de d. La sémantique de $a[c] \downarrow I$ correspond à toutes les lettres $\binom{a}{d}$ telles que la contrainte c est satisfaite, puis d est stockée dans les registres de I. Toutes les REM peuvent être traduites dans le FMA reconnaissant le même langage.

Les expressions régulières avec égalité, *Regular Expression with Equality* (REME) [63], sont un langage de spécification dérivé des REM permettant d'exprimer des langages de données sans faire apparaître explicitement de registre ou de mémoire. C'est une extension des expressions régulières où des contraintes sur les données peuvent être exprimées à l'aide de parenthèses de la forme $(e_=)$ ou $(e_\neq)$ (où e est une expression). Elles représentent respectivement des langages dont la première et dernière lettre sont associées à la même donnée $(=)$, ou à des données différentes $(\neq)$. Ce langage de spécification est plus simple à utiliser mais est strictement moins expressif que les REM et n'est pas clos pour l'intersection. Le gros avantage de cette sous classe est que les complexités des problèmes de reconnaissance d'un mot et du langage vide sont PTIME-complet.

2.3.3 L'algorithmique de reconnaissance

Même si les automates à mémoire possèdent une grande expressivité, spécialement pour la représentation de systèmes ouverts, peu de travaux proposent des algorithmes de reconnaissance les concernant. Cela peut s'expliquer par le fait que

la complexité du problème de la reconnaissances est élevée même pour les modèles les moins expressifs. Ainsi, pour les automates à registres (FMA) [55], le problème est déjà NP-complet [75].

À notre connaissance, les seuls travaux exploitant les automates à registres (FMA) comme un modèle dans le cadre de la supervision de système sont présentés dans [45]. Ils utilisent en réalité les TOPL-automates, un modèle aussi expressif que les FMA, comme formalisme de reconnaissance dans le cadre de la supervision de programmes Java. Les TOPL-automates permettent de représenter des propriétés dans le byte code Java et sont exprimables à l'aide d'un langage de spécification, les *TOPL properties*, permettant de décrire ces propriétés. Relativement proche des automates à registres, le modèle des *quantified event automata* (QEA) [73, 12] est développé spécifiquement pour la vérification et la supervision de systèmes. Le fonctionnement de la mémoire et des paramètres des transitions est similaire à ceux des TOPL automates, mais les gardes et la possibilité de spécifier des quantificateurs rendent les QEA bien plus expressifs. Les QEA sont utilisés dans l'outil MarQ [73] de vérification à la volée. L'outil fait preuve d'une grande efficacité et de bonne performance comparé à d'autres outils de vérification. Cela montre que malgré la grande complexité théorique des modèles d'automates à mémoire, il est possible de les utiliser efficacement dans des applications réelles comme la supervision de systèmes.

L'article [87] traite de la reconnaissance avec un modèle d'automate paramétré. Il y est défini un modèle d'automate temporisé paramétré dont les paramètres peuvent appartenir à un domaine arbitraire de données (entier, chaîne de caractères, BDD...). La reconnaissance à base d'automate paramétré consiste à déterminer, pour un mot w donné, les valeurs des paramètres spécifiés dans l'automate pour qu'il reconnaisse w. Cela s'apparente alors à la reconnaissance de langage temporisé de données, où il est possible d'exprimer des contraintes de temps paramétrées.

2.4 Synthèse

Il existe de nombreux langages de spécification et modèles d'automates permettant de représenter des propriétés temporisées. C'est un domaine qui a été très largement étudié, et qui a pour point central les automates temporisés [2]. Cette extension des automates finis est un formalisme de reconnaissance pour les langages temporisés. En plus d'avoir un fonctionnement simple et d'être capable d'exprimer des contraintes de temps relativement complexes, la grande partie des problèmes d'analyse du modèle (langage vide et appartenance) sont décidables. La majorité des extensions possibles des automates temporisés mènent à des modèles dont le problème du langage vide (*emptiness checking*) est indécidable [24, 50, 1].

Il existe plusieurs langages de spécification permettant d'exprimer des propriétés reconnaissables par les automates temporisés classiques. Ceux qui nous intéressent dans le cadre de cette thèse sont ceux dérivés des expressions régulières : *timed regular expressions*, *balenced timed regular expressions* et le langage de composition d'automates [7, 8, 25]. Ces langages permettent d'exprimer de manière explicite des langages temporisés et peuvent être directement traduits sous la forme d'un automate temporisé reconnaissant le même langage.

Pour ce qui est de la représentation des langages de données, de nombreux modèles d'automates à mémoire existent. L'un des modèles les plus connus est celui des automates à registres (FMA). Ils offrent une expressivité relativement limitée par rapport aux autres modèles d'automate, mais ont cependant de bonnes propriétés de décidabilité et de fermeture. L'une des particularités de ce modèle est la présence d'une contrainte d'injectivité interdisant de stocker une même lettre dans différents registres. À l'aide de cette propriété, le fonctionnement des FMA reste similaire à celui des automates finis. Cela a facilité la conception de plusieurs langages de spécification pour les FMA, étendant les expressions régulières [56, 63].

Pour notre cas d'application, l'analyse à la volée de flots de liens, l'expressivité des FMA est trop limitée car les langages qu'ils reconnaissent ne peuvent spécifier qu'un nombre fini de lettres différentes. En effet, les systèmes complexes étant définis sur de vastes environnements ouverts, les motifs que l'on souhaite représenter doivent pouvoir spécifier un nombre arbitraire d'entités différentes. Les automates à registres frais (FRA) [79] et les automates à historiques (HRA) [46] sont dérivés des FMA et permettent la spécification de langages utilisant un nombre arbitraire de lettres différentes. Là où les FRA sont une extension stricte des FMA ajoutant la possibilité de spécifier une lettre globalement fraîche, les HRA définissent un modèle mémoire très différent, à base d'historiques, qui permet d'obtenir une bien plus grande expressivité. L'accroissement de l'expressivité entraîne une difficulté d'analyse plus importante, où le problème du langage vide est *Ackermann*-complet pour les HRA. Nous souhaitons définir notre propre classe d'automates intermédiaire à ces deux modèles. Elle doit permettre de manipuler une quantité arbitraire de lettres dans sa mémoire, tout en préservant la mécanique d'injectivité des FMA et FRA pour faciliter la conception d'un langage de spécification.

Ainsi pour résumer notre approche, nous allons concevoir un modèle d'automate à mémoire, ainsi que le langage de spécification équivalent, dont l'expressivité est à mi-chemin de celles des HRA et des FRA afin de préserver une complexité d'analyse décidable. Ensuite nous allons étendre ce modèle avec les fonctionnalités des automates temporisés pour permettre la spécification de contraintes temporisées. Cette extension sera aussi effectuée sur le langage de spécification, en s'inspirant de langages déjà présents dans la littérature [25, 7].

Les domaines des langages de données et des langages temporisés sont en réalité

très proches l'un de l'autre. Les langages de données sont une généralisation des langages temporisés et les automates à registres sont *essentiellement équivalents* aux automates temporisés comme expliqué dans [40]. Cela nous conforte dans l'idée de pouvoir créer un modèle pouvant exprimer des langages temporisés à données.

Pour ce qui est du domaine de la reconnaissance, très peu de travaux existent sur les langages de données à notre connaissance. Ceux existant sont des applications au domaine de la vérification dynamique et de la supervision [12, 45]. On peut supposer qu'il existe plusieurs outils de supervision qui peuvent s'apparenter à des modèles d'automates à mémoire ou des logiques sur les alphabets infinis, mais dont le lien n'a pas été formellement étudié. Dans le cadre de la reconnaissance de langages temporisés, il existe plusieurs algorithmes traitant du problème et ce sur plusieurs formalismes de spécification [85, 82, 81]. En particulier, dans [10], un algorithme de reconnaissance de signaux sur les automates temporisés est présenté. Il adapte le formalisme des DBM [35], pour représenter les zones temporelles, au problème de la reconnaissance. C'est un travail similaire à celui que nous présentons dans la Section 5.2, et que nous avons publié simultanément dans [19]. La solution qu'ils présentent a l'avantage d'être plus abstraite que la nôtre, et ce par l'ajout d'une horloge globale qui permet de simplifier les calculs en réutilisant des opérations classiques des DBM. Nous présentons tout de même notre approche, qui a l'avantage de s'appliquer directement sur le modèle d'automates à mémoire que nous avons développé.

Chapitre 3

La spécification de propriétés hybrides

Dans l'optique d'exprimer des propriétés dans les systèmes complexes, nous nous sommes intéressés à la théorie des langages. Nous interprétons ces systèmes comme des flots de liens, ce qui correspond à une séquences d'interactions entre différents acteurs. Les propriétés prennent la forme d'un ensemble de scénarios, de motifs d'interactions entre les différents acteurs. Dans la théorie des langages, nous représentons les motifs comme des langages et les instances de ces motifs comme des mots, où les interactions sont les lettres de notre alphabet. Notre idée est de proposer un langage de spécification inspiré des expressions régulières mais capable d'exprimer des contraintes temporelles et des propriétés sur les environnements ouverts.

Quelques langages de spécifications dérivés des expressions régulières permettent de représenter des propriétés temporisées. Entre autres, les compositions modales d'automates temporisés [25] et les *Balanced Timed regular expressions* (BTRE) [8] nous servent de base pour définir des opérateurs de contraintes de temps. Dans la théorie des langages, les propriétés sur les environnements ouverts correspondent aux langages sur alphabet infini exprimées sous la forme d'automates à mémoire. Pour exprimer ces deux types de propriétés dans un même langage, nous avons défini le langage des *expressions temporisées à couches mémoire* (ETCM). Ce langage de spécification permettra l'expression de langages temporisés sur alphabet infini qui représenteront alors nos motifs.

Comme les ETCM mélangent deux formalismes relativement complexes, nous décidons de les définir en deux parties. Dans un premier temps sont présentées les *expressions à couches mémoire* (ECM), la version non-temporisée des ETCM permettant seulement d'exprimer des langages sur alphabet infini. Cela permet entre autre de détailler le fonctionnement particulier de la mémoire et des couches mémoire. Ensuite, les ETCM sont présentées de manière à mettre en avant la

temporisation des langages sur alphabet infini et le fonctionnement des opérateurs de contraintes de temps. Enfin, le chapitre se conclut sur une section présentant différents exemples de motifs dans les flots de liens exprimés avec des ETCM

3.1 La syntaxe des expressions à couches mémoire

Les expressions à couches mémoire (ECM) sont une extension des expressions régulières mais définissent des langages sur alphabet infini. Alors que les langages réguliers sont définis sur un alphabet fini de lettres Σ, les langages de données représentent des propriétés sur un alphabet infini $\mathcal{U}$. Dans les ECM, il est possible d'exprimer des langages définis sur les deux alphabets Σ et $\mathcal{U}$, cependant, afin d'éviter les ambiguïtés, ils sont considérés comme distincts, $\Sigma \cap \mathcal{U} = \emptyset$. La syntaxe des expressions à couches mémoire est présentée dans la Table 3.1.

Terminal $t ::=$		
	$\emptyset$	(Langage vide)
$\mid$	a	(Constante)
$\mid$	X^l	(Variable)
$\mid$	$\sharp X^l$	(Valeur fraîche)
Expression $e, e_1, e_2, \ldots ::=$		
	t	(Terminal)
$\mid$	$e_1 \cdot e_2$	(Concaténation)
$\mid$	$e\overline{X^l}!$	(Réinitialisation)
$\mid$	$e_1 + e_2$	(Disjonction)
$\mid$	$e_1 \,\&\, e_2$	(Conjonction)
$\mid$	$e^*/\bar{l}/$	(Itération)
$\mid$	$e \uparrow_n$	(Décalage couche)

TABLE 3.1 – La syntaxe des Expressions à Couches Mémoire. (ECM)

Comme les ECM sont une extension des expressions régulières, sa syntaxe en contient les expressions terminales et opérateurs. Ainsi, toute expression régulière peut être interprétée comme une ECM. On y retrouve le langage vide $\emptyset$ et les constantes $a \in \Sigma$, qui représentent les mots composés d'une unique lettre de l'alphabet fini. La syntaxe des ECM contient les opérateurs de concaténation $\cdot$, disjonction $+$, conjonction $\&$ et d'itération (étoile de Kleene) $*$. Ces opérateurs conservent leurs principes de base, cependant leurs fonctionnements ont été altérés afin de prendre en compte la présence de la mémoire.

3.1.1 Les variables

Les ECM se distinguent par la présence d'une mémoire permettant d'enregistrer de l'information au cours de l'évaluation d'une expression. Cette mémoire est manipulée à l'aide de variables, notées X^l, qui permettent d'identifier des lettres de l'alphabet $\mathcal{U}$. Les ECM prennent un aspect semblable aux langages de programmation, où plus on progresse dans la lecture d'une expression, plus le langage exprimé dépend du préfixe déjà reconnu.

Les variables stockent une ou plusieurs lettres de $\mathcal{U}$ au cours de l'évaluation d'une expression, afin d'en repérer les éventuelles répétitions. L'exposant l d'une variable indique le numéro de la couche d'injectivité dont elle fait partie. Chaque couche représente une contrainte sur l'ensemble de ses variables. Il est interdit, à un instant donné, qu'une lettre soit associée à différentes variables d'une même couche.

L'expression terminale X^l permet d'accéder aux lettres stockées dans la variable X^l, qui sont donc des lettres $\mathcal{U}$. Cette expression définit le langage des mots composés d'une seule lettre qui est l'une de celles associées à X^l. Si aucune lettre n'est actuellement stockée dans X^l alors le langage exprimé est $\emptyset$.

Le terminal $\sharp X^l$ permet d'ajouter une nouvelle lettre dans la variable X^l. Il exprime le langage qui est l'ensemble infini de tous les mots composés d'une unique lettre de $\mathcal{U}$, qui n'est actuellement associée à aucune variable de la couche l. La lettre est stockée dans la variable pour d'éventuelles utilisations futures, voir l'exemple 3.1.1.

3.1.2 La concaténation

La concaténation, représentée ici par le symbole $\cdot$, permet de mettre bout à bout deux expressions. Étant donné un mot u appartenant au langage de l'expression e_1 et un mot v appartenant au langage de e_2, la concaténation dans les expressions régulières stipule que le mot uv appartient au langage de $e_1 \cdot e_2$. Cette définition indique que les mots des deux langages sont choisis indépendamment l'un de l'autre. Or, dans les ECM, ce n'est pas toujours le cas, les variables qui sont associées à des lettres dans e_1 y sont toujours associées dans e_2. Cela va modifier le langage représenté par e_2 car ses variables sont initialisées aux lettres finales de celles de e_1.

Exemple 3.1.1 (Variables et concaténation) *Cet exemple permet de voir le fonctionnement des variables lorsqu'elles sont utilisées avec la concaténation. Considérons l'expression :*

$$\sharp X^1 \cdot X^1 \cdot \sharp Y^1 \cdot \sharp Y^1 \cdot Y^1$$

Dans cette expression deux variables sont présentes, X^1 et Y^1, et elles font toutes les deux partie de la couche 1. Sans perte de généralité, nous supposons que les deux variables sont initialement vides, c'est-à-dire qu'aucune lettre ne leur est associée. Cette expression est composée de 5 terminaux concaténés les uns après les autres, ils représentent chacun une lettre des mots du langage.

La première lettre d'un mot du langage est représentée par l'expression $\sharp X^l$. Cette expression représente une lettre arbitraire de $\mathcal{U}$ absente de la couche 1, donc n'importe quelle lettre de $\mathcal{U}$, car les variables sont initialement vides. Une fois qu'une première lettre, disons a, est reconnue, elle est associée à X^1. La seconde lettre est représentée par le second terminal X^1. Ce terminal représente une lettre arbitraire parmi celles actuellement stockées dans X^1, et est donc dans notre exemple une deuxième occurrence de a.

La troisième lettre est représentée par $\sharp Y^1$, et doit donc être différente de celles actuellement stockées dans la couche 1, donc différente de a. Cette troisième lettre, notons b, est stockée dans Y^1. Ainsi, après avoir reconnu la troisième lettre du mot, la variable Y^1 stocke b et la variable X^1 stocke la lettre a.

L'expression $\sharp Y^1$ représente la quatrième lettre des mots acceptés. Cette lettre aussi doit être absente de la couche 1, donc différente de la lettre a stockée dans X^1 mais aussi de la lettre b déjà stockée dans Y^1. Ainsi la quatrième lettre, notée c, est associée à Y^1, ainsi Y^1 stocke b et c. Enfin la cinquième et dernière lettre est reconnue par l'expression Y^1 et est donc une des lettres associées à Y^1. Donc dans notre exemple, la cinquième lettre est soit b soit c.

On peut en conclure que cette expression définit le langage des mots de cinq lettres de la forme $\{aabcb, aabcc\}$ pour toutes lettres $a, b, c \in \mathcal{U}$ différentes les unes des autres.

3.1.3 La réinitialisation

L'opérateur de ré-initialisation, noté $e\overline{X^l}!$, est paramétré par un ensemble de variables $\overline{X^l}$ et s'applique en suffixe d'une expression e. Il permet de ré-initialiser l'ensemble des variables de $\overline{X^l}$, c'est-à-dire de supprimer l'ensemble des lettres associées à ces variables. Cet opérateur ne modifie pas directement le langage de l'expression e, mais altère les valeurs des variables pour les expressions qui seraient concaténées à la suite de cette expression. Les lettres qui étaient auparavant associées aux variables ré-initialisées pourront de nouveau être associées à des variables dans les expressions suivantes.

Exemple 3.1.2 (Les ré-initialisations de variables)

Pour illustrer la ré-initialisation, nous allons prendre l'exemple du langage de l'expression suivante :

$$\sharp X^1 \cdot \sharp Y^1\{X^1\}! \cdot \sharp Z^1$$

Cette expression utilise trois variables qui font toutes parties de la couche 1*, elles ne peuvent donc pas stocker les mêmes lettres au même moment. Nous considérons que les variables sont initialement vides. Cette expression représente un langage contenant uniquement des mots de trois lettres.*

La première lettre, notée a est représentée par le terminal $\sharp X^1$*, et est donc une lettre de* $\mathcal{U}$ *qui n'est associée à aucune autre variable de la couche* 1*. Comme les variables sont initialement vides, alors a peut être une lettre quelconque de* $\mathcal{U}$*. La lettre a est associée à la variable* X^1*.*

La seconde lettre, notée b, est représentée par le terminal $\sharp Y^1$ *et est aussi une lettre de* $\mathcal{U}$ *qui n'est associée à aucune variable de la couche* 1*, donc différente de a qui est déjà associée à* X^1*. La lettre b est associée à la variable* Y^1*. L'opérateur* $\{X^1\}!$ *en suffixe de l'expression* $\sharp Y^1$*, indique que la variable* X^1 *est réinitialisée et donc la lettre a n'y est plus associée.*

La troisième et dernière lettre, notée c, est représentée par le terminal $\sharp Z^1$*. Cette lettre est donc différente des autres lettres de la couche* 1*, donc différente de b qui est associée à* Y^1*. Cependant, c peut être égale à a, la première lettre du mot, car elle n'est plus stockée dans la couche* 1*.*

Le langage de cette expression est donc l'ensemble des mots de trois lettres où la lettre du milieu est différente de ses voisines, formellement : $\{abc \mid a \neq b, b \neq c\}$ *où* $a, b, c \in \mathcal{U}$*.*

3.1.4 Les disjonction et conjonction

Les opérateurs de disjonction et conjonction dans les ECM sont identiques à ceux des expressions régulières. L'opérateur de disjonction, représenté par $+$, permet de laisser le choix entre deux expressions possibles. L'expression $e_1 + e_2$ va ainsi définir le langage qui est l'union des langages de e_1 et e_2. La conjonction, symbolisée par le symbole &, permet d'exprimer des langages dont les mots doivent respecter plusieurs expressions simultanément. L'expression e_1 & e_2 représente le langage qui est l'intersection des langages représentés par e_1 et e_2.

Exemple 3.1.3 (La conjonction et les couches) *Le fonctionnement des couches est particulièrement visible dans les expressions utilisant la conjonction. On peut alors voir deux couches comme deux ensembles distincts de variables qui sont chacun utilisé pour une des sous-expressions de la conjonction.*

$$\sharp X^1 \cdot X^1 \cdot \big((\sharp X^1 \cdot X^1) \,\&\, (\sharp Y^2 \cdot Y^2)\big)$$

Pour cette expression aussi nous supposons les variables initialement vides. Cette expression décrit un langage composé de mots de quatre lettres dont les deux

premières sont représentées par les deux premiers terminaux et les deux dernières par la conjonction.

Les deux premiers terminaux représentent un motif similaire au début du langage de l'exemple 3.1.1. Le premier terminal $\sharp X^1$ *va reconnaître et associer à* X^1 *une lettre a absente de la couche* 1*. Comme les variables sont initialement vides, a est une lettre quelconque de* $\mathcal{U}$*. Le second terminal représente l'une des lettres associée à* X^1*, donc la seule possible ici est a.*

La suite du motif $(\sharp X^1 \cdot X^1)$ & $(\sharp Y^2 \cdot Y^2!)$ *est la conjonction de deux expressions. La première* $\sharp X^1 \cdot X^1$ *est une répétition des deux premiers terminaux, et définit une partie des contraintes sur les troisième et quatrième lettres des mots du langage. La troisième lettre, notons-la b, va être associée à* X^1 *et doit donc être différente de celles actuellement stockées dans les variables de la couche 1, donc différente de a. Puis la quatrième lettre doit être l'une de celles associées à* X^1*, donc soit a soit b.*

La seconde expression de la conjonction, $\sharp Y^2 \cdot Y^2$*, décrit le reste des contraintes sur les troisième et quatrième lettres. La troisième lettre est contrainte par* $\sharp Y^2$*, qui indique qu'elle ne doit pas être présente dans la couche 2, et est stockée dans* Y^2*. Comme aucune lettre n'a été précédemment associée à une variable de la couche 2 dans l'intersection, cette expression n'ajoute pas de contrainte supplémentaire sur la valeur possible de la troisième lettre. Toutes les lettres de* $\mathcal{U}$*, a comprise, sont autorisées. L'expression* Y^2 *qui suit, indique que la quatrième lettre doit être l'une de celles stockée dans* Y^2*, donc la seule possible est celle qui vient d'y être stockée.*

Mises ensemble, ces contraintes indiquent que la troisième lettre, b, doit être stockée simultanément dans X^1 *et* Y^2*, et est donc différente de a. La quatrième lettre doit être une lettre qui est stockée à la fois dans* X^1 *et dans* Y^2*, donc la seule possibilité est b. Le langage reconnu par cette expression est donc* $\{aabb \mid a, b \in \mathcal{U}, a \neq b\}$*.*

3.1.5 L'itération

Le dernier opérateur dérivé des expressions régulières est l'itération $*$ qui est une généralisation de l'étoile de Kleene. Cet opérateur unaire s'écrit en suffixe d'une expression afin de décrire un langage composé d'une concaténation de mots du langage décrit par cette expression. Par exemple $(ab)^*$ représente le langage des mots ε, *ab*, *abab*, *ababab*... Comme pour la concaténation, le fonctionnement de l'étoile est modifié afin de tenir compte de l'évolution des valeurs des variables entre les itérations successives. Cependant, pour être une généralisation de l'étoile de Kleene, il est nécessaire que l'itération dispose d'un mécanisme permettant de garder indépendant le langage reconnu à chaque itération. Ce mécanisme est représenté par l'argument $/\bar{l}/$ qui est le sous-ensemble de couches dont les valeurs des variables sont transmises entre les itérations successives. À la fin de chaque

itération, les variables des couches absentes de $/\bar{l}/$ sont réinitialisées aux valeurs qu'elles avaient au début de l'itération (pas nécessairement vide). Donc l'opérateur $*/\emptyset/$ correspond à l'étoile de Kleene régulière car exactement le même langage est réutilisé pour chaque itération, alors que l'opérateur $*/\mathbb{Z}/$ permet à toutes les valeurs de chaque variable d'être propagées d'une itération à l'autre.

Exemple 3.1.4 *Cet exemple présente le fonctionnement de l'itération. Il s'agit d'une modification de l'exemple 3.1.3 où l'on a déplacé le terminal X^1 à la fin de l'expression, et ajouté l'opération d'itération autour de la conjonction.*

$$\sharp X^1 \cdot \left((\sharp X^1 \cdot X^1) \,\&\, (\sharp Y^2 \cdot Y^2)\right)^* /\{2\}/ \cdot X^1$$

Nous supposons toujours les variables initialement vides. Cette expression reconnaît des mots composés d'un nombre pair de lettres car chaque itération est toujours composée de deux lettres, comme on a pu le voir précédemment pour cette conjonction. L'expression débute par l'association d'une lettre arbitraire $a \in \mathcal{U}$ à la variable X^1.

Ensuite, l'expression permet d'itérer plusieurs fois des lettres exprimées par la conjonction de l'exemple 3.1.3. Cette itération est annotée avec $/2/$ ce qui indique que les lettres stockées dans Y^2 à la fin d'une itération seront transmises à la prochaine itération. Cependant, à la fin de chaque itération, toutes les variables de la couche 1 seront réinitialisées aux valeurs qu'elles avaient au début de l'itération, c'est-à-dire que la variable X^1 stockera toujours uniquement a au début de chaque itération.

Chaque itération correspond à deux lettres du mot qui doivent appartenir au langage de la conjonction. La partie $\sharp X^1 \cdot X^1$ représente toujours exactement le même langage à chaque itération car la valeur de X^1 est réinitialisée à la lettre a à la fin de chacune d'entre elles. La première lettre b_i de la i^e itération est associée à X^1 et doit donc être différente de a qui est déjà stockée. La seconde lettre de l'itération doit être une des lettres actuellement stockées dans X^1 donc soit a soit b_i.

La seconde sous-expression, $\sharp Y^2 \cdot Y^2$, concerne uniquement la couche 2, dont la valeur au début de chaque itération change. Lors de la première itération Y^2 est vide car n'a jamais été modifiée. Les lettres reconnues seront les mêmes que dans l'exemple 3.1.3, c'est-à-dire, que la première lettre de l'itération, notée ici b_1, sera associée à Y^2 et peut donc être une lettre quelconque de $\mathcal{U}$. Mais comme b_1 est aussi contrainte par $\sharp X^1$ alors b_1 doit être différente de a. De même la seconde lettre acceptée doit être à la fois dans X^1 et dans Y^2, donc la seule possible est b_1.

Au début de la seconde itération, X^1 reprend sa valeur initiale et devient de nouveau associée uniquement à a. Cependant, Y^2 garde les lettres qui lui étaient associées à la fin de la première itération et reste donc associée à b_1. Ainsi, au début

de la i^e itération, la variable Y^2 est associée à l'ensemble de lettres $\{b_1, b_2, \dots b_{i-1}\}$. La première lettre de la i^e itération, notée b_i, sera associée à la fois à X^1 et Y^2 et doit donc être différente de a et de toutes les lettres actuellement associées à Y^2. La seconde lettre de la i^e itération doit être une de celles associées aux deux variables, donc la seule possibilité est b_i car X^1 stocke uniquement a et b_i, et Y^2 ne contient pas a.

La dernière lettre du mot est reconnue par le terminal X^1 et doit donc être une lettre associée à X^1. Comme c'est à la fin de chaque itération que les variables sont réinitialisées, à ce moment la variable X^1 contient uniquement la lettre a. La variable Y^2 contient toujours toutes les lettres de b_1 à b_n où n est le nombre d'itérations.

Le langage reconnu par cette expression est donc :

$$\{ab_1b_1b_2b_2 \dots b_nb_na \mid n \in \mathbb{N}, \forall i, j, b_i \neq a, i \neq j \Rightarrow b_i \neq b_j\}$$

où $a \in \mathcal{U}$ et $\forall i \in \mathbb{N}^*, b_i \in \mathcal{U}$.

3.1.6 Le décalage de couche

Les ECM sont aussi dotées de l'opérateur de décalage de couches mémoire $\uparrow$ qui permet la définition d'opérateurs dérivés et de faciliter la composition entre les expressions. L'opérateur $e \uparrow_n$, où $n \in \mathbb{N}$ décale les numéros de toutes les couches mémoire des variables de l'expression e en y additionnant n. L'utilisation de cet opérateur ne modifie pas le langage associé à l'expression e mais est une réécriture des exposants des variables qui sont écrits dans l'expression.

3.2 La sémantique formelle des ECM

Cette section est consacrée à la présentation formelle de la sémantique des expressions à couches mémoire. Cependant, il est nécessaire de débuter par une présentation de la structure représentant la mémoire. Ainsi, la section débute par une présentation formelle des notions de variables et de couches avec la définition des contextes mémoire. Enfin nous définissons les langages sur alphabet infini et la sémantique des ECM pour expliquer comment elles les expriment.

3.2.1 Le contexte mémoire

Le contexte mémoire représente la connaissance que l'on garde de l'environnement que l'on étudie. Il est composé des variables qui stockent des lettres de l'alphabet infini durant la reconnaissance. Chaque variable, noté X^l, fait partie

d'une couche mémoire représentée par le numéro l en exposant. Les couches mémoire sont la particularité de notre modèle, elles segmentent la mémoire en parties indépendantes. On peut voir les contextes mémoire comme des fonctions d'association indiquant à quel ensemble de lettres est associée chaque variable.

Définition 3.2.1 (Contexte mémoire) *Étant donné un ensemble fini de noms de variables V, un ensemble fini de numéros de couches $L \subset \mathbb{Z}$ et $\mathcal{U}$ l'alphabet infini, nous définissions un contexte mémoire M comme une fonction d'association dont la signature est la suivante : $M : V \times L \to 2^{\mathcal{U}}$*

où $M(X^l) \subset \mathcal{U}$ est l'ensemble fini des lettres associées à la variable X^l.

La notion de couche mémoire est d'une grande importance dans le modèle de contexte mémoire que nous utilisons. Ces couches sont contraintes par une propriété d'injectivité entre les variables, il est interdit qu'une lettre soit associée à deux variables différentes d'une même couche.

Définition 3.2.2 (L'injectivité des couches) *Soit un contexte mémoire M défini sur l'ensemble fini de noms de variables V et l'ensemble fini de couches L, la contrainte d'injectivité est définie par :*

$$\forall (X, Y) \in V^2, \forall l \in L, X \neq Y \to M(X^l) \cap M(Y^l) = \emptyset$$

La contrainte d'injectivité donne une certaine aisance lors de la spécification car différentes variables d'une même couche représentent toujours des valeurs différentes. Cette contrainte est similaire au fonctionnement des registres des FMA [55].

L'injectivité dans les couches permet de définir la notion de lettres fraîches pour une couche l, c'est-à-dire les lettres qui ne sont associées à aucune variable de l. Certaines lettres peuvent être fraîches pour certaines couches mais pas pour d'autres. Seules les lettres fraîches pour une couche peuvent être associées à une de ses variables. Lors de la ré-initialisation d'une variable X^l, toutes les lettres qui étaient associées à X^l redeviennent fraîches pour la couche l.

3.2.2 Les langages sur alphabet infini

Les ECM représentent des langages sur un alphabet infini de lettres $\mathcal{U}$ et un alphabet fini Σ. L'alphabet Σ permet de représenter des lettres connues à l'avance dans les mots, alors que l'alphabet infini représente toutes les lettres qui seront représentées à l'aide des variables. Les deux alphabets sont considérés comme disjoints, $\mathcal{U} \cap \Sigma = \emptyset$, afin d'éviter des ambiguïtés sur les lettres représentées par les variables.

Définition 3.2.3 (Langage sur alphabet infini) *Un langage sur alphabet infini est un ensemble, fini ou non, de mots de $(\mathcal{U} \cup \Sigma)^*$, pour un alphabet fini Σ et un alphabet infini $\mathcal{U}$ donnés et disjoints.*

Le langage représenté par une ECM dépend de son contexte mémoire initial M_i, représentant les valeurs initiales des variables, et d'un contexte mémoire final M_f représentant l'état des variables attendu lors de l'évaluation de l'expression. Les deux contextes mémoire doivent être définis sur les mêmes ensembles de variables V et de couches C.

Définition 3.2.4 (Langage d'une ECM) *Soit l'ECM e, le langage représenté par e pour le contexte mémoire initial M_i et le contexte mémoire final M_f est noté $\mathbb{L}_{M_i}^{M_f}(e)$. Il est défini inductivement dans les Tables de sémantique 3.2, 3.3, 3.4 et 3.5. On note $\mathbb{L}_{M_i}(e) = \bigcup_{M_f} \mathbb{L}_{M_i}^{M_f}(e)$, le langage pour tout contexte final.*

3.2.3 La sémantique des expressions terminales

La sémantique des ECM est définie inductivement pour chaque élément de sa syntaxe. Pour présenter la sémantique nous considérons que les expressions sont définies sur les ensembles V de variables, L de couches. Le langage de chaque expression est présenté en fonction d'un contexte mémoire initial M_i et d'un contexte mémoire final M_f, tous deux définis sur V et L. Pour rappel, les expressions définissent des mots sur l'alphabet infini $\mathcal{U}$ et de l'alphabet fini Σ disjoint : $\mathcal{U} \cap \Sigma = \emptyset$.

$$
\begin{array}{lcl}
\mathbb{L}_{M_i}^{M_f}(\emptyset) & = & \emptyset \\
\mathbb{L}_{M_i}^{M_f}(a) & = & \{a \mid a \in \Sigma, M_i = M_f\} \\
\mathbb{L}_{M_i}^{M_f}(X^l) & = & \{s \mid s \in M_i(X^l), M_i = M_f\} \\
\mathbb{L}_{M_i}^{M_f}(\sharp X^l) & = & \{s \mid \begin{array}{l} s \in \mathcal{U} \setminus \bigcup_{Y \in V} M_i(Y^l) \\ \quad M_f(X^l) = M_i(X^l) \cup \{s\}, \\ \quad \forall Y^k \neq X^l, M_i(Y^k) = M_f(Y^k)\}\end{array}
\end{array}
$$

TABLE 3.2 – Les expressions terminales.

La Table 3.2 présente la sémantique des expressions terminales des ECM. L'expression $\emptyset$ exprime le langage vide indépendamment des valeurs des contextes mémoire et des horloges. Celle d'une constante $a \in \Sigma$ représente un mot composé de l'unique lettre a. De même, l'expression d'une variable X^l représente le langage des mots composés d'une lettre qui est associée à X^l dans le contexte mémoire initial. Ces expressions ne peuvent pas modifier les valeurs des variables, ainsi la

contrainte $M_i = M_f$ assure que les contextes initial et final sont identiques. Si le contexte mémoire initial est différent du final, alors le langage représenté est $\emptyset$.

Les expressions suivantes modifient la valeur d'une variable X^l dans le contexte mémoire. Seule la variable spécifiée dans l'expression voit sa valeur modifiée, toutes les autres ont les mêmes valeurs dans le contexte mémoire initial et final : $\forall Y^k \neq X^l, M_i(Y^k) = M_f(Y^k)$. L'expression $\sharp X^l$ représente tous les mots composés d'une seule lettre s, où $s \in \mathcal{U}$ doit être fraîche pour la couche l, c'est-à-dire qu'elle n'est associée à aucune variable de la couche l dans M_i. La lettre du mot reconnu est ensuite associée à X^l, en plus de celles qui y sont déjà associées, dans le contexte mémoire final M_f.

Exemple 3.2.1 (L'ajout d'une lettre fraîche) *Soit l'expression $e_1 = \sharp X^1$ et le contexte mémoire M_1 définis sur l'ensemble de variables $V = \{X, Y\}$ et l'ensemble de couches $L = \{1, 2\}$. Supposons que M_1 soit initialisé ainsi : $M_1 = \{X^1 \to \{a\}, Y^1 \to \{b\}, X^2 \to \{c\}, Y^2 \to \{d\}\}$. Calculons le langage $\mathbb{L}_{M_1}(\sharp X^1)$ en utilisant la Table 3.2.*

La Table 3.2 exprime la sémantique en fonction d'un contexte mémoire final M_f. Comme le langage non contraint par un contexte mémoire final est donné par la formule $\mathbb{L}_{M_i}(e) = \bigcup_{M_f} \mathbb{L}_{M_i}^{M_f}(e)$, il est nécessaire de calculer le langage pour tout contexte mémoire final. La sémantique de l'expression $\sharp X^l$, dans la Table 3.2, indique un ensemble de contraintes définissant les contextes mémoire finaux pertinents, c'est-à-dire ceux qui permettent d'exprimer des langages non vides.

Les contextes mémoire finaux M_f pertinents pour e_1 sont les suivants :

- *les lettres associées à X^1 dans M_f sont les mêmes que celles qui y sont associées dans M_1 plus une unique lettre s ;*
- *les lettres associées aux variables, autres que X^1, sont exactement les mêmes que celles qui leur sont associées dans M_1.*

Ils sont donc de la forme $M_f = \{X^1 \to \{a, s\}, Y^1 \to \{b\}, X^2 \to \{c\}, Y^2 \to \{d\}\}$.

Dans la sémantique, la lettre s est l'unique lettre composant les mots du langage $\mathbb{L}_M^{M_f}(e_1)$. C'est une lettre de $\mathcal{U}$ qui n'est associée à aucune variable de la couche 1 dans M_1, donc une lettre fraîche pour la couche 1 : $s \in \mathcal{U} \backslash \{a, b\}$. Il existe donc un contexte mémoire M_f pertinent pour chaque valeur de s. Si M_f est tel que $s = d$, alors le langage défini contient les mots composés seulement d'une lettre d : $\mathbb{L}_{M_1}^{M_f}(e_1) = \{d\}$. Le langage $\mathbb{L}_{M_1}(e_1)$ est défini pour tout contexte mémoire final, donc pour toutes lettres $s \in \mathcal{U} \backslash \{a, b\}$. Ainsi, le langage de e_1 est $\mathbb{L}_{M_1}(e_1) = \{s \mid s \in \mathcal{U} \backslash \{a, b\}\}$, soit tous les mots composés d'une seule lettre, et dont la lettre n'est ni a ni b.

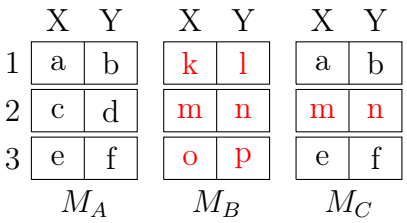


FIGURE 3.1 – Trois contextes mémoire, $M_C = \mathsf{comp}(M_A, \{2\}, M_B)$

3.2.4 Les opérateurs du langage

Les opérateurs définissent des langages en combinant les langages exprimés par leurs sous-expressions. La Table 3.3 représente les opérateurs des ECM qui sont des généralisations des opérateurs des expressions régulières.

$$
\begin{array}{lcl}
\mathbb{L}^{M_f}_{M_i}(e_1 \cdot e_2) & = & \bigcup_{M'} \mathbb{L}^{M'}_{M_i}(e_1) \cdot \mathbb{L}^{M_f}_{M'}(e_2) \\
\mathbb{L}^{M_f}_{M_i}(e_1 + e_2) & = & \mathbb{L}^{M_f}_{M_i}(e_1) \cup \mathbb{L}^{M_f}_{M_i}(e_2) \\
\mathbb{L}^{M_f}_{M_i}(e_1 \,\&\, e_2) & = & \mathbb{L}^{\mathsf{comp}(M_i,\mathsf{layer}(e_1),M_f)}_{M_i}(e_1) \;\cap\; \mathbb{L}^{\mathsf{comp}(M_i,\mathsf{layer}(e_2),M_f)}_{M_i}(e_2) \\
 & \text{si} & \mathsf{layer}(e_1) \cap \mathsf{layer}(e_2) = \emptyset \\
 & & \wedge\; M_f = \mathsf{comp}(M_i, \mathsf{layer}(e_1) \cup \mathsf{layer}(e_2), M_f) \\
\mathbb{L}^{M_f}_{M_i}(e^*/\bar{l}/) & = & \bigcup_{M'} \mathbb{L}^{M'}_{M_i}(e) \cdot \mathbb{L}^{M_f}_{\mathsf{comp}(M_i,\bar{l},M')}(e^*/\bar{l}/) \\
 & & \cup\{\varepsilon \mid M_i = M_f\} \\
\multicolumn{3}{l}{\mathsf{comp}(M, \bar{l}, M_c) = \{X^l \mapsto M_c(X^l) \mid l \in \bar{l}\} \cup \{X^l \mapsto M(X^l) \mid l \notin \bar{l}\}}
\end{array}
$$

TABLE 3.3 – Les opérateurs du langage

Dans la Table 3.3 sont utilisées les fonctions layer et comp. La fonction layer prend en paramètre une expression e, définie sur un ensemble de couches L et renvoie l'ensemble des numéros de couches utilisées dans e, ainsi $\mathsf{layer}(e) \subseteq L$. On considère qu'une couche est utilisée dans une expression si l'expression est composée d'au moins une expression terminale concernant une variable de la couche.

La fonction comp prend en paramètre deux contextes mémoire, M et M_c, ainsi qu'un ensemble fini de couches $\bar{l} \subseteq L$ pour créer le contexte mémoire qui est une composition des couches de M et M_c. Les valeurs des variables du contexte mémoire généré sont les mêmes que dans M_c si leurs couches font partie $\bar{l}$, sinon elles ont les mêmes valeurs que dans M.

Exemple 3.2.2 (Composition de contextes) *Soit les contextes mémoire M_A et M_B définis sur l'ensemble de variables $V = \{X, Y\}$ et l'ensemble de couches*

$L = \{1, 2, 3\}$. *Les valeurs des contextes sont présentées dans la Figure 3.1. Comme ils sont définis sur les mêmes* L *et* V*, on peut les composer et par exemple définir* $M_C = \mathsf{comp}(M_A, \{2\}, M_B)$*, visible dans la Figure 3.1.*

La concaténation

Le symbole $\cdot$ représente la concaténation pour les ECM et pour les langages. La concaténation d'un langage L_1 avec un langage L_2 consiste en l'ensemble des mots pouvant être composés en mettant bout à bout un mot de L_1 (en préfixe) avec un mot de L_2 (en suffixe) : $L_1 \cdot L_2 = \{vw \mid v \in L_1, w \in L_2\}$. L'opérateur de concaténation des ECM n'est pas directement la concaténation des langages de ses sous-expressions car les langages exprimés par les opérandes ne sont pas indépendants. Un *effet de bord* a lieu sur les mémoires des deux expressions. En effet, le contexte mémoire final de l'expression préfixe devient le contexte mémoire initial de l'expression suffixe.

Pour calculer le langage de $e_1 \cdot e_2$ en fonction du contexte mémoire initial M_i et contexte mémoire final M_f, il est nécessaire de calculer les contextes mémoire finaux M' de e_1 qui seront ensuite les contextes initiaux de e_2. Les contextes mémoire M' représentent les effets de bord que produisent les préfixes sur leurs possibles suffixes. Chaque mot $w \in \mathbb{L}_{M_i}^{M_f}(e_1 \cdot e_2)$ est la concaténation de deux mots $w = uv$, pour lesquels il existe un contexte mémoire intermédiaire M', tel que le préfixe u vérifie $u \in \mathbb{L}_{M_i}^{M'}(e_1)$ et le suffixe v vérifie $v \in \mathbb{L}_{M'}^{M_f}(e_2)$.

Exemple 3.2.3 (La concaténation) *Soit l'expression* $e_2 = \sharp X^1 \cdot X^1 \cdot X^1!$*, définie sur l'ensemble de variables* $V = \{X\}$ *et l'ensemble de couches* $C = \{1\}$*. Nous utilisons la Table 3.3 pour déterminer* $\mathbb{L}(e_2)$*, c'est-à-dire le langage de l'expression* e_2 *de contexte mémoire initial* $M_\emptyset$*, le contexte mémoire vide. Il est nécessaire de calculer le langage pour tout contexte mémoire final* M_f*, comme indiqué par la formule* $\mathbb{L}_{M_i}(e) = \bigcup_{M_f} \mathbb{L}_{M_i}^{M_f}(e)$*. En utilisant la sémantique de la concaténation, on décompose le langage de* e_2 *en :*

$$\mathbb{L}_{M_\emptyset}^{M_f}(e_2) = \bigcup_{M'} \mathbb{L}_{M_\emptyset}^{M'}(\sharp X^1) \cdot \bigcup_{M''} \mathbb{L}_{M'}^{M''}(X^1) \cdot \mathbb{L}_{M''}^{M_f}(X^1!)$$

Les contextes mémoire intermédiaires M' *et* M'' *représentent les effets de bord des sous-expressions. Même si ces contextes mémoire ne sont pas contraints, comme pour l'exemple 3.2.1, seuls certains contextes sont pertinents. En effet,* $\emptyset$ *est l'élément absorbant pour la concaténation, ainsi, si l'un des contextes mémoire n'est pas pertinent alors le langage de l'expression* e_2 *sera vide.*

Comme pour l'exemple 3.2.1, on peut déduire que $\mathbb{L}_{M_\emptyset}(\sharp X^1)$ *représente tous les mots composés d'une seule lettre* $s \in \mathcal{U}$ *fraîche pour la couche 1. Le contexte* M' *doit garder les mêmes valeurs que* $M_\emptyset$ *pour toutes les variables qui ne sont pas*

X^1, et associer la lettre s à X^1. Comme le contexte mémoire initial est $M_\emptyset$ alors les valeurs pertinentes pour M' sont de la forme $\{X^1 \to \{s\}\}$ où s est une lettre quelconque de $\mathcal{U}$.

L'expression terminale suivante, X^1, définit un langage avec pour contexte initial M', et pour tout contexte final M''. Cette expression ne modifie pas la valeur du contexte mémoire, donc le seul contexte mémoire pertinent est $M'' = M'$. Le langage de la seconde expression terminale est $\mathbb{L}^{M''}_{M'}(X^1) = \{s\}$, car s est la seule lettre associée à X^1.

La dernière expression terminale, $X^1!$, définit un langage avec pour contexte mémoire initial M'' et contexte mémoire final M_f. On peut déduire de cette expression quels sont les M_f pertinents. Comme l'expression réinitialise la variable X^1, alors son contexte mémoire final est M'' où plus aucune lettre n'est associée à X^1, donc $M_f = M_\emptyset$. Le langage représenté par la dernière expression terminale est $\mathbb{L}^{M_\emptyset}_{M''}(X^1!) = \{s\}$, où s et la lettre associée à X^1 dans M''.

On peut donc en conclure que :

$$\begin{aligned}\mathbb{L}(e_2) &= \mathbb{L}^{M_\emptyset}_{M_\emptyset}(e_2) = \bigcup_{s\in\mathcal{U}}\{s\}\cdot\{s\}\cdot\{s\} \\ &= \{sss \mid s \in \mathcal{U}\}\end{aligned}$$

La disjonction

L'opérateur de disjonction, noté +, représente l'union des langages exprimés par les sous-expressions. La signification de cet opérateur et sa sémantique sont classiques. Le langage représenté par la disjonction de e_1 et e_2 étant donné un contexte mémoire initial M_i et un contexte mémoire final M_f est l'union du langage de e_1 et du langage de e_2 utilisant les mêmes contextes mémoire initiaux et finaux.

La conjonction

L'opérateur de conjonction, noté &, permet d'exprimer l'intersection des langages de deux expressions. L'utilisation de cet opérateur est contrainte dans les ECM, il n'est pas possible de faire la conjonction de deux expressions utilisant les mêmes couches. Cette contrainte permet d'assurer la cohérence du contexte mémoire final en s'assurant que les deux expressions sont indépendantes l'une de l'autre vis-à-vis de la mémoire.

Dans la Table 3.3 est défini le langage de l'expression e_1 & e_2 avec pour contexte mémoire initial M_i et contexte mémoire final M_f. Ce langage est l'intersection de ceux de e_1 et e_2 utilisant les contextes mémoire M_i et M_f. Les langages des sous-expressions sont calculés avec pour contexte mémoire initial M_i et comme contexte mémoire final seulement les couches de M_f qui sont utilisées dans la sous-expression. Ces contextes mémoire finaux sont représentés par les expressions

de la forme : $\mathsf{comp}(M, \mathsf{layer}(e_1), M_f)$, dans le cas de l'expression e_1. Grâce à la fonction comp, les couches qui ne sont pas utilisées par la sous-expression e_1 ont leurs valeurs remplacées par celles de M_i.

La contrainte $M_f = \mathsf{comp}(M_i, \mathsf{layer}(e_1) \cup \mathsf{layer}(e_2), M_f)$ assure que les valeurs des variables des couches mémoires qui ne sont utilisées par aucune des deux sous-expressions sont inchangées.

Exemple 3.2.4 (La conjonction)
Soit l'expression $e_3 = ((\sharp X^1 \cdot X^1)\ \&\ (X^2 \cdot X^2))$ *définie sur l'ensemble de variables* $V = \{X\}$ *et l'ensemble de couches* $C = \{1, 2, 3\}$. *Étant donné le contexte mémoire initial* $M_3 = \{X^1 \to \{a, b, c\}, X^2 \to \{c, d, e\}, X^3 \to \{k\}\}$ *nous souhaitons calculer* $\mathbb{L}_{M_3}(e_3)$ *en utilisant la sémantique de la Table 3.3.*

Tout d'abord, il est nécessaire de vérifier la contrainte syntaxique : les sous-expressions de la conjonction n'utilisent pas les mêmes couches. La sous-expression $\sharp X^1 \cdot X^1$ *utilise uniquement la couche* 1 *et* $X^2 \cdot X^2$ *utilise uniquement la couche* 2. *Donc* e_3 *respecte bien la contrainte syntaxique de la conjonction. Comme pour les exemples précédents, il est nécessaire de calculer le langage pour tout contexte mémoire final* M_f.

La sémantique de la Table 3.3 nous permet de décomposer le langage en :

$$\mathbb{L}^{M_f}_{M_3}(e_3) = \mathbb{L}^{M'_f}_{M_3}(\sharp X^1 \cdot X^1) \cap \mathbb{L}^{M''_f}_{M_3}(X^2 \cdot X^2)$$

où $M'_f = \mathsf{comp}(M_3, \{1\}, M_f)$ *et* $M''_f = \mathsf{comp}(M_3, \{2\}, M_f)$.

Le contexte M'_f *est le résultat de la composition de* M_3 *avec la couche* 1 *de* M_f. *Donc* $M'_f(X^2) = \{c, d, e\}$, $M'_f(X^3) = \{k\}$ *et* $M'_f(X^1) = M_f(X^1)$. *On calcule le langage* $\mathbb{L}^{M'_f}_{M_3}(\sharp X^1 \cdot X^1)$ *pour toute valeur possible de* M_f *et donc pour toutes valeurs possibles de* $M'_f(X^1)$. *On peut décomposer ce langage avec la formule de la concaténation en :* $\bigcup_{M'} \mathbb{L}^{M'}_{M_3}(\sharp X^1) \cdot \mathbb{L}^{M'_f}_{M'}(X^1)$.

On sait que $\mathbb{L}^{M'}_{M_3}(\sharp X^1)$ *est l'ensemble des mots composés d'une lettre s fraîche pour la couche* 1. *On peut donc déduire que tout les* M' *pertinents associent s à* X^1, *donc* $M'(X^1) = \{a, b, c, s\}$ *où* $s \in \mathcal{U} \backslash \{a, b, c\}$, *et les autres variables de* M' *ont les mêmes valeurs que dans* M_3. *On peut donc formaliser :* $\mathbb{L}{M_3}^{M'}(\sharp X^1) = \{s\}$. *Étant donné un* M' *pertinent, on déduit que* $\mathbb{L}^{M'_f}_{M'}(X^1) = \{a, b, c, s\}$ *si* $M' = M'_f$. *Les variables des couches* 2 *et* 3 *sont restées inchangées donc :* $M'(X^2) = M_3(X^2) = M'_f(X^2)$ *et* $M'(X^2) = M_3(X^2) = M'_f(X^2)$. *Donc cette sous-expression nous permet de déterminer que les valeurs pertinentes de* $M'_f(X^1)$ *sont toutes les valeurs possibles de* $M'(X^1)$. *En concaténant les langages on obtient* $\mathbb{L}^{M'_f}_{M_3}(\sharp X^1 \cdot X^1) = \{sa, sb, sc, ss\}$ *qui dépend de la valeur de* $s \in \mathcal{U} \backslash \{a, b, c\}$. *Enfin, par définition de* M'_f, *on détermine que les* M_f *pertinents vérifient* $M_f(X^1) = \{a, b, c, s\}$.

Le langage de la seconde sous-expression dépend de M''_f, la composition de M_3 avec la couche 2 de M_f, donc $M''_f(X^1) = \{a, b, c\}$, $M''_f(X^3) = \{k\}$, et où $M''_f(X^2)$ peut être un ensemble de lettre quelconque. On peut déterminer son langage en décomposant la sous-expression avec la formule de la concaténation : $\mathbb{L}^{M''_f}_{M_3}(X^2 \cdot X^2) = \bigcup_{M''} \mathbb{L}^{M''}_{M_3}(X^2) \cdot \mathbb{L}^{M''_f}_{M''}(X^2)$. La sémantique de l'expression X^2 nous indique seule valeurs pertinente pour M'' est $M'' = M_3$, on peut aussi en déduire que la seul valeur pertinente de M''_f est $M''_f = M'' = M_3$. On peut voir que $\mathbb{L}^{M''}_{M_3}(X^2) = \mathbb{L}^{M''_f}_{M'}(X^2) = \{c, d, e\}$. Donc, on peut déduire que $\mathbb{L}^{M''_f}_{M_3}(X^2 \cdot X^2) = \{cc, cd, ce, dc, dd, de, ec, ed, ee\}$.

Le langage de e_3 est l'intersection des langages de ses sous-expressions :

$$\mathbb{L}^{M_f}_{M_3}(e_3) = \{sa, sb, sc, ss\} \cap \{cc, cd, ce, dc, dd, de, ec, ed, ee\}$$

où $s \in \mathcal{U} \backslash \{a, b, c\}$. Du fait de l'intersection, on peut déduire les valeurs de s permettant d'obtenir une intersection non-vide. Comme les seules lettres utilisées dans la seconde expression sont $\{c, d, e\}$ alors $s \in \{c, d, e\} \cap \mathcal{U} \backslash \{a, b, c\} = \{d, e\}$. On peut aussi en déduire qu'il existe seulement deux M_f pertinentes, un pour chaque valeur de $s \in \{d, e\}$, et qui sont de la forme $M_f = \{X^1 \to \{a, b, c, s\}, X^2 \to \{c, d, e\}, X^3 \to \{k\}\}$. La valeur de $M_f(X^3)$ est contrainte à être égale à celle de $M_3(X^3)$ par la sémantique : toutes les couches qui ne sont pas utilisées par les sous-expressions de la conjonction doivent avoir les mêmes valeurs dans les contextes mémoire initial et final. On peut donc conclure que $\mathbb{L}_{M_3}(e_3) = \{dc, dd, ec, ee\}$.

L'itération

L'opérateur d'itération, noté $*$, est une généralisation de l'étoile de Kleene dans les ECM. Cet opérateur s'applique sur une expression e pour exprimer un langage qui est composé de la concaténation d'un nombre quelconque de mots du langage de e. Or, lorsque deux ECM sont concaténées, un effet de bord a lieu entre les langages concaténés, cette mécanique apparaît aussi dans l'itération de manière plus explicite. C'est le paramètre $/\bar{l}/$ qui représente les couches qui vont subir les effets de bord d'une itération sur l'autre, ce qui fait évoluer le langage à chaque itération. À la fin de chaque itération, les couches absentes de $/\bar{l}/$ sont réinitialisées aux valeurs qu'elles avaient au début de l'itération.

Le langage de l'expression $e^*/\bar{l}/$ est défini récursivement dans la Table 3.3. De manière classique, l'itération définit un langage qui est soit le mot vide, soit une concaténation d'un nombre quelconque de mots de l'expression e. Cependant dans les ECM, le langage de e est modifié pour chaque mot concaténé, dû à l'effet de bord du contexte mémoire.

Le langage de $e^*/\bar{l}/$ est défini pour un contexte mémoire initial M_i et final M_f. Le mot vide ε fait partie du langage $\mathbb{L}^{M_f}_{M_i}(e^*/\bar{l}/)$ seulement lorsque $M_i = M_f$,

de par la définition de la Table 3.3. Un mot non vide $w \in \mathbb{L}_{M_i}^{M_f}(e^*/\bar{l}/)$ est de la forme $w = uv$ tel qu'il existe un contexte mémoire M' où $u \in \mathbb{L}_{M_i}^{M'}(e)$. Le contexte mémoire M' est le contexte mémoire final à la fin d'une itération du langage de e. Pour représenter l'effet de bord d'une itération à l'autre, la fonction comp va créer le contexte mémoire initial pour l'itération suivante qui est composé des couches $/\bar{l}/$ de M' et des autres couches de M_i.

Exemple 3.2.5 (L'itération avec les couches)
Soit l'expression $e_4 = (\sharp X^1 \cdot \sharp X^2)^*/\{1\}/$ *définie sur l'ensemble de variables* $V = \{X\}$ *et l'ensemble de couches* $\{1, 2\}$. *On souhaite calculer le langage de* e_4 *pour le contexte mémoire initial* $M_4 = \{X^1 \to \{a\}, X^2 \to \{b\}\}$. *Le langage est calculé pour tout contexte mémoire final* M_f.

On décompose le langage de e_4 *avec la sémantique de la Table 3.3 :*

$$\mathbb{L}_{M_4}^{M_f}(e_4) = \{\varepsilon \mid M_4 = M_f\} \cup \bigcup_{M^0} \mathbb{L}_{M_4}^{M^0}(\sharp X^1 \cdot \sharp X^2) \cdot \mathbb{L}_{M_4^0}^{M_f}(e_4)$$

où $M_4^0 = \mathsf{comp}(M_4, 1, M_4^0)$ *représente le contexte* M^0 *où la couche* 2 *est réinitialisée à sa valeur dans* M_4.

Le mot ε *appartient à* $\mathbb{L}_{M_4}^{M_f}(e_4)$ *si et seulement si* $M_4 = M_f$ *par la sémantique de la Table 3.3, donc* $M_f = M_4$ *est un contexte mémoire final pertinent.*

Pour déterminer les mots non-vides de $\mathbb{L}_{M_4}(e_4)$, *il faut calculer les langages* $\mathbb{L}_{M_4}^{M^0}(\sharp X^1 \cdot \sharp X^2)$ *en fonction des* M^0 *pertinents. Sans rentrer dans les détails, l'expression* $\sharp X^1 \cdot \sharp X^2$ *représente les mots composés de deux lettres, la première étant fraîche pour la couche* 1 *et la seconde fraîche pour la couche* 2. *Ces deux lettres sont associées respectivement aux variables* X^1 *et* X^2 *dans le contexte mémoire final* M^0. *Ainsi, les contextes* M^0 *pertinents dépendent des valeurs fraîches* $s_0 \in \mathcal{U}\backslash\{a\}$ *et* $s_0' \in \mathcal{U}\backslash\{b\}$, *et sont de la forme* $M^0 = \{X^1 \to \{a, s_0\}, X^2 \to \{b, s_0'\}\}$. *Étant donné un contexte mémoire* M^0 *pertinent,* $\mathbb{L}_{M_4}^{M^0}(\sharp X^1 \cdot \sharp X^2) = \{s_0 s_0'\}$.

Ces mots sont concaténés au langage calculé récursivement $\mathbb{L}_{M_4^0}^{M_f}(e_4)$ *où* $M_4^0 = \{X^1 \to \{a, s_0\}, X^2 \to \{b\}\}$. *Comme précédemment,* $\mathbb{L}_{M_4^0}^{M_f}(e_4)$ *se décompose en* $\{\varepsilon \mid M_4^0 = M_f\} \cup \bigcup_{M^1} \mathbb{L}_{M_0'}^{M_1}(\sharp X^1 \cdot \sharp X^2) \cdot \mathbb{L}_{M_5^1}^{M_f}(e_4)$, *où* $M_4^1 = \mathsf{comp}(M_4, \{1\}, M^1)$. *Comme précédemment, le mot* ε *fait partie du langage seulement si* $M_4^0 = M_f$. *Ce qui signifie que tous les* M_4^0 *sont des contextes pertinents pour* M_f. *Ce qui nous permet de déduire que pour chaque* $s_0' \in \mathcal{U}\backslash\{b\}$ *il existe un* M_4^0 *pertinent et* $\mathbb{L}_{M_4}^{M_4^0} = \{s_0 s_0'\} \cdot \{\varepsilon\} = \{s_0 s_0'\}$.

Observons de manière plus générale le langage à une itération arbitraire. Après k *itérations, nous calculons le langage* $\mathbb{L}_{M_4^k}^{M_f}$ *qui est concaténé en suffixe au langage des itérations précédentes. Le contexte mémoire* M_5^k, *est de la forme* $M_4^k = \{X^1 \to$

$\{a, s_0, s_1, \ldots, s_k\}, X^2 \to \{b\}\}$. *Le langage de* $\mathbb{L}^{M_f}_{M_5^k}(e_4)$ *se décompose en :*

$$\{\varepsilon \mid M_4^k = M_f\} \cup \bigcup_{M^{k+1}} \mathbb{L}^{M^{k+1}}_{M_4}(\sharp X^1 \cdot \sharp X^2) \cdot \mathbb{L}^{M_f}_{M_4^{k+1}}(e_4)$$

où $M_4^{k+1} = \mathsf{comp}(M_4, \{1\}, M^k)$.

La partie $\{\varepsilon \mid M_4^k = M_f\}$ *nous permet de déduire que tous les contextes mémoire de la forme de* M_4^k *sont des valeurs pertinentes pour* M_f*. Donc, étant donné un* M_4^k *en fonction des valeurs de* $s_0, s_1, \ldots s_k$*, alors* $\mathbb{L}^{M_4^k}_{M_4} = \{s_0 s'_0 \ldots s_k s'_k\}$*, où les* $s_0, s_1, \ldots, s_k$ *sont toutes différentes de* a *et des unes des autres, et les* $s'_0, s'_1, \ldots, s'_k$ *sont toutes différentes de* $\{b\}$*.*

La seconde partie définit l'ensemble des mots pour les itérations suivantes. Comme on a pu le voir précédemment, les M^{k+1} *pertinents sont définis pour toutes valeurs fraîches* $s_{k+1} \in \mathcal{U} \backslash M_4^k(X^1)$ *et* $s'_{k+1} \in \mathcal{U} \backslash M_4^k(X^2)$ *et sont de la forme :* $M^{k+1} = \{X^1 \to \{a, s_0, \ldots s_{k+1}\}, X^2 \to \{b, s'_{k+1}\}\}$*. Cela permet de déduire que* $\mathbb{L}^{M^{k+1}}_{M_4^k}(\sharp X^1 \cdot \sharp X^2) = \{x s_{k+1} x' s'_{k+1} \mid (x, x') \in \mathbb{R}_+^2\}$*. Ce langage se concatène avec les langage des itérations précédentes et sera le préfixe des langages des itérations suivantes :* $\mathbb{L}^{M_f}_{M_4^{k+1}}(e_4)$*.*

Ainsi on peut déduire que $\mathbb{L}_{M_4}(e_4)$ *représente l'ensemble des mots de* $\mathcal{U}^*$ *de taille paire, tels que toutes les lettres de position impaire sont différentes les unes des autres et différentes de* a*, et toutes les lettres de position paire sont différentes de* b*. Plus formellement :*

$$\mathbb{L}_{M_4}(e_4) = \{\varepsilon\} \cup \{s_0 s'_0 \ldots s_k s'_k \mid k \in \mathbb{N}, \forall i, s_i \neq a, s'_i \neq b, \forall j \neq i, s_i \neq s_j\}$$

3.2.5 La ré-initialisation de variables

La Table 3.4 présente l'opérateur de ré-initialisation de variable, noté $\overline{X^l}!$. Cet opérateur s'applique en suffixe d'une expression e et prend en paramètre un ensemble de variables, noté $\overline{X^l}$. Cet opérateur ré-initialise les variables de $\overline{X^l}$ dans le contexte mémoire final de e. Ainsi, toutes les variables de $\overline{X^l}$ valent nécessairement $\emptyset$ dans le contexte mémoire final de $e\overline{X^l}!$, noté M_f dans la Table 3.4.

$$\begin{aligned}\mathbb{L}^{M_f}_{M_i}(e\overline{X^l}!) = \; & \{w \mid \; \exists M' \in \mathsf{release}(M_f, \overline{X^l}), w \in \mathbb{L}^{M'}_{M_i}(e) \\ & \qquad \forall X^l \in \overline{X^l}, M_f(X^l) = \emptyset\} \\ \text{où } \mathsf{release}(M, \overline{X^l}) = & \{M' \mid \forall Y^k \notin \overline{X^l}, M'(Y^k) = M(Y^k)\}\end{aligned}$$

TABLE 3.4 – L'opérateur de ré-initialisation

La fonction release prend en paramètre un contexte mémoire M et un ensemble de variables $\overline{X^l}$ et crée l'ensemble des contextes mémoire $\overline{M}$. Chaque $M' \in \overline{M}$ est

une copie de M où tous les sous-ensembles finis de lettres peuvent êtres associés aux variables de $\overline{X^l}$. Les variables qui ne sont pas dans $\overline{X^l}$ doivent être associées aux mêmes lettres que dans M donc $\forall Y^k \notin \overline{X^l}, M'(Y^k) = M(Y^k)$.

Le langage représenté par $e\overline{X^l}!$ est l'union des langages représentés par l'expression e pour le contexte mémoire initial M_i et pour tout contexte mémoire final $M' \in \mathsf{release}(M_f, \overline{X^l})$. Il est nécessaire de faire l'union pour tous ces contextes mémoire car M_f est le contexte mémoire où les variables de $\overline{X^l}$ ont déjà été ré-initialisées. Ainsi, nous utilisons release pour générer les contextes mémoire finaux du langage avant la ré-initialisation.

Exemple 3.2.6 (La ré-initialisation de variables) *Étant donné l'expression* $e_5 = \sharp X^1\{Y^1\}! \cdot (Y^1 + X^1) \cdot \sharp Y^1$ *définie sur l'ensemble de variables* $V = \{X, Y\}$ *et l'ensemble de couches* $C = \{1\}$. *Utilisons les règles de sémantique pour déterminer le langage de* e_5 *pour le contexte mémoire initial* $M_5 = \{X^1 \to \{a\}, Y^1 \to \{b\}\}$ *lui aussi défini sur* C, *et* V. *Pour calculer* $\mathbb{L}_{M_5}(e_5)$ *il est nécessaire de calculer le langage pour tous les contextes mémoire finaux* M_f.

Avec la sémantique de la concaténation, on peut décomposer e_5 *en trois sous-expressions concaténées successivement :*

$$\mathbb{L}^{M_f}_{M_5}(e_5) = \bigcup_{M'} \mathbb{L}^{M'}_{M_5}(\sharp X^1\{Y^1\}!) \cdot \bigcup_{M''} \mathbb{L}^{M''}_{M'}(Y^1 + X^1) \cdot \mathbb{L}^{M_f}_{M''}(\sharp Y^1)$$

La première sous-expression $\sharp X^1\{Y^1\}!$ *définit un langage pour tous les contextes mémoire finaux* M' *et pour le contexte mémoire initial* M_5. *Le langage de cette expression est défini par la sémantique de l'opérateur de ré-initialisation dans la Table 3.4. Cette sémantique indique que les* M' *pertinents satisfont tous* $M'(Y^1) = \emptyset$ *et le langage de cette expression est* $\mathbb{L}^{M'_r}_{M_5}(\sharp X^1)$ *pour tous* $M'_r \in \mathsf{release}(M', \{Y^1\})$. *Par définition de la fonction* release, *les contextes* M'_r *sont des copies du contexte* M' *sauf pour la variable* Y^1 *qui peut être associée à ensemble de lettres quelconque. Or, il reste encore à déterminer quelles sont les valeurs pertinentes de* $M'(X^1)$, *on va donc considérer toutes les valeurs possibles de* $M'_r(C^1)$ *et déterminer lesquelles sont pertinentes en calculant* $\mathbb{L}^{M'_r}_{M_5}(\sharp X^1)$. *Pour calculer le langage* $\mathbb{L}^{M'_r}_{M_5}(\sharp X^1)$ *nous utilisons la sémantique de la Table 3.2, qui nous permet de déterminer que cette expression représente l'ensemble des mots d'une lettre* s, *fraîche pour la couche* 1. *Par définition de* M_5, *nous pouvons déduire que* $s \in \mathcal{U} \backslash \{a, b\}$ *et pour chaque valeur de* s *il existe un* M'_r *associant* s *à* X^1. *Ainsi, les valeurs pertinentes de* M' *sont de la forme* $M' = \{X^1 \to \{a, s\}, Y^1 \to \emptyset\}$ *pour chaque valeur* s *différente de* a *et* b.

Étant donné l'un des contextes mémoire M', *le langage de la seconde sous-expression est défini pour tout contexte mémoire final* M'' : $\mathbb{L}^{M''}_{M'}(Y^1 + X^1)$. *La sous expression est la disjonction des expressions* X^1 *et* Y^1, *et représente l'union de leurs langages :* $\mathbb{L}^{M''}_{M'}(Y^1) \cup \mathbb{L}^{M''}_{M'}(X^1)$. *Les contextes mémoire* M'' *pertinents sont*

l'ensemble des contextes mémoire pertinents pour chacune des sous-expressions. Le langage $\mathbb{L}^{M''}_{M'}(Y^1)$ *est défini comme l'ensemble des mots d'une lettre qui est actuellement associée à* Y^1*, or* $M'(Y^1) = \emptyset$ *donc* $\mathbb{L}_{M'}(Y^1) = \emptyset$*, il n'existe donc pas de contexte mémoire final pertinent pour ce langage. Cependant, le langage* $\mathbb{L}^{M''}_{M'}(X^1)$ *n'est pas vide car* $M'(X^1) = \{a, s\}$*, donc* $\mathbb{L}^{M''}_{M'}(X^1) = \{a, s\}$ *et les contextes finaux pertinents sont* $M'' = M'$*.*

Le langage de la dernière sous-expression $\mathbb{L}^{M_f}_{M''}(\sharp Y^1)$ *est l'ensemble des mots d'une lettre* s'*, fraîche pour la couche* 1 *dans* M''*. Comme* $M'' = M'$*, on peut déduire que* $s' \in \mathcal{U} \backslash \{a, s\}$ *et que les contextes* M_f *pertinents ont la lettre* s' *associée à* Y^1*, et sont donc de la forme* $M_f = \{X^1 \to \{a, s\}, Y^1 \to \{s'\}\}$*. En mettant bout à bout les langages on obtient que le langage de* e_5 *est :*

$$\mathbb{L}_{M_5}(e_5) = \{sas', sss' \mid s \neq a, s \neq b, s \neq s'\}$$

3.2.6 L'opérateur de décalage et les clôtures rationnelles

La sémantique de l'opérateur de décalage de couche mémoire est présentée dans la Table 3.5. Cet opérateur permet de modifier dynamiquement le numéro des couches d'une expression sans en changer le langage. Il est noté $\uparrow_n$ où $n \in \mathbb{Z}$ indique de combien est modifié le numéro de la couche. Ainsi l'expression $(X^1 \cdot \sharp X^2 \cdot X^1) \uparrow_3$ est équivalente à l'expression $X^4 \cdot \sharp X^5 \cdot X^4$. Dans la Table 3.5, la modification des numéros de couches est représentée comme une modification des contextes mémoire initial et final.

$$\mathbb{L}^{M_f}_{M_i}(e \uparrow_n) = \mathbb{L}^{M'_f}_{M'_i}(e) \text{ s.t. } \forall X^l . M_i(X^l) = M'_i(X^{l-n}) \wedge M_f(X^l) = M'_f(X^{l-n})$$

TABLE 3.5 – L'opérateur de décalage de couches

Ainsi l'expression $(X^1 \cdot Y^2) \uparrow_3$ est définie sur l'ensemble de couches $L = \{4, 5\}$ et l'ensemble de variables $V = \{X, Y\}$. Un contexte mémoire initial et final pertinent pour cette expression est $M = \{X^4 \to \{a\} Y^5 \to \{b\}\}$. La sémantique définit le langage $\mathbb{L}^M_M((X^1 \cdot Y^2) \uparrow_3)$ comme $\mathbb{L}^{M'}_{M'}(X^1 \cdot Y^2)$ où $M' = \{X^1 \to \{a\} Y^2 \to \{b\}\}$. La sémantique applique le décalage des couches directement dans les contextes mémoire.

Cet opérateur permet, entre autre, de définir des opérateurs dérivés pour les ECM, tels que les opérateurs rationnels. Ce qu'on appelle ici les opérateurs rationnels sont les opérateurs de concaténation, disjonction, conjonction et l'étoile de Kleene. À la différence des opérateurs éponymes dans les ECM, les opérateurs

rationnels assurent que les langages des sous-expressions sont indépendants les uns des autres. Par exemple, dans le cas de la concaténation rationnelle, il n'y a pas d'effet de bord du langage préfixe au langage suffixe.

$$
\begin{array}{ll}
e_1 \cdot_r e_2 & \stackrel{\mathsf{def}}{=} e_1 \cdot (e_2 \uparrow_{\mathsf{max}(\mathsf{layer}(e_1))-\mathsf{min}(\mathsf{layer}(e_2))}) \\
e_1 +_r e_2 & \stackrel{\mathsf{def}}{=} e_1 + (e_2 \uparrow_{\mathsf{max}(\mathsf{lay}(e_1))-\mathsf{min}(\mathsf{layer}(e_2))}) \\
e^{*_r} & \stackrel{\mathsf{def}}{=} e^{*/\emptyset/} \\
e_1 \,\&_r\, e_2 & \stackrel{\mathsf{def}}{=} e_1 \,\&\, e_2
\end{array}
$$

TABLE 3.6 – Les opérateurs rationnels

La Table 3.6 présente la dérivation en ECM des différents opérateurs rationnels. Ils sont notés avec l'indice r pour éviter les confusions avec leurs équivalents dans les ECM. Certaines de leurs définitions utilisent l'opérateur de décalage de couches et les contextes mémoire initial et final doivent donc prendre en compte les couches réellement utilisées.

L'opérateur de concaténation $\cdot_r$ est dérivé de celui des ECM en s'assurant qu'aucun effet de bord n'ait lieu de l'expression préfixe à l'expression suffixe. Pour ce faire, l'opérateur $\uparrow$ est utilisé pour décaler suffisamment l'ensemble des couches utilisées par l'expression du suffixe afin de s'assurer que les expressions utilisent des variables différentes.

De même, dans la dérivation de l'opérateur de disjonction $+_r$, les couches de l'une des sous-expressions sont décalées afin de s'assurer que les deux n'utilisent pas les mêmes variables. Cette modification est nécessaire dans le cas où $+_r$ est utilisé dans une itération afin que les deux langages soient indépendant l'un de l'autre à chaque itération.

L'opérateur de conjonction rationnelle $\&_r$ est directement dérivé en l'opérateur de conjonction & des ECM. En effet, la conjonction est syntaxiquement contrainte pour obliger les sous-expressions à utiliser des ensembles de couches différents, et donc à être indépendantes.

Enfin l'étoile de Kleene $*_r$, comme pour la concaténation rationnelle, ne permet pas les effets de bord. Cet opérateur se dérive dans les ECM, en utilisant la mécanique symbolisée par $/\bar{l}/$ permettant de sélectionner les couches ayant un effet de bords d'une itération sur l'autre. La dérivation de $*_r$ est simplement $*/\emptyset/$ où l'ensemble de couches $/\bar{l}/$ est vide.

3.3 Les expressions temporisées

Les ECM permettent de spécifier des propriétés sur les environnements ouverts, représentées par des lettres sur un alphabet infini. Afin de représenter des

propriétés temporisées, les ECM sont étendues avec des opérateurs permettant d'exprimer des contraintes temporisées. Nous définissons les expressions temporisées à couches mémoire (ETCM), comme une extension des ECM, permettant de spécifier des langages temporisés sur alphabet infini. La Table 3.7 présente la syntaxe des ETCM, avec l'ajout des trois nouveaux opérateurs utilisés afin d'exprimer les langages temporisés. Ces opérateurs sont une adaptation des opérateurs de compositions d'automates temporisés de [25].

Avant de détailler ces nouveaux opérateurs, nous effectuons un rappel sur la notion de langage temporisé sur alphabet infini. Ensuite, les nouveaux opérateurs sont présentés de manière informelle afin d'illustrer leur fonctionnement. La section se conclut avec la définition formelle de la sémantique des nouveaux opérateurs ainsi que les modifications qui sont apportées à celles des opérateurs préexistants des ECM.

3.3.1 Les langages temporisés sur alphabet infini

Les ETCM expriment des langages temporisés, c'est-à-dire des ensembles de mots temporisés [2]. Les lettres y sont appelées *événements* car chacune d'entre elles est associée à une valeur numérique indiquant l'instant où l'événement a lieu. Nous commençons par la définition des séquences d'événements temporisés (*timed event sequence*, TES) [7]. Ce formalisme permet de représenter les langages temporisés avec une information supplémentaire sur la temporalité des événements apparaissant au même instant.

Définition 3.3.1 (Séquence d'événements temporisés (TES)) *Soit l'alphabet infini d'événements $\mathcal{U}$ et l'alphabet fini d'événements connus Σ, tel que $\mathcal{U} \cap \Sigma = \emptyset$. Une séquence d'événements temporisés (TES) est un élément de $(\mathbb{Q}_+ \cup \mathcal{U} \cup \Sigma)^*$. Les valeurs de $\mathbb{Q}_+$ représentent des durées de temps pendant lesquelles aucun événement n'a lieu, appelés des délais.*

Un exemple de TES est $a\ 3\ b\ c\ 4\ 1\ b\ 3$ où $a, b, c \in \Sigma \cup \mathcal{U}$. Toutes les TES peuvent être ré-écrites sous une forme canonique équivalente où chaque événement est précédé d'un unique délai. Ainsi, la forme canonique de l'exemple précédent est $0\ a\ 3\ b\ 0\ c\ 5\ b$, où les délais successifs sont additionnés et un délai nul est ajouté entre chaque paire d'événements simultanés ($b\ c$ devient $b\ 0\ c$). Il est important de noter que le délai final de la séquence a été supprimé dans la forme canonique car il ne précède aucun événement. Cela signifie que l'on ne fait pas la distinction entre les TES qui diffèrent seulement par leurs délais finaux. Les TES présentées ici diffèrent de celles originalement présentées dans [7], par la présence de l'alphabet infini $\mathcal{U}$. Une TES de longueur finie a une durée qui est la somme des délais précédant les événements qu'elle contient. Étant donné la TES $w = x_1 l_1 x_2 l_2 \ldots x_n l_n$, nous notons $\vec{w} = \Sigma_{k=1}^{n} x_k$ la durée de w.

Définition 3.3.2 (Langage temporisé sur alphabet infini) *Un langage temporisé sur alphabet infini est un ensemble de séquences d'événements temporisés défini à la fois sur un alphabet fini Σ et un alphabet infini $\mathcal{U}$.*

3.3.2 La syntaxe des ETCM

Cette section illustre les nouveaux opérateurs des ETCM présentés dans la Table 3.7. Les ETCM sont une extension stricte des ECM qui expriment des langages temporisés sous forme de TES. Cela implique que tout ECM peut être interprétée comme une ETCM, et représente donc un langage temporisé. Ainsi, une ECM e représente le langage temporisé contenant toutes les TES w dont en retirant les délais on obtient un mot non-temporisé appartenant au langage de e défini par la sémantique des ECM.

Expression $e, e_1, e_2, \ldots ::=$		
	e_{ECM}	(expression ECM)
$\mid$	$\overset{c}{\triangledown}\, e$	(Start)
$\mid$	$e \overset{c}{\ddagger}_I$	(Check)
$\mid$	$e_{\setminus a}$	(Masquage)

TABLE 3.7 – La syntaxe des Expressions Temporisées à Couches Mémoires (ETCM).

Exemple 3.3.1 *Étant donné l'ECM $\sharp X^1 \cdot X^1$, elle représente le langage non-temporisé $\{aa \mid a \in \mathcal{U}\}$. En interprétant cette même expression comme une ETCM, elle représente le langage temporisé des TES de la forme $\{xax'a \mid a \in \mathcal{U}, x, x' \in \mathbb{Q}_+\}$. Chaque lettre des mots du langage est précédée d'un délai. Comme l'expression ne contient aucune contrainte de temps alors les délais peuvent être de durée arbitraire.*

Les opérateurs $\overset{c}{\triangledown}$ et $\overset{c}{\ddagger}_I$, inspiré de [25, 8], permettent de spécifier des contraintes sur la durée des mots. Le symbole $\overset{c}{\triangledown}$ s'applique en préfixe d'une expression afin d'indiquer le début du calcul de la durée pour les contraintes annotées par c. Le symbole $\overset{c}{\ddagger}_I$ est l'opérateur de contrainte de temps. Il s'applique en préfixe d'une expression e pour indiquer que la durée des mots représentées par e, depuis le précédent $\overset{c}{\triangledown}$, est incluse dans l'intervalle I. L'intervalle I est un sous-ensemble continu de $[0, \infty[$.

Exemple 3.3.2 *Par exemple, l'expression* $a\cdot \overset{1}{\triangledown}\ b\cdot c\ \overset{1}{\ddagger}_{[1,1]}\ \cdot d\ \overset{1}{\ddagger}_{[2,2]}$ *représente les séquences d'événements temporisés de la forme* $x_a a x_b b x_c c x_d d$ *où* $x_a, x_b, x_c, x_d \in \mathbb{Q}$ *tel que* $x_b + x_c = 1$ *et* $x_b + x_c + x_d = 2$*. Il n'y a aucune contrainte sur le délai* x_a *précédant l'événement* a*.*

L'identifiant c, en exposant des deux opérateurs, permet d'exprimer des contraintes de temps qui se chevauchent sur l'expression, comme dans l'exemple 3.3.3. Ces indices sont l'équivalent des couleurs dans les expressions temporisées équilibrées [8], où encore des horloges dans les automates temporisés.

Exemple 3.3.3 *Par exemple, l'expression* $\overset{1}{\triangledown}\ a\cdot \overset{2}{\triangledown}\ b\ \overset{1}{\ddagger}_{[1,1]}\ \cdot c\ \overset{2}{\ddagger}_{[1,1]}$ *contient deux contraintes de temps croisées. Les mots reconnus par cette expression sont de la forme* $x_a a x_b b x_c c$*. La contrainte de temps d'indice* 1 *indique que la durée de* ab *doit être égale à* 1*, c'est-à-dire* $x_a + x_b = 1$*. La contrainte d'indice* 2 *indique que la durée de* bc *doit être égale à* 1*, c'est-à-dire* $x_b + x_c = 1$*.*

L'initialisation de la mesure de la durée par $\overset{c}{\triangledown}$ peut apparaître plusieurs fois dans une expression pour le même exposant c. Cela peut notamment se produire avec l'utilisation de l'itération $*$ ou de la disjonction $+$. Pour vérifier si une contrainte de temps $\overset{c}{\ddagger}_I$ est satisfaite, on mesure la durée à partir du $\overset{c}{\triangledown}$ le plus proche qui précède $\overset{c}{\ddagger}_I$, ou le début du mot si aucun $\overset{c}{\triangledown}$ ne précède.

Exemple 3.3.4 *En prenant l'exemple de l'expression* $e = a\cdot(b\ \overset{1}{\ddagger}_{[0,1]}\ \cdot\ \overset{1}{\triangledown}\ c)^*/\emptyset/$ *on peut se représenter les contraintes de temps en indiquant dans les mots les symboles* $\overset{c}{\triangledown}$ *et* $\overset{c}{\ddagger}_I$*. Ainsi, les mots de cette expression sont de la forme :* $x_a a x_b b\ \overset{1}{\ddagger}_{[0,1]}\overset{1}{\triangledown}\ x_c c x'_b b\ \overset{1}{\ddagger}_{[0,1]}\overset{1}{\triangledown}\ x'_c c x''_b b\ \overset{1}{\ddagger}_{[0,1]}\ \ldots$

Chaque contrainte $\overset{1}{\ddagger}_{[0,1]}$ *est mesurée par rapport au* $\overset{1}{\triangledown}$ *qui lui précède. La première contrainte n'est précédée par aucun* $\overset{1}{\triangledown}$*, la durée est alors mesurée depuis le début du mot, donc on en déduit que* $x_a + x_b \in [0,1]$*. Les contraintes suivantes sont systématiquement précédées d'un* $\overset{1}{\triangledown}$*, donc on déduit les contraintes* $x_c + x'_b \in [0,1]$*,* $x'_c + x''_b \in [0,1]$*... Le langage de l'expression* e *est donc l'ensemble des séquences d'événements temporisés de la forme* $a(bc)^*$ *tel que le délai séparant deux* b *est toujours inférieur à* 1 *unité de temps.*

Le masquage

Pour des raisons techniques (généralisation du mot vide) on introduit l'opérateur de *masquage* de lettre, noté $e_{\backslash a}$, où $a \in \Sigma$, qui permet de cacher des lettres de

Σ dans le langage représenté par e. Masquer une lettre a dans un mot ou un langage signifie remplacer toutes les occurrences de cette lettre par le mot vide ε dans tous les mots du langage représenté. Cela permet donc de représenter explicitement le mot vide ε dans les ECM.

Exemple 3.3.5 *Par exemple l'expression $((ab)^*/\emptyset/c)_{\setminus b}$, représente le langage de $(ab)^*c$ dont masque la lettre b. Le langage de $(ab)^*c$ est l'ensemble des mots de la forme $ababab\ldots abc$ où le sous-mot ab apparaît un nombre arbitraire de fois, ou voire même absent. Ainsi, $((ab)^*/\emptyset/c)_{\setminus b}$ représente l'ensemble des mots de la forme :*

$$a\varepsilon a\varepsilon \ldots a\varepsilon c = a^*c$$

Le but premier de cet opérateur est de permettre l'expression des contraintes de temps qui débutent ou se terminent entre deux événements. En effet, l'opérateur $\overset{c}{\triangledown}$ commence à mesurer la durée de l'expression à partir de la dernière lettre précédant l'expression sur laquelle il est appliqué. Par exemple, dans l'expression : $a\cdot \overset{1}{\triangledown}\ b\ \overset{1}{\ddagger}_I \cdot c$, la durée mesurée est le délais depuis a. Ainsi, la durée est mesurée à partir de a. De même, $\overset{c}{\ddagger}_I$ indique que la contrainte de temps est mesurée à l'instant de la dernière de l'expression sur laquelle elle est appliquée. Dans l'exemple précédent, on mesure la contrainte à l'instant exact de b. Le masquage permet de faire disparaître des lettres sur lesquelles des contraintes de temps sont définies et donc de faire débuter ou terminer la mesure de la durée entre deux lettres.

La possibilité d'exprimer de telles contraintes temporelles sur des événements *silencieux* accroît l'expressivité des ETCM. En effet, il est démontré dans [17] qu'il existe des contraintes qui ne sont pas exprimables sans cette fonctionnalité.

Exemple 3.3.6 *En reprenant l'exemple de [17], on représente le langage des mots dont les lettres sont séparées par des délais de longueur paire avec l'expression :*

$$((\overset{1}{\triangledown}\ b\ \overset{1}{\ddagger}_{[2,2]})^*/\emptyset/a)^*/\emptyset/_{\setminus b}$$

Les mots de ce langage sont de la forme $x_1ax_2a\ldots$ où $\forall i, x_i \mod 2 = 0$. La première itération de l'expression, $(\overset{1}{\triangledown}\ b\ \overset{1}{\ddagger}_{[2,2]})^*/\emptyset/$ contient la lettre b qui est masquée, et spécifie la possibilité d'accepter plusieurs délais de 2 unités de temps avant une lettre a.

L'opérateur de masquage permet de spécifier des contraintes de temps sur le délai suivant la dernière lettre du mot. Cependant, comme indiqué dans la Section 3.3.1, nous ne faisons pas la distinction entre des mots qui ne diffèrent que par leurs délais finaux. Ainsi, la contrainte de temps sur le délai final permet uniquement d'agir sur sa valeur minimale.

Exemple 3.3.7 *Par exemple, l'expression* $(a^*/\emptyset/\cdot \overset{1}{\nabla} b \overset{1}{\ddagger}_{[2,3]})_{\setminus b}$ *représente, avant le masquage, l'ensemble des séquences d'événements temporisés composé d'une répétition de l'événement* a *un nombre quelconque de fois suivant d'un* b *précédé d'un délai de* 2 *à* 3 *unités de temps. Ainsi, les TES représentées sont celles dont la forme canonique appartient à* $\{x_a a x'_a a \dots x^n_a a x_b b \mid x_b \in [2,3]\}$*. Donc, dans les séquences effectivement représentées, le* b *peut être suivi par un délai quelconque.*

Une fois le masquage pris en compte, le b *est remplacé par* ε *dans toutes les TES du langage précédemment décrit :* $\{x_a a x'_a a \dots x^n_a a x_b \varepsilon \mid x_b \in [2,3]\}$*. Cet ensemble contient toutes les séquences composées d'une répétition de* a *et se terminant par un délai d'au moins* 2 *unités de temps. C'est un cas particulier où la borne maximale de la contrainte de temps ne restreint pas le langage.*

3.3.3 La sémantique des ETCM

La sémantique des ETCM reprend celle des ECM de la Section 3.2 et l'adapte aux langages temporisés. La modification principale de la sémantique est l'ajout des valuations d'horloges en plus des contextes mémoire. Ces valuations fonctionnent de manière similaire aux effets de bord de la mémoire et servent à mesurer la durée des mots pour vérifier si les contraintes de temps sont satisfaites.

Les valuations d'horloges

La notion d'horloge et de valuation temporelle sont des notions classiques dans les automates temporisés. Dans les ETCM, les horloges sont représentées par les identifiants c en exposant des opérateurs $\overset{c}{\nabla}$ et $\overset{c}{\ddagger}_I$. Les horloges permettent de mesurer la durée des mots, et comme dans les automates temporisés, leurs valeurs sont réinitialisées par $\overset{c}{\nabla}$. Les valuations d'horloges sont les structures permettant de représenter les valeurs des horloges.

Définition 3.3.3 (Valuation temporelle) *Étant donné un ensemble* C *d'horloges, la valuation temporelle* $v : C \to \mathbb{Q}_+$ *associe à chaque horloge une valeur positive représentant un temps écoulé.*

Il existe deux opérations classiques sur les valuations d'horloges : l'incrément et la réinitialisation d'horloges. Étant donné une valuation d'horloges v et un délai $x \in \mathbb{Q}_+$, on note par $v + x$ la valuation résultant de l'incrémentation telle que $\forall c, (v + x)(c) = v(c) + x$. De même, la réinitialisation d'une horloge $c \in C$ est représentée par $v[c \leftarrow 0]$, où $v[c \leftarrow 0](c) = 0$ et $\forall c' \neq c, v[c \leftarrow 0](c') = v(c')$.

Le langage d'une ETCM

En plus des ensembles de variables V et de couches L, les ETCM sont définies avec un ensemble d'horloges C. Ce dernier correspond à tous les indices c des opérateurs $\overset{c}{\triangledown}$ et $\overset{c}{\ddagger}_I$ pouvant être présents dans l'expression. Le langage représenté par une ETCM dépend de son contexte mémoire initial M_i, représentant les valeurs initiales des variables, et des valeurs initiales des horloges représentées par la valuation v_i. Le contexte mémoire M_i est défini sur des ensembles de variables et de couches incluant V et L, de même que la valuation est définie sur un ensemble d'horloges incluant C.

La sémantique des ETCM, représentée par les Tables 3.8 et 3.9, est définie pour un contexte mémoire final M_f et une valuation d'horloge finale v_f. Ils contraignent le langage représenté à être l'ensemble des mots dont l'évolution de la mémoire et des horloges permet d'atteindre ces valeurs. Leur représentation est nécessaire afin de représenter la dynamique de la mémoire et des horloges, et représenter la notion d'effet de bord.

Définition 3.3.4 (Langage d'une ETCM) *Le langage temporisé des ETCM est défini inductivement dans les Tables de sémantique 3.8 et 3.9. Le langage d'une ETCM e est un ensemble de TES représenté en fonction : d'un contexte mémoire initial M_i, d'une valuation d'horloges initiale v_i, d'un contexte mémoire final M_f et d'une valuation d'horloge finale v_f et est noté $\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e)$. Cependant, étant donné une expression e, nous définissons son langage uniquement par rapport à un contexte mémoire initial M_i et les valeurs initiales de toutes les horloges sont 0, que l'on représente par la valuation $\boldsymbol{0}$. On note ainsi $\mathbb{L}_{M_i}(e) = \bigcup_{M_f,v_f} \mathbb{L}^{M_f,v_f}_{M_i,\boldsymbol{0}}(e)$.*

3.3.4 La temporisation de la sémantique des ECM

La Table 3.8 reprend la sémantique des ECM et l'adapte à l'expression de langage temporisé. On y distingue principalement l'ajout des valuations d'horloges initiales v_i et finales v_f pour déterminer le langage représenté par une expression. La sémantique des expressions terminales se distingue par l'ajout de délais, potentiellement nuls, précédant chaque lettre des mots. Ces délais apparaissent dans la sémantique des expressions terminales a, X^l et $\sharp X^l$, comme une valeur $x \in \mathbb{Q}_+$. La valuation d'horloges finale v_f des expressions terminales est égale à la valuation initiale incrémentée par le délai précédant la lettre du mot, $v_f = v_i + x$.

La sémantique des opérateurs est très similaire à celle des ECM, les valuations d'horloges sont utilisées de manière très similaire aux contextes mémoire. Par exemple, dans la concaténation $e_1 \cdot e_2$, en plus de déterminer les contextes mémoire intermédiaire M', on détermine aussi les valuations d'horloges v' intermédiaires qui seront les valuations finales de e_1 et initiales de e_2.

$$
\begin{array}{lcl}
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(\emptyset) & = & \emptyset \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(a) & = & \{xa \mid a \in \Sigma, x \in \mathbb{Q}_+, M_i = M_f, v_f = v_i + x\} \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(X^l) & = & \{xs \mid s \in M_i(X^l), x \in \mathbb{Q}_+, M_i = M_f, v_f = v_i + x\} \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(\sharp X^l) & = & \{xs \mid \; s \in \mathcal{U} \backslash \bigcup_{Y \in V} M_i(Y^l), x \in \mathbb{Q}_+ \\
 & & \qquad v_f = v_i + x M_f(X^l) = M_i(X^l) \cup \{s\}, \\
 & & \qquad \forall Y^k \neq X^l, M_i(Y^k) = M_f(Y^k)\} \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e_1 \cdot e_2) & = & \bigcup_{M',v'} \mathbb{L}^{M',v'}_{M_i,v_i}(e_1) \cdot \mathbb{L}^{M_f,v_f}_{M',v'}(e_2) \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e_1 + e_2) & = & \mathbb{L}^{M_f,v_f}_{M_i,v_i}(e_1) \cup \mathbb{L}^{M_f,v_f}_{M_i,v_i}(e_2) \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e_1 \;\&\; e_2) & = & \bigcup_{v_1,v_2} \mathbb{L}^{\mathsf{comp}(M_i,\mathsf{layer}(e_1),M_f),v_1}_{M_i,v_i}(e_1) \;\cap\; \mathbb{L}^{\mathsf{comp}(M_i,\mathsf{layer}(e_2),M_f),v_2}_{M_i,v_i}(e_2) \\
 & \text{si} & \mathsf{layer}(e_1) \cap \mathsf{layer}(e_2) = \emptyset \wedge \mathsf{clock}(e_1) \cap \mathsf{clock}(e_2) = \emptyset \\
 & & \wedge\; M_f = \mathsf{comp}(M_i, \mathsf{layer}(e_1) \cup \mathsf{layer}(e_2), M_f) \\
 & & \wedge\; \forall c \in C, v_f(c) = \mathsf{min}(v_1(c), v_2(c)) \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e^*/\bar{l}/) & = & \bigcup_{M',v'} \mathbb{L}^{M',v'}_{M_i,v_i}(e) \cdot \mathbb{L}^{M_f,v_f}_{\mathsf{comp}(M_i,\bar{l},M'),v'}(e^*/\bar{l}/) \\
 & & \cup\{\varepsilon \mid M_i = M_f, v_i = v_f\} \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e\overline{X^l}!) & = & \{w \mid \; \exists M' \in \mathsf{release}(M_f, \overline{X^l}), w \in \mathbb{L}^{M',v_f}_{M_i,v_i}(e) \\
 & & \qquad \forall X^l \in \overline{X^l}, M_f(X^l) = \emptyset\} \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e \uparrow_n) & = & \mathbb{L}^{M'_f,v_f}_{M'_i,v_i}(e) \\
 & & \text{s.t. } \forall X^l . M_i(X^l) = M'_i(X^{l-n}) \wedge M_f(X^l) = M'_f(X^{l-n})
\end{array}
$$

TABLE 3.8 – La sémantique des expressions temporisées à couches mémoire.

De même pour l'opérateur d'itération $e^*/\bar{l}/$, les valuations d'horloges finales d'une itération sont transmises comme la valuation initiale des itérations suivantes. La mécanique de réinitialisation partielle des variables, représentée par $/\bar{l}/$, n'a pas d'équivalent pour les horloges.

La sémantique de l'opérateur de conjonction $e_1 \& e_2$ ajoute une contrainte supplémentaire sur ses sous-expressions. En plus de ne pouvoir utiliser les mêmes couches, les expressions e_1 et e_2 ne peuvent pas utiliser les même horloges. Cela correspond à la contrainte $\mathsf{clock}(e_1) \cap \mathsf{clock}(e_2) = \emptyset$, où $\mathsf{clock}(e_1) \subseteq C$ représente l'ensemble des horloges utilisées dans e_1, c'est-à-dire l'ensemble des c tel qu'il existe $\overset{c}{\nabla}$ ou $\overset{c}{\ddagger}_I$ dans e_1. Le langage de chaque sous-expression de e_1 & e_2 est calculé avec respectivement pour une valuation d'horloge finale v_1 ou v_2. La valeur d'une horloge dans v_f est sa valeur finale dans le langage de la sous-expression (e_1 ou e_2) qui l'utilise, donc si $c \in \mathsf{clock}(e_1)$ alors $v_f(c) = v_1(c)$. Si une horloge c n'est utilisée dans aucune sous-expression alors sa valeur finale $v_f(c)$ sera $v_i(c)$ incrémentée par la durée des mots du langage. Cette mécanique est représentée par la contrainte

$$
\begin{array}{lll}
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e \overset{c}{\ddagger}_I) & = \{w \mid v_f(c) \in I, w \in \mathbb{L}^{M_f,v_f}_{M_i,v_i}(e)\} \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(\overset{c}{\triangledown}\ e) & = \mathbb{L}^{M_f,v_f}_{M_i,v_i[c\leftarrow 0]}(e) \\
\mathbb{L}^{M_f,v_f}_{M_i,v_i}(e_{\backslash a}) & = \{w'_1 w'_2 \dots w'_n \mid \ w_1 w_2 \dots w_n \in \mathbb{L}^{M_f,v_f}_{M_i,v_i}(e), \\
 & \qquad \forall i, w_i = a \implies w'_i = \varepsilon, w_i \neq a \implies w_i = w'_i\}
\end{array}
$$

TABLE 3.9 – La sémantique des opérateurs de temps

$v_f(c) = \mathsf{min}(v_1(c), v_2(c))$ car la seule possibilité pour que la valeur d'une horloge diffère entre v_1 et v_2 est qu'elle soit réinitialisée dans l'une des sous-expressions. Alors qu'il n'existe qu'une unique valeur possible pour les contextes mémoire des sous-expressions, il peut exister plusieurs valuations d'horloges finales v_1 et v_2, voir l'exemple 3.3.8.

3.3.5 Les opérateurs de contraintes de temps

La sémantique des opérateurs permettant d'exprimer des contraintes de temps donnée dans la Table 3.9. Elle présente la sémantique des trois nouveaux opérateurs des ETCM.

L'opérateur $\overset{c}{\ddagger}_I$ contraint la valeur finale de l'horloge c à être dans l'intervalle I. Cette valeur finale est la durée du mot depuis la dernière réinitialisation de l'horloge, par $\overset{c}{\triangledown}$, ou depuis le début du mot. Ainsi, la sémantique du langage de $e \overset{c}{\ddagger}_I$ consiste à calculer le langage de e et de filtrer tous les mots dont la valuation finale v_f associe à l'horloge c une valeur incluse dans l'intervalle I.

La sémantique de l'opérateur *start* $\overset{c}{\triangledown}\ e$ permet de réinitiliser une horloge c afin de recommencer la mesure des durées pour les contraintes de temps qui contraignent la suite du mot. Dans la sémantique cela consiste simplement à réinitialiser la valeur de l'horloge c dans la valuation d'horloge initiale.

La sémantique du masquage $e_{\backslash a}$ est triviale car elle définit le langage composé de l'ensemble des mots représenté par e avec les mêmes contextes mémoire et valuations où toutes les occurences de la lettre $a \in \Sigma$ sont remplacées par ε.

Exemple 3.3.8 *Pour illustrer la sémantique des contraintes de temps et de la conjonction, nous déterminons le langage de l'expression :*

$$e_6 = a \cdot ((b^*/\emptyset/ \overset{1}{\ddagger}_{[0,10)}) \ \&\ (\overset{2}{\triangledown}\ b \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/)$$

L'expression e_6 est définie uniquement avec des constantes $\Sigma = \{a, b\}$ afin d'illustrer le fonctionnement des valuations d'horloges.

Nous souhaitons ici calculer le langage $\mathbb{L}_{M_\emptyset}(e_6)$*, où* $M_\emptyset$ *est le contexte mémoire dans lequel chaque variable n'est associée à aucune lettre. Ce sera le seul contexte mémoire utilisé dans cette expression car elle ne contient aucune variable,* $V = L = \emptyset$*. L'expression* e_6 *est définie sur l'ensemble d'horloges* $C = \{1, 2\}$*. Le langage de* e_6 *est calculé avec la valuation d'horloges initiale* $\boldsymbol{0}$ *et pour tout contexte mémoire final et toute valuation d'horloge finale* $\mathbb{L}_{M_\emptyset}(e_6) = \bigcup_{M_f, v_f} \mathbb{L}^{M_f, v_f}_{M_\emptyset, \boldsymbol{0}}(e_6)$*. Cependant on sait que le langage ne contient aucune variable, ainsi* $\mathbb{L}_{M_\emptyset}(e_6) = \bigcup_{v_f} \mathbb{L}^{M_\emptyset, v_f}_{M_\emptyset, \boldsymbol{0}}(e_6)$*.*

Ainsi il faut déterminer quelles valuations d'horloges finales sont pertinentes pour l'expression e_6*, c'est-à-dire quelles valeurs de* v_f *définissent un langage non-vide. Par la sémantique de la concaténation on peut décomposer le langage de* e_6 *en*

$$\mathbb{L}^{M_\emptyset, v_f}_{M_\emptyset, \boldsymbol{0}}(e_6) = \bigcup_{v'} \mathbb{L}^{M_\emptyset, v'}_{M_\emptyset, \boldsymbol{0}}(a) \cdot \mathbb{L}^{M_\emptyset, v_f}_{M_\emptyset, v'}((b^*/\emptyset/ \overset{1}{\ddagger}_{[0,10)}) \,\&\, (\overset{2}{\triangledown}\ b \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/)$$

Comme l'expression a ne contient aucune contrainte de temps alors la lettre a peut être précédée par des délais de longueur arbitraire et les valeurs pertinentes de v' *satisfont* $v'(1) = v'(2) = x_a \in \mathbb{Q}_+$ *et* $\mathbb{L}^{M_\emptyset, v'}_{M_\emptyset, \boldsymbol{0}}(a) = \{x_a a\}$*. Cependant, les valeurs de* v' *sont pertinentes si elles permettent de définir un langage non vide pour le suffixe de la concaténation.*

Le suffixe de la concaténation est le langage résultant de la conjonction de deux expressions :

$$\mathbb{L}^{M_\emptyset, v_f}_{M_\emptyset, v'}((b^*/\emptyset/ \overset{1}{\ddagger}_{[0,10)}) \,\&\, (\overset{2}{\triangledown}\ b \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/) = \bigcup_{v_1, v_2} \mathbb{L}^{M_\emptyset, v_1}_{M_\emptyset, v'}(b^*/\emptyset/ \overset{1}{\ddagger}_{[0,10)}) \bigcap \mathbb{L}^{M_\emptyset, v_2}_{M_\emptyset, v'}((\overset{2}{\triangledown}\ b \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/)$$

Il faut alors déterminer toutes les valeurs pertinentes pour les deux valuations d'horloges v_1 *et* v_2*. Pour* v_1 *cela est relativement simple, car l'expression* $b^*/\emptyset/ \overset{1}{\ddagger}_{[0,10)}$ *se conclut par une contrainte de temps et ne contient aucune réinitialisation. Le langage de* $b^*/\emptyset/$ *est l'ensemble* $\{x_1 b x_2 b \dots x_n b \mid \forall i, x_i \in \mathbb{Q}_+\}$*. On peut donc déterminer grâce à la contrainte de temps que :*

$$\mathbb{L}^{M_\emptyset, v_1}_{M_\emptyset, v'}(b^*/\emptyset/ \overset{1}{\ddagger}_{[0,10)}) = \{x_1 b x_2 b \dots x_n b \mid 0 \leq n, 0 \leq x_a + x_1 + \dots + x_n < 10\}$$

Comme les valeurs des horloges sont incrémentées simultanément alors les valeurs de v_1 *pertinentes vérifient* $v_1(1) = v_2(2) = x_a + x_1 + \dots + x_n$*. Ce qui nous permet alors de raffiner les* v' *pertinentes à celles dont la valeur* x_a *des horloges est strictement inférieure à* 10*.*

Les valeurs pertinentes de v_2 *sont déterminées en calculant le langage de la seconde sous-expression de la conjonction. À l'aide de la sémantique de l'itération*

on peut déterminer que

$$\mathbb{L}^{M_\emptyset,v_2}_{M_\emptyset,v'}\left((\overset{2}{\triangledown}\ b\ \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/\right) = \{\varepsilon \mid v' = v_2\} \cup \bigcup_{v''} \mathbb{L}^{M_\emptyset,v''}_{M_\emptyset,v'}(\overset{2}{\triangledown}\ b\ \overset{2}{\ddagger}_{[2,2]}) \cdot \mathbb{L}^{M_\emptyset,v_2}_{M_\emptyset,v''}\left((\overset{2}{\triangledown}\ b\ \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/\right)$$

Une valeur pertinente de v_2 est $v_2 = v'$ et le langage reconnu est $\{\varepsilon\}$. Pour déterminer les autres v_2 pertinentes, il faut calculer les v'' pertinents. Par la sémantique de $\overset{2}{\ddagger}_{[2,2]}$ on sait que les seuls v'' pertinent vérifient $v''(2) = 2$ et tel que $\mathbb{L}^{M_\emptyset,v''}_{M_\emptyset,v'}(\overset{2}{\triangledown}\ b)$ est non vide. À l'aide de l'application de la sémantique de $\overset{2}{\triangledown}$ on obtient que $\mathbb{L}^{M_\emptyset,v''}_{M_\emptyset,v'}(\overset{2}{\triangledown}\ b) = \mathbb{L}^{M_\emptyset,v''}_{M_\emptyset,v'[2\leftarrow 0]}(b) = \{2b\}$ où $v''(1) = x_a + 2$ et $v''(2) = 2$. On peut alors en déduire que les v_2 pertinents sont de la forme $v_2(1) = x_a + 2k$ où $k \in \mathbb{N}^*$ et $v(2) = 2$ et :

$$\mathbb{L}^{M_\emptyset,v_2}_{M_\emptyset,v'}\left((\overset{2}{\triangledown}\ b\ \overset{2}{\ddagger}_{[2,2]})^*/\emptyset/\right) = \{x'_1 b x'_2 b \dots x'_k b \mid \forall i, x'_i = 2\}$$

En effectuant l'intersection des langages on peut observer que le mot ε est accepté dans les deux langages et qu'une valeur pertinente de v_f est $v_f = v'$. De plus, le langage est l'intersection du langage $\{x_1 b x_2 b \dots x_n b \mid x_a + x_1 + \dots + x_n < 10\}$ et du langage $\{x'_1 b x'_2 b \dots x'_k \mid \forall i, x_i = 2\}$. Ce qui correspond aux langages de la forme

$$\{x_1 b x_2 b \dots x_k b \mid k \in \mathbb{N}, x_a + x_1 + \dots x_k < 10 \wedge \forall i \in [1,k], x_i = 2\}$$

où $v_f(1) = v_1(1) = v_2(1) = x_a + 2k$ et $v_f(2) = \mathsf{min}(v_1(2), v_2(2)) = \mathsf{min}(x_a + 2k, 2) = 2$.

On peut donc en conclure que

$$\mathbb{L}_{M_\emptyset}(e_6) = \{x_a a x_1 b x_2 b \dots x_k b \mid x_a + x_1 + \dots x_k < 10 \wedge \forall i \in [1,k], x_k = 2\} \cup \{x_a a \mid x_a \in \mathbb{Q}_+\}$$

3.3.6 Les opérateurs réguliers dans les ETCM

Les langages représentés par les ECM sont clos pour l'union, l'intersection, la concaténation et l'étoile de Kleene, comme présenté dans la Section 3.2.6. Comme les ETCM définissent des langages temporisés, il devient nécessaire à nouveau de montrer comment dériver les opérateurs réguliers dans les ETCM. Le fonctionnement de la mémoire restant le même que dans les ECM, cela consiste à adapter les dérivations de la Table 3.6 pour prendre en compte les délais et contraintes de temps.

$$e_1 \cdot_r e_2 \quad \stackrel{\mathsf{def}}{=} e_1 \cdot \overset{\mathsf{clock}(e)}{\nabla} \left(e_2 \uparrow_{\mathsf{max}(\mathsf{layer}(e_1))-\mathsf{min}(\mathsf{layer}(e_2))}\right)$$
$$e^{*_r} \qquad \stackrel{\mathsf{def}}{=} \left(\overset{\mathsf{clock}(e)}{\nabla} e\right)^{*_\emptyset}$$

TABLE 3.10 – Les opérateurs réguliers

Les opérateurs réguliers de disjonction et de conjonction permettant respectivement de représenter l'union et l'intersection des langages de leurs sous-expressions. Leurs dérivations dans les ETCM est la même que celles de la Table 3.6. Cependant, les dérivations dans les ETCM de la concaténation et de l'étoile de Kleene ont changées, et sont présentées dans la Table 3.10.

Le langage d'une ETCM est défini uniquement par rapport à un contexte mémoire initial. Bien que la sémantique permette de paramétrer $\mathbb{L}$ avec une valuation d'horloges initiale autre que $\mathbf{0}$, cela est seulement utilisé pour représenter les effets de bords à travers les expressions concaténées. La concaténation rationnelle assure l'indépendance des langages concaténés en s'assurant que les contextes mémoire initiaux soient indépendants entre les deux expressions et aussi que leurs valuations d'horloges initiales soient effectivement $\mathbf{0}$. Ainsi la dérivation consiste à réinitialiser ∇ toutes les horloges au début de l'expression e_2, tout en appliquant le $\uparrow$ afin de garder les mémoires indépendantes. Pour l'étoile de Kleene cela consiste aussi à recréer l'indépendance des langages entre chaque itération et à réinitialiser toutes les horloges au début de chacune d'entre elles.

3.4 Les opérateurs dérivés

Pour simplifier l'expression de motifs dans les flots de liens, nous ajoutons quelques éléments à la syntaxe des ETCM, listés dans la Table 3.11. Ces différentes extensions ne modifient pas l'expressivité des ETCM mais sont utiles en pratique.

Terminal $t_1, t_2, \ldots ::= \ldots$
- $@$ (Lettre universelle)
- $X^l!$ (Lire et ré-initialiser)
- $\sharp X^l!$ (Valeur fraîche et ré-initialiser)
- $t_1 \rightsquigarrow t_2$ (Liens)

Expression $e, e_1, e_2, \ldots ::= \ldots$
- $\langle e \rangle_I$ (Contrainte de temps)
- $e_1 \otimes e_2$ (Mélange)

TABLE 3.11 – Extension des ETCM

3.4.1 L'expression terminale universelle

Lors de la spécification d'un langage, il est classique de ne pas contraindre certaines lettres des mots du langage. Par exemple, si l'on définit le langage des mots débutant par a, alors toutes les lettres suivant la première sont arbitraires. Dans notre situation, une telle mécanique est utile pour indiquer l'éventuelle présence de lettres/événements parasites qui apparaissent dans le flux de données lu.

Dans les ETCM, la présence de deux alphabets et la segmentation de la mémoire sous forme de variables et de couches peut compliquer l'expression d'une telle lettre. En effet, une lettre peut être une constante de Σ, une lettre fraîche, ou bien associée à une variable dans une couche. C'est un problème classique dans les langages dérivés des expressions régulières. Ainsi, le symbole "." est couramment utilisé pour représenter un lettre quelconque (exemple : Perl, regexp bash...).

Dans les ETCM nous représentons la lettre universelle avec l'expression terminale @ pour éviter toute confusion avec la concaténation $\cdot$. Étant donné un alphabet fini Σ et un alphabet infini $\mathcal{U}$ disjoints, l'expression @ représente l'ensemble des mots d'une lettre de $\Sigma \cup \mathcal{U}$. Plus formellement, étant donné un contexte mémoire initial M_i et final M_f, tous les deux définis sur l'ensemble de couches L et de variables V, la sémantique de @ est la suivante :

$$\mathbb{L}_{M_i}^{M_f}(@) = \{xs \mid s \in \Sigma \cup \mathcal{U}, M_i = M_f, x \in \mathbb{Q}_+\}$$

L'ajout de cette expression terminale ne modifie pas l'expressivité des ETCM mais évite l'écriture de disjonctions excessivement longues. Soient l'alphabet fini $\Sigma = \{a_1, a_2, \ldots, a_n\}$, la couche $l_@ \in L$ jamais utilisée dans les expressions et initialement vide dans tout contexte mémoire, et $X \in V$. Alors l'expression @ se dérive dans les ETCM de la manière suivante :

Définition 3.4.1 (Dérivation du symbole universel)

$$@ \stackrel{\mathsf{def}}{=} (\sharp X^{l_@}\{X^{l_@}\}!) + a_1 + a_2 + \cdots + a_n$$

L'expression $\sharp X^{l_@}\{X^{l_@}\}!$ représente l'ensemble des mots composés d'une seule lettre de $\mathcal{U} \cup \Sigma$. Le contexte mémoire n'est pas modifié, car soit la lettre lue est associée à $X^{l_@}$ puis la variable est directement réinitialisée, soit il s'agit d'une constante. Comme la couche $l_@$ est spéciale, elle n'est jamais utilisée ailleurs dans les expressions et est initialement vide ; alors toutes les lettres de $\mathcal{U}$ sont fraîches pour cette couche et sont effectivement représentées par $\sharp X^{l_@}$.

3.4.2 La ré-initialisation terminale

La modification de la mémoire, dans les ETCM, est principalement représentée par les symboles '$\sharp$', pour l'association de valeurs fraîches, et '!' pour la réinitialisation. Les utilisations de ces deux symboles sont cependant différentes,

l'une est une expression terminale ($\sharp$), tandis que l'autre est un opérateur qui s'applique en suffixe d'une expression. Ainsi, dans la Définition 3.4.2 sont présentées deux expressions terminales permettant d'exprimer une ré-initialisation d'une variable suite à la lecture de la lettre qu'elle représente.

Définition 3.4.2 (Ré-initialisation terminale)

$$X^l! \quad \stackrel{\mathsf{def}}{=} X^l\{X^l\}!$$
$$\sharp X^l! \quad \stackrel{\mathsf{def}}{=} \sharp X^l\{X^l\}!$$

Ces deux notations permettent de réduire et simplifier des expressions où une variable est ré-initialisée au moment où elle est utilisée. Ces expressions sont dérivables très simplement en ETCM en appliquant l'opérateur de ré-initialisation en suffixe des expressions terminales X^1 et $\sharp X^l$ respectivement. Ces opérateurs terminaux permettent entre autre d'exprimer des liens dont les variables sont ré-initialisées tel que $X! \rightsquigarrow \sharp Y$ (voir Section 3.4.4).

3.4.3 Les contraintes de temps parenthésées

L'opérateur de contrainte de temps, noté $\langle e \rangle_{[a,b]}$, permet d'exprimer une contrainte sur la durée des mots de l'expression e. Elle reprend la syntaxe de l'opérateur des contraintes de temps des *Timed regular expressions* [7] et permet d'abstraire la notion d'horloge et d'alléger l'expression des contraintes de temps. Cela permet entre autre de ne plus avoir d'annotations et d'horloges explicite dans les expressions.

Définition 3.4.3 (La dérivation de la durée) *Étant donné une expression e définie sur un ensembles d'horloges C on dérive :*

$$\langle e \rangle_{[a,b]} \stackrel{\mathsf{def}}{=} \overset{c}{\triangledown}\ e\ \overset{c}{\ddagger}_I$$

où $c \notin C$

La définition 3.4.3 présente comment dériver cet opérateur dans les ETCM. Cela consiste simplement à remplacer dans l'expression $\langle$ par $\overset{c}{\triangledown}$ et $\rangle_I$ par $\overset{c}{\ddagger}_I$ où l'identifiant c est un nouvel identifiant jamais utilisé dans e. Il est important que cet identifiant soit différent afin d'éviter d'interférer avec d'autres contraintes de temps.

3.4.4 L'opérateur de mélange et les *liens*

L'opérateur de mélange $\otimes$ (*shuffle*) permet d'exprimer un langage comme étant le mélange de deux autres langages. Il permet ainsi d'exprimer deux propriétés apparaissant en parallèle, et indépendante l'une de l'autre. Dans les langages rationnels, le langage de $\mathcal{L}_1 \otimes \mathcal{L}_2$ est l'ensemble des mots de la forme $v_1w_1v_2w_2 \dots v_nw_n$ tel que $v_1v_2 \dots v_n \in \mathcal{L}_1$ et $w_1w_2 \dots w_n \in \mathcal{L}_2$. Les langages rationnels sont clos pour cet opérateur.

L'article [6] définit un opérateur de mélange sur les langages temporisés. Or, il est prouvé que l'ensemble des langages reconnaissables par les automates temporisés n'est pas clos pour cet opérateur. En effet, il définit des langages représentables par des *stopwatch automata*, plus expressifs que les automates temporisés [36].

Notre objectif est de proposer une variante plus contrainte du mélange qui reste tout de même utile en pratique, et en même temps préserve la propriété de clôture. Ce nouvel opérateur de mélange pour les langages temporisés est similaire à la composition parallèle présentée dans [2], où il n'y a pas de synchronisations des langages sur les lettres communes des deux alphabets. L'opérateur se base sur la représentation des mots comme des séquences ordonnées de paires $\binom{e_i}{t_i}$ où les e_i sont les identifiants des événements et $t_i \in \mathbb{R}_+$ les numéros des instants où ils ont lieu. Les séquences sont ordonnées sur les numéros des instants. En considérant ce formalisme, il devient possible d'adapter naturellement la définition du mélange de langages rationnels aux langages temporisés. Par exemple, si l'on reprend l'exemple de [41] où $\mathcal{L}_1 = \{\binom{a}{t_1}\binom{a}{t_1+1}\binom{a}{2}\}$ et $\mathcal{L}_2 = \{\binom{b}{1}\binom{b}{s} \mid 1 < s\}$ alors :

$$\begin{aligned}
\mathcal{L}_1 \otimes \mathcal{L}_2 = \quad & \{\tbinom{a}{0}\tbinom{a}{1}\tbinom{b}{1}\tbinom{a}{2}\tbinom{b}{s}\} \cup \{\tbinom{a}{0}\tbinom{a}{1}\tbinom{b}{1}\tbinom{b}{s}\tbinom{a}{2} \mid 1 < s\} \\
\cup \quad & \{\tbinom{a}{t_1}\tbinom{b}{1}\tbinom{a}{t_1+1}\tbinom{a}{2}\tbinom{b}{s}\} \cup \{\tbinom{a}{t_1}\tbinom{b}{1}\tbinom{a}{t_1+1}\tbinom{b}{s}\tbinom{a}{2} \mid 1 < s\} \\
\cup \quad & \{\tbinom{a}{t_1}\tbinom{b}{1}\tbinom{b}{s}\tbinom{a}{t_1+1}\tbinom{a}{2} \mid 1 < s\} \cup \{\tbinom{b}{1}\tbinom{a}{1}\tbinom{a}{2}\tbinom{a}{2}\tbinom{b}{s}\} \\
\cup \quad & \{\tbinom{b}{1}\tbinom{a}{1}\tbinom{a}{2}\tbinom{b}{2}\tbinom{a}{2}\} \cup \{\tbinom{b}{1}\tbinom{a}{1}\tbinom{b}{s}\tbinom{a}{2}\tbinom{a}{2} \mid 1 < s\}
\end{aligned}$$

Le mélange doit respecter les contraintes de temps de chaque langage. Ainsi, l'ordonnancement de la forme *aaabb* n'apparaît pas dans $\mathcal{L}_1 \otimes \mathcal{L}_2$, car tous les mots de $\mathcal{L}_1$ ont une duré de 2 secondes et le premier événement de $\mathcal{L}_2$ apparaît à l'instant 1.

Cet opérateur permet de mélanger les événements de deux mots temporisés, mais pas le temps, qui est commun aux deux mots. Pour les séquences d'événements temporisés, l'opérateur de mélange est défini de la manière suivante :

$$\begin{aligned}
v_1w_1v_2w_2 \dots v_nw_n \in \mathcal{L}_1 \otimes \mathcal{L}_2 \Leftrightarrow \quad & v_1\vec{w_1}v_2\vec{w_2} \dots v_n\vec{w_n} \in \mathcal{L}_1 \\
\wedge \quad & \vec{v_1}w_1\vec{v_2}w_2 \dots \vec{v_n}w_n \in \mathcal{L}_2
\end{aligned}$$

où tous les v_i et w_i, $i \in [O, n]$ sont des TES et, pour rappel, notation $\vec{v_i}$ représente la durée de v_i.

La durée de toutes les TES $u \in v \otimes w$ résultantes du mélange des séquences v et w, ont une durée égale à la plus longue entre v et de w : $\vec{u} = \mathsf{max}(\vec{v}, \vec{w})$. On peut voir le mélange de deux mots temporisés comme l'insertion, aux bons instants des événements de la séquence la plus courte dans la séquence la plus longue. Cependant, comme des événements peuvent arriver aux mêmes instants dans les séquences mélangées, plusieurs TES peuvent en résulter. Par exemple, $a1b2c \otimes a1c3b = \{aa1bc2c1b, aa1cb2c1b\}$ car les événements b et c ont lieu au mêmes instants et les deux ordonnancements sont possibles. Les événements a des deux TES ont lieu à l'instant 0, mais même si ce sont les mêmes événements ils doivent apparaître une fois dans chaque séquence.

Une telle définition s'étend naturellement aux langages sur alphabet infini car elle représente un mélange des événements/lettres des mots. Dans les langages rationnels, le mélange peut être défini au niveau des expressions régulières de la façon suivante (cf.[76]) :

$$\varepsilon \otimes e = e \otimes \varepsilon = e$$
$$(e_1 \cdot e_2) \otimes (e_3 \cdot e_4) = e_1 \cdot (e_2 \otimes (e_3 \otimes e_4)) + (e_3 \cdot ((e_1 \cdot e_2) \otimes e_4)$$

où $e, e_i, (i \in \{1, 2, 3, 4\})$ sont des expressions. Avec une telle définition dérivée on peut supposer qu'il y a un effet de bord de la mémoire lors des différentes concaténations. Les langages mélangés ne sont alors plus indépendants, mais il est possible avec l'opérateur de décalage de couche de rendre indépendantes les sous-expressions, de manière similaire à ce qui est présenté dans la Table 3.6 pour définir la concaténation rationnelle dans les ECM.

Propriété 3.4.1 (Clôture pour le mélange) *Les langages exprimables par les ETCM sont clos pour le mélange, ($\otimes$).*

La preuve de cette propriété est effectuée directement sur le modèle d'automate servant de formalisme de reconnaissance aux ETCM et elle est présentée dans la Section 4.4.3. Il est envisageable de définir une dérivation du mélange dans les ETCM, similaire à celle de [76], en utilisant le décalage de couches $\uparrow_n$ pour assurer l'indépendance des expressions. Cependant, comme le principe de dérivation des ETCM n'a pas encore pu être défini, une telle définition du mélange serait incomplète. En effet, sans principe de dérivation il n'est pas trivial de déterminer l'ensemble des préfixes des expressions qui sont mélangées, spécialement s'il est défini avec une conjonction &.

La représentation des liens

L'opérateur de mélange peut être problématique lorsque l'on souhaite exprimer des langages contenant des n-uplets, par exemple sur les flots de liens. On ne

souhaite pas que deux n-uplets soient mélangés ensembles lorsqu'ils apparaissent au même instant. Considérer un n-uplet comme un seul et même symbole n'est pas une solution pertinente dans le cas des ETCM car on ne pourrait pas comparer entre eux les éléments qui le composent. Pour représenter des n-uplets nous définissons l'opérateur $\rightsquigarrow$ permettant de construire une expression terminale à partir d'autres expressions terminales.

Le mélange fonctionne en entrelaçant les expressions terminales. L'opérateur de lien est défini comme terminal, ainsi le mélange ne peut pas séparer les deux terminaux liés ensemble. L'opérateur de lien se traduit en ETCM en tant que concaténation des lettres composant le n-uplet avec une contrainte de temps sur tous les éléments du n-uplet indiquant que les événements arrivent durant le même instant.

On peut formellement l'écrire ainsi :

$$t_1 \rightsquigarrow t_2 \stackrel{\mathsf{def}}{=} \langle t_1 \cdot t_2 \rangle_{[0,0]}$$

Un exemple d'utilisations du lien et du mélange ensemble est :
$(a \rightsquigarrow b \cdot c \rightsquigarrow d) \otimes (e \rightsquigarrow f)$ qui représente le langage $\{abcdef, abefcd, efabcd\}$.

3.5 Exprimer des motifs sur les flots de liens

Dans cette section nous illustrons l'utilisation des ETCM en représentant des motifs et propriétés dans le modèle des flots de liens. Nous nous intéressons à des flots de liens représentant des historiques de connexions internet où chaque lien représente une connexion directe entre deux entités représentées par leurs adresses IP.

Le ping

Un comportement très classique que l'on retrouve sur internet est le *ping* de serveur. Le *ping* correspond généralement à des connexions chroniques d'un ordinateur vers un serveur afin d'exécuter régulièrement une tâche. Un exemple classique est la synchronisation d'une boîte e-mail, où le gestionnaire de courrier éléctronique consulte régulièrement le serveur pour vérifier si de nouveaux *e-mails* ont été reçus. Dans les ETCM il est possible de représenter un tel scénario avec l'expression suivante :

$$(\sharp X^1 \rightsquigarrow \sharp Y^1 \cdot (\langle X^1 \rightsquigarrow Y^1 \rangle^*_{[5,5]} / \emptyset /)) \otimes (@ \rightsquigarrow @)^* / \emptyset /$$

L'expression est composée de deux sous-expressions mélangées ensembles par $\otimes$. La sous-expression $(@ \rightsquigarrow @)^* / \emptyset /$ représente tout mot composé de liens, et sert

à représenter toutes les connections du réseaux qui ne participent pas au motif recherché, aussi appelées *parasites*. Il est nécessaire de représenter les parasites afin d'être capable d'exprimer des propriétés dans les réseaux réels.

La seconde sous-expression, $\sharp X^1 \rightsquigarrow \sharp Y^1 \cdot (\langle X^1 \rightsquigarrow Y^1 \rangle_{[5,5]})^* / \emptyset /$, correspond au *ping*. Il représente un lien se répétant précisément toutes les 5 unités de temps à partir de sa première apparition. Cette version courte du motif représente un *ping* qui apparaît au minimum une fois. On peut dérouler des itérations pour spécifier un nombre minimal de *pings* nécessaire à la reconnaissance d'une instance.

Le triangle

Le motif dit du triangle est élémentaire dans les réseaux et les graphes [62, 72]. Dans les réseaux sociaux, le triangle est une communauté élémentaire qui correspond à une composante fortement connexe dans un graphe dynamique. Dans le domaine de la sécurité, le triangle peut représenter une symptôme d'attaque par déni de service (Distributed Denial of Service, DDoS), deux utilisateurs inconnus se concertent pour attaquer simultanément une même cible. L'ETCM suivante représente le motif du triangle :

$$((\sharp X^1 \rightsquigarrow \sharp Y^1) \cdot ((X^1 \leftrightsquigarrow \sharp Z^1 \cdot Y^1 \leftrightsquigarrow Z^1) + (Y^1 \leftrightsquigarrow \sharp Z^1 \cdot X^1 \leftrightsquigarrow Z^1)) \otimes (@ \rightsquigarrow @)^* / \emptyset /$$

où $X^1 \leftrightsquigarrow Z^1$ est un raccourci pour $X^1 \rightsquigarrow Z^1 + Z^1 \rightsquigarrow X^1$.

Cette expression est le mélange de la sous-expression $(@ \rightsquigarrow @)^* / \emptyset /$ avec l'expression représentant le triangle. La sous-expression représente ici aussi la présence des liens parasites dans le réseau.

L'expression du triangle débute par la reconnaissance de sa première arête, $\sharp X^1 \rightsquigarrow \sharp Y^1$, qui permet de reconnaître les deux premiers nœuds qui composent le triangle (X^1 et Y^1). Les deux autres arêtes du triangle sont connectées au troisième nœud qui le compose Z^1. Ces arêtes peuvent apparaître dans n'importe quel ordre et sens (vers Z^1 ou depuis Z^1), ainsi les opérateurs $\leftrightsquigarrow$ et $+$ permettent de couvrir les différentes possibilités. La première utilisation de Z^1 est une lettre fraîche, ainsi lors de la reconnaissance de la première des arêtes le nœud sera associé à Z^1.

Dans le cas du DDoS, le motif du triangle est orienté et contient une contrainte de temps forte pour s'assurer que les attaques sont simultanées. L'ETCM correspondante est la suivante :

$$(\sharp X^1 \rightsquigarrow \sharp Y^1 \cdot \langle (X^1 \rightsquigarrow \sharp Z^1 \cdot Y^1 \rightsquigarrow Z^1) + (Y^1 \rightsquigarrow \sharp Z^1 \cdot X^1 \rightsquigarrow Z^1) \rangle_{[0,1]}) \otimes (@ \rightsquigarrow @)^* / \emptyset /$$

Dans cette version du triangle, la première arête correspond à la coopération entre les deux attaquants, X^1 et Y^1. Les deux arêtes suivantes sont les attaques

simultanées vers la cible Z^1. La simultanéité est précisée par la contrainte de temps de 0 à 1 unités de temps.

$$(\sharp O^1 \rightsquigarrow \sharp S^1) \cdot (O^1 \rightsquigarrow \sharp S^1)^*/\{1\}/ \\ \cdot \langle ((S^1 \rightsquigarrow \sharp T^1)\&(\sharp H^2 \rightsquigarrow @)) \cdot ((S^1 \rightsquigarrow T^1)\&(\sharp H^2 \rightsquigarrow @))^*/\{1,2\}/\rangle_{[0,2]}$$

Cette dernière expression représente un motif général pour le DDoS, qui ressemble plus à un diamant qu'à un triangle. On représente ici le DDoS par un organisateur O^1 qui va ordonner à tous ses servants S^1 d'attaquer une même cible T^1 simultanément. La première partie de l'expression représente la coordination, on y découvre qui est l'organisateur ainsi que ses servants. Le premier lien de l'expression représente O^1 envoyant un message à son premier servant S^1. Ce lien permet d'identifier l'organisateur dans O^1 que l'on va supposer unique dans ce motif. Ainsi, dans l'itération qui suit, l'organisateur est toujours le même symbole qui envoie des messages à différents servants $\sharp S^1$, qui sont tous identifiés dans S^1.

La seconde partie de l'expression représente les attaques simultanées. Sa durée est donc contrainte afin de représenter la simultanéité (l'intervalle $[0, 2]$ a ici été choisi arbitrairement petit). Cette partie est composée d'une première conjonction permettant d'identifier la cible de l'attaque dans la variable T^1. Puis elle est suivie d'une itération sur la même conjonction où la cible est déjà identifiée ainsi la variable T^1 est simplement utilisée. La première expression de la conjonction est $S^1 \rightsquigarrow T^1$ qui représente l'attaque d'un servant, et la seconde est $H^2 \rightsquigarrow @$ qui mémorise quels servants ont attaqué la cible. La variable H^2 sert d'historique et permet de savoir quels servants ont effectivement attaqué la cible. La seconde itération peut être déroulée plusieurs fois pour représenter un DDoS contenant un nombre minimum de servants et d'attaques.

Les chemins

Le problème de la recherche de chemin entre deux nœuds est un classique dans les graphes. Dans les graphes dynamiques, en plus de l'adjacence des arêtes, il est nécessaire de vérifier qu'elles apparaissent dans le bon ordre. C'est-à-dire que l'ordre d'apparition des arêtes est primordial, il faut que les arêtes du début du chemin soit présentes avant celles de la fin. Un tel motif représente une vaste variété de scénarios de communications longue distance dans les réseaux, par exemple l'envoi d'un courrier électronique traversant différents routeurs. Il est possible de représenter ce motif avec l'ETCM suivante :

$$((a \rightsquigarrow b) + (a \rightsquigarrow \sharp X^1 \cdot (X^1 \rightsquigarrow \sharp X^1)^*/\{1\}/ \cdot X^1 \rightsquigarrow b)) \otimes (@ \rightsquigarrow @)^*/\emptyset/$$

Cette expression représente le motif d'un chemin du nœud a au nœud b, tous les deux connus. Il est composé de la possibilité d'une connexion directe entre les deux nœuds, $a \rightsquigarrow b$, ou bien de la présence de différents nœuds intermédiaires inconnus, tous représentés par X^1. Le premier nœud intermédiaire est nécessairement accédé via a et est associé à X^1, cela est représenté par l'expression $a \rightsquigarrow \sharp X^1$. Ensuite, il est possible d'accéder à un nombre quelconque d'autres nœuds intermédiaires via ceux associés à X^1. Chaque nouveau nœud reconnu est ensuite associé à la variable X^1, qui représente donc l'ensemble des nœds accessibles depuis a. Le chemin est alors reconnu lorsque l'un des nœuds intermédiaires accède à b, via $X^1 \rightsquigarrow b$.

Ce motif de chemin reconnaît tous les chemins de a à b, même ceux contenant des cycles, mais représente aussi tous les nœuds atteignables depuis a et les associe à X^1. Ainsi, il est difficile d'obtenir du contexte mémoire final les nœuds intermédiaires qui font effectivement partie du chemin. La seule information que l'on obtient est qu'il existe effectivement un tel chemin. Dans l'optique d'obtenir les nœuds composant le chemin, nous modifions l'expression de la façon suivante :

$$(@ \rightsquigarrow @)^*/\emptyset/\otimes$$
$$((a \rightsquigarrow b) + ((a \rightsquigarrow \sharp X^1 \& a \rightsquigarrow \sharp Y^2) \cdot (X^1 \rightsquigarrow \sharp X^1 \& Y^2! \rightsquigarrow \sharp Y^2)^*/\{1,2\}/ \cdot X^1 \rightsquigarrow b))$$

Ce motif diffère par l'ajout de deux conjonctions dans la sous-expression représentant les chemins contenant des nœuds intermédiaires. Ces conjonctions introduisent la variable Y^2, qui est définie sur une couche différente de la variable X^1. Les deux variables de l'expression sont toujours utilisées simultanément pour représenter les nœuds intermédiaires du chemin. La variable X^1 accumule les valeurs des nœuds intermédiaires et assure une progression du chemin. À chaque itération de $X^1 \rightsquigarrow \sharp X^1$, une nouvelle arête du chemin est reconnue, son origine est n'importe quel nœud intermédiaire déjà connu et sa destination est un nouveau nœud intermédiaire, différent de tous ceux reconnus précédemment. Cette expression est en conjonction avec $Y^2! \rightsquigarrow \sharp Y^2$, qui assure que la première lettre d'une itération est la dernière de la précédente. En effet, Y^2 est toujours associée uniquement au dernier nœud atteint dans le chemin. Ainsi, la conjonction de ces deux expressions représente les liens dont l'origine est le dernier nœud qui a été atteint par le chemin et la destination est un nœud qui n'a jamais été visité auparavant.

Cette nouvelle expression représente l'ensemble des chemins acycliques allant de a à b, où X^1 est associée à tous les nœuds composant le chemin dans le contexte mémoire final.

3.6 Synthèse

Les expressions temporisées à couches mémoire (ETCM) permettent la spécification de langages temporisés sur alphabet infini. Ce langage de spécification

est dérivé des expressions régulières et permet l'utilisation d'une généralisation des opérateurs rationnels pour définir des motifs dynamiques dans des environnements ouverts. La spécification de lettres de l'alphabet infini est faite via une mémoire accessible sous la forme de variables. Les contraintes de temps sont spécifiées avec des opérateurs dédiés et de manière orthogonale à la spécification des événements/lettres.

La sémantique des ETCM est définie dans les Tables 3.8 et 3.9. Sa définition est contextualisée par deux structures représentant la mémoire et le temps : le contexte mémoire et la valuation d'horloges. Elles représentent formellement la notion d'effet de bord du langage qui caractérise la dynamique des motifs exprimés. En dépit de l'orthogonalité des concepts, la sémantique des ETCM est rendue complète (voire cryptique) par leur mélange. Dans la Section 3.4, nous définissons plusieurs opérateurs dérivés des ETCM qui permettent de définir un peu plus facilement des motifs sur les flots de liens dans la Section 3.5.

La définition de la sémantique des ETCM comme une construction ensembliste ne permet pas d'expliquer le principe effectif de la reconnaissance. Pour cela, nous devons introduire un formalisme dédié, ce qui est l'objectif du prochain chapitre.

Chapitre 4

Les automates à couches mémoire

Les expressions temporisées à couches mémoire (ETCM), définies dans le chapitre précédent, permettent la spécification de motifs et propriétés sur les environnements ouverts et dynamiques. Dans l'optique de détecter ces motifs dans des jeux de données, nous avons commencé par développer un formalisme de reconnaissance équivalent aux ETCM. Pour le définir, nous nous sommes intéressés, de manière classique, à la théorie des automates car les automates représentent par définition une spécification des étapes de la reconnaissance d'un langage. La richesse de la littérature sur les différentes classes d'automates pour les langages temporisés et les langages à mémoire nous permet de positionner précisément la classe des langages exprimables par les ETCM. Cela va ainsi faciliter la définitions des propriétés du modèle et nous offrir des outils pour estimer les complexités des différentes analyses sur les automates.

Nous avons ainsi développé le modèle des automates temporisés à couches mémoire (ATCM), qui est équivalent aux ETCM. Ce modèle reprend à la fois les caractéristiques des automates temporisés pour être capable de traiter les contraintes de temps, et les caractéristiques des automates à mémoire pour reconnaître les lettres d'un alphabet infini. Dans un premier temps seront présentés les automates à couches mémoire, sans aucune caractéristique temporisée. Cela facilite la caractérisation des propriétés de l'ensemble des langages reconnaissables par ces automates, telles que les propriétés de clôtures. De plus, cela permet de positionner et comparer notre modèle à d'autres classes d'automates à mémoire similaires présentes dans la littérature. Ensuite seront intégrées les composantes des automates temporisés permettant d'exprimer des contraintes de temps. Les caractéristiques des automates temporisés et à mémoire se présenteront de manière assez orthogonale, ce qui permettra entre autre de préserver dans le modèle final les propriétés des deux modèles. Enfin, nous démontrons l'équivalence entre les langages exprimables par les ETCM, décrites au chapitre précédant, et ceux reconnus par les automates temporisés à couches mémoire.

4.1 Les automates à couches mémoire

Intuitivement, un automate à mémoire est une machine à états reconnaissant des mots définis sur un alphabet infini $\mathcal{U}$, où les états disposent d'une mémoire stockant des lettres de $\mathcal{U}$. Les états sont représentés par des paires (q, M) où q est une place de l'automate et M un contexte mémoire stockant dans des variables un sous-ensemble fini de lettres.

Les automates à couches mémoire (ACM) sont une généralisation des ν-automates [31, 19]. La mémoire manipulée est la même que les ETCM, donnée par la Définition 3.2.1. Cette mémoire est régie par une contrainte d'injectivité interdisant deux variables d'une même couche d'être associées aux même lettres. La notion de couche représente donc une segmentation de la mémoire en sous-ensembles indépendants de variables, relâchant la contrainte d'injectivité entre les variables de couches différentes. Cela permet entre autre de simuler plusieurs évolutions de la mémoire en parallèle et de manière indépendante.

Définition 4.1.1 (Les Automates à couches mémoire (ACM))
Les Automates à couches mémoire sont définis par rapport à un alphabet infini de données $\mathcal{U}$ et sont représentés comme des n-uplets de la forme :

$$A = (Q, q_0, F, \Delta, V, L, M_0)$$

Où :
- *Q est l'ensemble fini des places de l'automate,*
- *$q_0 \in Q$ et $F \subseteq Q$ sont respectivement la place initiale et l'ensemble des places finales,*
- *Δ est l'ensemble fini des transitions de l'automate,*
- *V, L sont respectivement l'ensemble fini des variables et l'ensemble fini des couches,*
- *$M_0 : V * L \mapsto 2^{\mathcal{U}}$ est le contexte mémoire initial,*

La structure de la mémoire dans les ACM est la même que celle des ECM (Définition 3.2.1). Ainsi, les ensembles de variables V et de couches L représentant la structure de la mémoire. On y fixe le contexte mémoire initial M_0 indiquant les lettres initialement associées à chaque variable.

À la différence des ETCM qui sont définis à la fois sur $\mathcal{U}$ et l'alphabet fini Σ, les ACM sont uniquement définis sur un alphabet infini. Ainsi il n'y a pas de notion de *lettre constante* dans les ACM. Cependant, le contexte mémoire initial M_0 permet de définir un ensemble fini de lettres connues à l'avance, permettant de simuler les *constantes*, de façon similaire aux FMA.

Définition 4.1.2 (Transition) *Les transitions sont des n-uplets de la forme :*

$$\delta = (q, \nu, \alpha, \overline{\nu}, q') \in \Delta$$

où :
- $q, q' \in Q$ *sont les places d'origine et de destination de* δ,
- $\nu \subseteq 2^{V \times L}$ *est l'ensemble de variables modifiables dans la transition,*
- $\alpha : L \to V \cup \{\sharp\}$ *indique pour chaque couche la variable utilisée par la transition,*
- $\overline{\nu} \subseteq 2^{V \times L}$ *est l'ensemble de variables réinitialisées*

Lors de la lecture d'une lettre il est nécessaire de franchir une transition. La fonction α indique les variables consultées par la transition, au plus une variable pour chaque couche. Si pour une couche l, $\alpha(l) = \sharp$ alors aucune variable de l n'est consultée par la transition.

Lors de la lecture d'une lettre a, la transition est franchissable si, pour chaque variable X^l telle que $\alpha(l) = X$:
- a est associée à X^l et X^l n'est pas modifiable ($X^l \notin \nu$)
- si X^l est modifiable ($X^l \in \nu$), alors a est fraîche pour la couche l (associée à aucune des variables de la couche l).

Seule une lettre *fraîche* peut être associée à une variable X^l, c'est-à-dire que lors de la lecture, la lettre ne doit être associée à aucune variable de la couche l, même pas X^l.

L'ensemble $\overline{\nu}$ est l'ensemble des variables qui doivent être réinitialisées par la transition. Comme pour les ETCM, plus aucune lettre n'est associée aux variables de $\overline{\nu}$ dans l'état atteint par la transition.

Remarque 4.1.1 (ε-transition) *Étant donné une transition* $\delta = (q, \nu, \alpha, \overline{\nu}, q')$*, si pour toutes les couches* $l \in L$*,* $\alpha(l) = \sharp$*, alors* δ *est appelée* ε*-transition. Pour simplifier les notations, nous écrivons directement* $\alpha = \varepsilon$ *dans ce cas là.*

4.1.1 La dynamique des ACM

Un mot appartient au langage d'un ACM s'il existe un chemin de transitions allant de l'état initial à un état final de l'automate. L'état initial d'un ACM est défini comme la paire (q_0, M_0) associant la place initiale q_0 avec le contexte mémoire initial M_0. Les états finaux sont de la forme (q_f, M) où $q_f \in F$ est une place finale et M est un contexte mémoire quelconque. Les états sont reliés les uns aux autres via des transitions représentées par Δ. Il existe une transition de l'état (q, M) à l'état (q', M'), qui peut être franchie lors de la lecture du symbole $c \in \mathcal{U} \cup \{\varepsilon\}$, s'il existe une transition $\delta \in \Delta$ de q à q' où $M' = \sigma(\delta, M, c) \neq \bot$.

Définition 4.1.3 (Passage de transition) *Étant donné l'état (q, M), la fonction σ définit la valeur du contexte mémoire de l'état atteint via la transition $\delta = (q, \nu, \alpha, \overline{\nu}, q') \in \Delta$ lors de la lecture du symbole $e \in \mathcal{U} \cup \{\varepsilon\}$.*

$$\sigma(\delta, M, e) = \begin{cases} \bot & si \ \ \exists l \in L.\alpha(l) = X \wedge X^l \in \nu \wedge \exists Y, e \in M(Y^l) \quad (1) \\ & ou \ \exists l \in L.\alpha(l) = X, X^l \notin \nu \wedge e \notin M(X^l) \quad (2) \\ & ou \ e = \varepsilon \wedge \exists l \in L.\alpha(l) \neq \sharp \quad (3) \\ M' & sinon, \\ & \quad avec \ M'(X^l) = \begin{cases} \emptyset & if \ X^l \in \overline{\nu} \quad (4) \\ M(X^l) \cup \{e\} & if \ \alpha(l) = X \wedge X^l \in \nu \quad (5) \\ M(X^l) & if \ X^l \notin \nu \vee \alpha(l) \neq X \quad (6) \end{cases} \end{cases}$$

La fonction σ est partielle. Nous symbolisons par $\bot$ les cas où la fonction ne renvoie aucun résultat. Cela symbolise qu'il n'est pas possible de franchir la transition δ lors de la lecture de e.

Les cas (1) à (3) indiquent sous quelles conditions une transition ne peut pas être franchie. Pour toutes les couches où la lettre e est associée à une variable, aucune variable de la couche ne doit déjà être associée à e dans M, cas (1). De même, pour toutes les couches où une des variables X^l est consultée mais non modifiable ($X^l \notin \nu$) alors e doit être associée à X^l dans M, cas (2). Enfin le cas (3) représente le franchissement d'une transition lorsque aucun symbole n'est lu ($e = \varepsilon$), alors la transition doit être une ε-transition.

Les cas (3) à (5) définissent les valeurs pour chacune des variables du nouveau contexte. Le cas (3) gère la réinitialisation de la variable, le (4) l'ajout d'une valeur à une variable, enfin (5) est le cas où la valeur de la variable n'est pas modifiée.

Définition 4.1.4 (Franchissement de transition)
Nous notons par $(q, M) \xrightarrow[\delta]{e} (q', M')$ le franchissement de la transition δ depuis l'état (q, M) à l'état (q', M') lors de la lecture du symbole $e \in \mathcal{U} \cup \{\varepsilon\}$ où :

$$M' = \sigma(\delta, M, e) \neq \bot$$

Nous appelons un chemin de transitions une séquence $\delta_1\delta_2 \dots \delta n \in \Delta$ telle que le franchissement successif des ces transitions lors de la lecture d'un mot permet d'aller d'un état (q_1, M_1) à un état (q_{n+1}, M_{n+1}).

Définition 4.1.5 (Langage d'un ACM) *Étant donné un automate à couches mémoire A et $\mathbb{L}(A)$ le langage qu'il reconnaît. Un mot w appartient à $\mathbb{L}(A)$ si et seulement s'il existe un chemin de transitions dont le franchissement, lors de la lecture de w, permet d'aller de l'état initial de A à un état final, formellement noté : $w \in \mathbb{L}(A) \Leftrightarrow w = w_1w_2w_3 \dots w_n \in (\mathcal{U} \cup \{\varepsilon\})^*$ et $\exists \delta_1, \delta_2 \dots \delta_n \in \Delta^n$ tel que $(q_0, M_0) \xrightarrow[\delta_1]{w_1} (q', M_n) \xrightarrow[\delta_2]{w_2} \cdots \xrightarrow[\delta_n]{w_n} (q_n, M_n)$ tel que $q_n \in F$.*

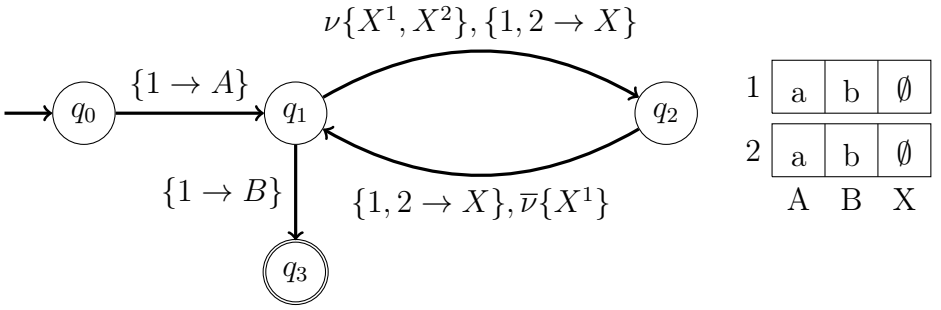


FIGURE 4.1 – Exemple d'automate à couches mémoire

Exemple 4.1.1 (Exemple d'ACM) *L'automate de la Figure 4.1 reconnaît le langage des mots de la forme $ax_1x_1x_2x_2 \dots x_nx_nb$ où tous les x_i sont différents les uns des autres et différents de a et b.*

À la droite de l'automate est représenté le contexte mémoire initial. L'ACM est défini sur deux couches $L = \{1, 2\}$ et sur trois variables $V = \{A, B, X\}$ où les deux variables A et B sont initialisées avec les lettres a et b, respectivement.

L'automate a pour place initial q_0, représentée avec une flèche entrante sans place d'origine, et pour place final q_3, représenté avec un contour doublé. Les transitions sont annotées par les ensembles ν, α et $\overline{\nu}$. Les ensembles ν et $\overline{\nu}$ sont identifiés par leurs symboles respectifs et contiennent les variables modifiables et réinitialisées. Les ensembles ν et $\overline{\nu}$ vides ne sont pas représentés sur les transitions afin d'alléger les notations. Les α sont représentés par des tables d'associations indiquant pour chaque couche la variable consultée. On note par $\{1 \to A\}$ le cas où $\alpha(1) = A$ et $\alpha(2) = \sharp$. On dénote par $\{1, 2 \to X\}$ le cas où $\alpha(1) = \alpha(2) = X$.

Le chemin de transitions allant de la place q_0 à q_1 puis directement à q_3 représente le mot ab sans lettres x intermédiaires. Cependant, les chemins de transitions acceptants peuvent boucler entre les places q_1 et q_2 afin de reconnaître les lettres intermédiaires x_ix_i. La transition allant de q_1 à q_2 va reconnaître une lettre fraîche et l'associer aux variables X^1 et X^2, ainsi la lettre doit être différente de toutes celles déjà associées dans les deux couches. La transition allant de q_2 à q_1 est franchissable lors de la lecture d'une lettre qui est à la fois associée à X^1 et à X^2, puis réinitialise la variable X^1. Ainsi, la variable X^2 va accumuler toutes les lettres intermédiaires x_i afin de s'assurer que la transition allant de q_1 à q_2 lise une lettre différente de toutes celles lues jusqu'à présent. La variable X^1 est toujours associée au plus à une lettre, et permet de s'assurer que la lettre lue pour franchir la transition allant de q_2 à q_1 est celle précédemment lue.

4.2 Les propriétés du modèle

Les automates à mémoire permettent la définition de langages sur un alphabet infini via l'utilisation d'une mémoire. Cette mémoire est similaire à un contexte dont l'évolution de la valeur altère le langage reconnaissable. Bien que ces modèles soient très différents des automates finis, on observe dans la littérature qu'il est possible de retrouver plusieurs de leurs propriétés dans les modèles d'automates à mémoire. Les automates à registres (FMA) [55] définissent ainsi la classe des *langages quasi-rationnels* car plusieurs des propriétés des langages rationnels y sont vérifiées. Entre autre, les propriétés de clôtures pour les opérations rationnelles sont étudiées dans les différentes classes d'automates à mémoire.

Dans l'optique de caractériser l'ensemble des langages reconnaissables par les ACM, et de définir les propriétés du modèle, nous allons le positionner par rapport aux autres modèles d'automates à mémoire. La première partie de cette section présente les propriétés de clôtures des ACM pour les différentes opérations sur les langages : opérateurs rationnels, complément... Nous allons comparer les langages reconnaissables par les ACM à ceux des autres classes d'automates à mémoire afin de se positionner par rapport aux autres modèles de la littérature. Cela permettra entre autre d'en déduire plusieurs propriétés liées à la complexité et à la décidabilité de problèmes classiques de la littérature. Nous nous intéresserons principalement aux problèmes de la reconnaissance d'un mot, c'est-à-dire de déterminer si un mot appartient au langage reconnaissable par un ACM.

4.2.1 La clôture pour les opérations rationnelles

La classe des langages rationnels est définie par les opérations d'union, de concaténation et l'étoile de Kleene. Cette classe est aussi close pour le complément de langages et donc aussi pour l'intersection. Les langages exprimables par les ACM sont clos pour les trois opérations rationnelles ainsi que l'intersection, mais pas pour le complément. Dans cette section sont présentées les constructions permettant de formuler ces propriétés ainsi que leurs preuves.

Dans les différentes preuves sont manipulés deux ACM quelconques $A_1 = (Q_1, q_1, F_1, \Delta_1, V_1, L_1, M_1)$ et $A_2 = (Q_2, q_2, F_2, \Delta_2, V_2, L_2, M_2)$ définis sur des ensembles de couches disjoints, $L_1 \cap L_2 = \emptyset$. Cette supposition n'implique pas de perte de généralité car la dénomination de ses couches n'influe pas sur le langage reconnu par un automate. Il est toujours possible de renommer les couches d'un ACM sans pour autant en modifier le langage.

Propriété 4.2.1 *Les langages représentés par les ACM sont clos pour l'union de langages ($\cup$).*

Preuve : Soit l'automate $A_{1\cup 2}$ construit à partir des automates A_1 et A_2 et reconnaissant $\mathbb{L}(A_1) \cup \mathbb{L}(A_2)$ comme :

$$A_{1\cup 2} \stackrel{\mathsf{def}}{=} (Q_1 \cup Q_2 \cup \{q_0\}, q_0, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \cup \Delta', V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2, \mathcal{U})$$

où $\Delta' = \{(q_0, \emptyset, \varepsilon, \emptyset, q_i) \mid i \in [1, 2])\}$.

Cette construction classique des automates finis consiste à garder la structure des deux automates et permettre d'accéder de manière non-déterministe à la place initiale de l'un des automates. Pour ce faire, une nouvelle place $q_0 \notin Q_1 \cup Q_2$ est ajoutée, qui sera la place initiale de $A_{1\cup 2}$ et deux ε-transitions allant de q_0 vers les places q_1 et q_2. Les places finales sont l'union de celles des automates A_1 et A_2. Comme les ensembles de couches L_1 et L_2 sont disjoints, le contexte mémoire initial de $A_{1\cup 2}$ est simplement $M_1 \cup M_2$ car chaque variable existe uniquement dans l'un des contextes mémoire.

Il est trivial de voir que l'automate $A_{1\cup 2}$ représente le langage $\mathbb{L}(A_1) \cup \mathbb{L}(A_2)$. En effet, les deux automates A_1 et A_2 sont reproduits tels quels dans $A_{1\cup 2}$, n'interagissent pas entre eux, et fonctionnent sur des parties indépendantes de la mémoire. □

Propriété 4.2.2 *Les langages représentés par les ACM sont clos pour l'intersection de langages ($\cap$).*

Preuve : L'automate $A_{1\cap 2}$, construit à partir de A_1 et A_2, reconnaît l'intersection des langages $\mathbb{L}(A_1)$ et $\mathbb{L}(A_2)$ et est construit en adaptant la construction du produit synchrone des automates finis.

$$A_1 \cap A_2 \stackrel{\mathsf{def}}{=} (Q_1 \times Q_2, (q_{0_1}, q_{0_2}), F_1 \times F_2, \Delta_{1\times 2} \cup \Delta_{1\varepsilon} \cup \Delta_{2\varepsilon}, V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2, \mathcal{U})$$

L'ensemble des places de $A_1 \cap A_2$ est l'ensemble des paires $\{(q_1, q_2) \mid q_1 \in Q_1, q_2 \in Q_2\}$. Ainsi la place initiale est la paire dont les deux éléments sont les places initiales de A_1 et A_2 et les places finales sont les paires dont les deux éléments sont des places finales dans leurs automates respectifs. Le contexte mémoire initial est l'union des deux contextes mémoire initiaux, comme dans le cas de l'union.

$$\Delta_{1\times 2} = \left\{ ((q_1, q_2), \nu_1 \cup \nu_2, \alpha', \overline{\nu}_1 \cup \overline{\nu}_2, (q_1', q_2')) \;\middle|\; \begin{array}{l} (q_1, \nu_1, \alpha_1, \overline{\nu}_1, q_1') \in \Delta_1, \alpha_1 \neq \varepsilon \\ (q_2, \nu_2, \alpha_2, \overline{\nu}_2, q_2') \in \Delta_2, \alpha_2 \neq \varepsilon \end{array} \right\}$$

avec $\alpha'(l) = \begin{cases} \alpha_1(l) & \text{if } l \in L_1 \\ \alpha_2(l) & \text{if } l \in L_2 \end{cases}$

L'ensemble de transitions $\Delta_{1\times 2}$ représente la synchronisation des transitions des deux automates. Les ε-transitions ne sont pas prises en compte ici car elles sont silencieuses, ne nécessitent pas la lecture d'une lettre pour être franchies, et ne sont donc pas synchronisées. Comme les couches des automates sont disjointes, leurs fonctionnements respectifs vont être simulés sur leurs couches propres. Pour chaque paire de transitions $(q_1, \nu_1, \alpha_1, \overline{\nu}_1, q_1') \in \Delta_1$, $(q_2, \nu_2, \alpha_2, \overline{\nu}_2, q_2') \in \Delta_2$ on va créer une transition de la place (q_1, q_2) à la place (q_1', q_2'), contenant l'union des opérations mémoire de chacune des transition.

$$\Delta_{1\varepsilon} = \{(q, q_2), \nu, \varepsilon, \overline{\nu}, (q', q_2) \mid (q, \nu, \varepsilon, \overline{\nu}, \pi, q') \in \Delta_1, q_2 \in Q_2\}$$
$$\Delta_{2\varepsilon} = \{(q_1, q), \nu, \varepsilon, \overline{\nu}, (q_1, q') \mid (q, \nu, \varepsilon, \overline{\nu}, \pi, q') \in \Delta_2, q_1 \in Q_1\}$$

Les ensembles $\Delta_{1\varepsilon}$ et $\Delta_{2\varepsilon}$ d'ε-transitions reprennent toutes les ε-transitions des automates A_1 et A_2. Comme elles ne sont pas synchronisées, elles sont reprises telles quelles et modifient uniquement la place de leur automate d'origine.

Si un mot w appartient au langage $\mathbb{L}(A_{1\cap 2})$, alors par définition il existe un chemin de transitions dans l'automate $A_{1\cap 2}$ acceptant chaque lettre du mot. Chaque transition de ce chemin va simuler les actions des automates A_1 et A_2 sur leurs couches respectives. Ainsi il existera nécessairement un chemin de transitions acceptant w dans chacun de ces automates. De même, si un mot w est accepté dans les deux automates A_1 et A_2, alors il existe un chemin de transitions dans les deux automates allant de leurs états initiaux vers leurs états finaux, et il existe dans $A_{1\cap 2}$, la synchronisation de ces chemins qui accepte w. □

Propriété 4.2.3 *Les langages représentés par les ACM sont clos pour la concaténation de langages $(\cdot)$.*

Preuve : L'automate $A_{1\cdot 2}$, construit à partir de A_1 et A_2, représente le langage $\mathbb{L}(A_1) \cdot \mathbb{L}(A_2)$:

$$A_{1\cdot 2} \stackrel{\mathsf{def}}{=} (Q_1 \cup Q_2, q_1, F_2, \Delta_1 \cup \Delta_2 \cup \Delta_{1\cdot 2}, V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2)$$

Cette construction est similaire à celle de la concaténation pour les automates finis, elle contient toutes les places et transitions des automates A_1 et A_2 afin de pouvoir simuler leurs dynamiques. Sa place initiale est q_1, la place initiale de A_1, et ses places finales sont F_2, les places finales de A_2. Comme pour les constructions précédentes, le contexte mémoire initial est la concaténation de ceux des automates A_1 et A_2, et comme ils utilisent des ensembles de couches différents, ils assurent que $A_{1\cdot 2}$ simule les automates concaténés de manière indépendante.

$$\Delta_{1\cdot 2} = \{(q, \nu, \alpha, \overline{\nu}, q_2) \mid (q, \nu, \alpha, \overline{\nu}, q_f) \in \Delta_1, q_f \in F_1\}$$

Les transitions de $\Delta_{1\cdot2}$ font le lien entre les deux automates. Ce sont des copies des transitions finales de A_1 où la destination a été modifiée afin qu'elles permettent d'accéder à la place initiale de A_2.

Prouvons que l'automate $A_{1\cdot2}$ représente le langage $\mathbb{L}(A_1) \cdot \mathbb{L}(A_2)$: Si un mot $w = w_1 w_2 \dots w_n \in \mathcal{U} \cup \{\varepsilon\}$ appartient à $\mathbb{L}(A_{1\cdot2})$, alors il existe un chemin de transitions $\delta_1 \dots \delta_n \in (\Delta_1 \cup \Delta_2 \cup \Delta')^n$ tel que $(q_1, M_1 \cup M_2) \xrightarrow[\delta_1]{w_1} \dots \xrightarrow[\delta_n]{w_n} (q_f, M_f)$ avec $q_f \in F_2$ et M_F un contexte mémoire quelconque. Comme le chemin débute à la place q_1 et qu'il se termine à une place $q_f \in F_2 \subseteq Q_2$ alors il existe un unique $k \in [0, n]$ tel que $\delta_k \in \Delta'$. L'unicité de cette transition vient du fait que les transitions de Δ' sont les seules permettant de passer des places de Q_1 aux places de Q_2, de plus, il n'existe aucune transition allant d'une place de Q_2 vers une place de Q_1. On peut donc en déterminer que $\forall i < k$, $\delta_i \in \Delta_1$ et $\forall j > k$, $\delta_j \in \Delta_2$.

En considérant les k premières transitions du chemin acceptant w, il existe un contexte mémoire M' tel que $(q_1, M_1 \cup M_2) \xrightarrow[\delta_1]{w_1} \dots \xrightarrow[\delta_k]{w_k} (q_{0_2}, M')$. La transition δ_k est la copie d'une transition $\delta'_k \in \Delta_1$ ayant pour destination une place $q'_f \in F_1$, finale pour l'automate A_1. De plus les transitions de Δ_1 exploitent uniquement les couches de L_1, par définition de A_1, on peut donc en déduire que toutes les variables des couches L_2 ont les mêmes valeurs dans $M_1 \cup M_2$ que dans M', il existe donc un contexte mémoire M'_1 tel que $M' = M'_1 \cup M_2$. Ainsi, on obtient le chemin de transitions $(q_1, M_1) \xrightarrow[\delta_1]{w_1} \dots \xrightarrow[\delta'_k]{w_k} (q'_f, M'_1)$ acceptant le mot $w_1 w_2 \dots w_k$ dans A_1. Ainsi le mot $w_1 w_2 \dots w_k \in \mathbb{L}(A_1)$.

De même, les transitions $\delta_{k+1} \dots \delta_n$ font partie de Δ_2. Le segment du chemin $(q_2, M') \xrightarrow[\delta_{k+1}]{w_{k+1}} \dots \xrightarrow[\delta_n]{w_n} (q_f, M_f)$ est un chemin de transitions allant de la place initiale à une place finale de A_2 et dont les transitions utilisent uniquement les couches mémoire de L_2 par définition de Δ_2. Comme $M' = M'_1 \cup M_2$ alors on peut déduire qu'il existe un contexte mémoire M'_2 tel que $M_f = M'_1 \cup M'_2$, d'où $(q_2, M_2) \xrightarrow[\delta_{k+1}]{w_{k+1}} \dots \xrightarrow[\delta_n]{w_n} (q_f, M'_2)$ est un chemin de transitions acceptant le mot $w_{k+1} \dots w_n$ dans l'automate A_2.

Cela permet donc de prouver que tous les mots représentés par l'automate $A_{1\cdot2}$ sont la concaténation de mots des langages $\mathbb{L}(A_1) \cdot \mathbb{L}(A_2)$, ce que l'on note formellement par : $\mathbb{L}(A_{1\cdot2}) \subseteq \mathbb{L}(A_1) \cdot \mathbb{L}(A_2)$. L'autre sens de la preuve d'équivalence de langage, $\mathbb{L}(A_1) \cdot \mathbb{L}(A_2) \subseteq \mathbb{L}(A_{1\cdot2})$ est similaire à ce premier sens. Sans entrer dans les détails, elle consiste à montrer que pour tout $w_1 \in \mathbb{L}A_1$ et $w_2 \in \mathbb{L}A_2$ il existe des chemins de transitions dans leurs automates respectifs acceptant ces mots. En remplaçant la transition finale du chemin de A_1 par la transition correspondante dans Δ' on obtient un chemin de transitions acceptant $w_1 w_2$ dans l'automate $A_{1\cdot2}$. □

Propriété 4.2.4 *L'ensemble des langages représentés par les ACM est clos pour l'étoile de Kleene (^{*_r}).*

Preuve : Étant donné un ACM $A = (Q, q_0, F, \Delta, V, L, M_0)$ de langage $\mathbb{L}(A)$, nous souhaitons construire l'automate A_* représentant le langage $\mathbb{L}(A)^*$. La difficulté avec les automates à mémoire est l'évolution de la mémoire durant la reconnaissance, provoquant un effet de bord sur le contexte mémoire initial de chaque itération. Il est donc nécessaire que chaque itération soit capable de retrouver la valeur du contexte mémoire initial M_0. L'automate A_* est défini ainsi :

$$\begin{aligned} A_* \stackrel{\mathsf{def}}{=} \; & ((Q \times 2^{V\times L}) \cup \{(q_*, \emptyset), (q_\varepsilon, \emptyset)\}, (q'_0, \emptyset), (F \times 2^{V\times L}) \cup \{(q_\varepsilon, \emptyset)\}, \\ & \Delta' \cup \Delta_\varepsilon \cup \Delta_f, V \cup \{\Omega\}, L \cup L_w, M_0 \cup M_w \cup M_\Omega) \end{aligned}$$

Pour simuler le retour au contexte mémoire initial, il est nécessaire de retenir les valeurs initiales de la mémoire, car aucun mécanisme ne permet de retrouver des lettres précédemment associées aux variables après une réinitialisation ou affectation. Les opérations modifiant la mémoire seront effectuées sur des nouvelles variables initialement vides, représentées par les couches mémoire L_w, où $L_w \cap L = \emptyset$. Ainsi, pour chaque couche $l \in L$, il existe une unique couche mémoire correspondante que l'on note $l_w \in L_w$. L'idée de la construction est donc que chaque variable $X^l \in V \times L$ de A correspond à deux variables dans A_*, X^l qui contient les valeurs initiales et $X^{l_w} \in V \times L_w$ dans laquelle seront associées de nouvelles lettres durant la reconnaissance. De plus, pour vérifier la fraîcheur d'une lettre dans une couche $l \in L$ sans modifier les valeurs initiales, une nouvelle variable Ω est ajoutée à chaque couche. Les transitions vont associer une lettre à Ω^l puis réinitialiser Ω^l, ainsi la transition pourra être franchie seulement si la lettre est fraîche sans pour autant que la valeur de Ω^l ne sois modifiée.

Toutes les nouvelles variables sont initialement vides, $\forall l \in L, X \in V, M_0(X^l_w) = M_0(\Omega^l) = \emptyset$. Le contexte M_w représente les valeurs initiales des couches L_w et M_Ω représente les valeurs initiales de la variable Ω dans chaque couche.

$$\begin{aligned} M_w &= \{X^l_w \to \emptyset \mid X \in V, l_w \in L_w\} \\ M_\Omega &= \{\Omega^l \to \emptyset \mid l \in L \cup L_w\} \end{aligned}$$

Afin de pouvoir utiliser ces nouvelles variables, il est nécessaire de dupliquer les places et modifier les transitions de l'automate. Chaque place de Q est associée à l'ensemble de variables $S \in 2^{V\times L}$ qui ont été réinitialisées durant l'itération. Lorsqu'une variable est réinitialisée, ses valeurs initiales doivent être considérées comme supprimées jusqu'à la prochaine itération. Ces métadonnées sont utilisées durant la construction pour définir correctement les transitions sortantes d'une place.

L'ensemble de transitions Δ' va simuler les transitions de Δ en utilisant les couches L_w.

$$\Delta' = \left\{ \begin{array}{ll} ((q,S),\nu_w,\alpha',\overline{\nu_w},(q',S')) \mid & \exists (q,\nu,\alpha,\overline{\nu},q') \in \Delta, \alpha' \in \Gamma(\alpha,S), \\ & S' = S \cup \overline{\nu}, \\ & \nu_w = \{X^{l_w} \mid X^l \in \nu\} \cup \{\Omega^l \mid l \in L\}, \\ & \overline{\nu_w} = \{X^{l_w} \mid X^l \in \overline{\nu}\} \cup \{\Omega^l \mid l \in L\} \end{array} \right\}$$

Comme toutes les modifications sont effectuées sur les variables des couches L_w, les variables des ensembles ν et $\overline{\nu}$ sont remplacées par celles correspondantes dans L_w. Les variables Ω sont allouées et réinitialisées afin de pouvoir vérifier la fraîcheur d'une lettre dans les couches de L. La méta-donnée S' de la place d'arrivée q' est mise à jour afin de prendre en compte les variables réinitialisées dans la transition. Pour chaque transition de Δ, il existe plusieurs transitions dans Δ' avec différentes valeurs pour α, définie par la fonction Γ suivante :

$$\Gamma(\alpha,S) = \left\{ \begin{array}{ll} \alpha' \mid & \forall l \in L, \alpha(l) = \sharp \implies \alpha'(l) = \alpha'(l_w) = \sharp \\ & \wedge \alpha(l) = X \neq \sharp \implies \big(X^l \in S \backslash \nu \implies (\alpha'(l) = \sharp \wedge \alpha'(l_w) = X)\big) \\ & \qquad \vee X^l \notin S \cup \nu \implies ((\alpha'(l) = X \wedge \alpha'(l_w) = \sharp) \\ & \qquad\qquad\qquad\qquad \vee (\alpha'(l) = \sharp \wedge \alpha'(l_w) = X)) \\ & \qquad \vee X^l \in \nu \implies \alpha'(l_w) = X \wedge \alpha'(l) \in \big\{X \mid X^l \in S \cup \{\Omega^l\}\big\} \end{array} \right\}$$

Pour chaque couche l, différentes valeurs de $\alpha(l)$ sont possible en fonction de la métadonnée S. L'ensemble $\Gamma(\alpha,S)$ contient un α' pour chaque combinaison possible de valeurs de $\alpha'(l)$. Pour chaque couche $l \in L$, il y a quatre situations déterminant les valeurs de $\alpha'(l)$ et du $\alpha'(l_w)$ correspondant :

- Si $\alpha(l) = \sharp$ alors aucune variable de la couche n'est utilisée et donc $\alpha'(l) = \alpha'(l_w) = \sharp$.
- Si $\alpha(l) = X$ et $X^l \notin S \cup \nu$ alors la transition accepte une lettre associée à X^l qui peut être une des lettres qui y ont été initialement associées. Il y a donc deux valeurs possibles pour α', soit $\alpha'(l) = X$ et $\alpha'(l_w) = \sharp$ et la transition représente une des lettres initialement associées à X^l, soit $\alpha'(l) = \sharp$ et $\alpha'(l_w) = X$ et la transition représente une lettre qui a été associée à X^l dans l'itération courante.
- Si $\alpha(l) = X$ et $X^l \in S$, alors les valeurs initialement associées à X^l ont été oubliées et la transition représente seulement celles qui y ont été associées depuis le début de l'itération donc $\alpha'(l) = \sharp$ et $\alpha'(l_w) = X$.
- Si $\alpha(l) = X$ et $X^l \in \nu$, alors on essaye d'associer une lettre à la variable X^l, donc $\alpha'(l_w) = X$. Cependant, pour vérifier la fraîcheur dans la couche

l, il faut vérifier que la lettre n'est soit associée à aucune des variables de la couche l (donc $\alpha'(l) = \Omega$), soit qu'elle est associée à l'une des variables dont les valeurs initiales ont été oubliées, ce qui signifie que $\alpha'(l)$ peut être n'importe quelle variable de la couche l présente dans S.

Les transitions de Δ' permettent de simuler le comportement de l'automate A, il est ensuite nécessaire d'ajouter des transitions permettant d'itérer sur A. Ces transitions sont représentées par Δ_f :

$$\Delta_f = \{(q, \nu, \alpha, \{X^{l_w} \mid l_w \in L_w\}, (q_0, \emptyset)) \mid (q, \nu, \alpha, \overline{\nu}, q_f) \in \Delta', q_f \in F \times 2^{V \times L}\}$$

Les transitions de Δ_f sont des copies des transitions de Δ' dont la destination est une place finale. Chaque transition de Δ_f réinitialise toutes les variables des couches de L_w et a pour destination la place initiale de A, $(q_0, \emptyset)$, où la métadonnée a été réinitialisée.

Enfin, pour permettre de n'effectuer aucune itération et représenter le mot $\{\varepsilon\}$, les places $(q'_0, \emptyset)$ et $(q_\varepsilon, \emptyset)$ ont été ajoutées. La première est la nouvelle place initiale et la seconde est une nouvelle place finale.

$$\Delta_\varepsilon = \{((q'_0, \emptyset), \emptyset, \varepsilon, \emptyset, (q_0, \emptyset)), ((q'_0, \emptyset), \emptyset, \varepsilon, \emptyset, (q_\varepsilon, \emptyset))\}$$

Les deux ε-transitions de Δ_ε permettent de choisir entre reconnaître le mot ε en accédant à la place $(q_\varepsilon, \emptyset)$, où reconnaître $\mathbb{L}(A)^+$ en accédant à la place $(q_0, \emptyset)$.

Démontrons maintenant que le langage $\mathbb{L}(A^*)$ représente effectivement $\mathbb{L}(A)^*$ en montrant d'abord que $\mathbb{L}(A)^* \subseteq \mathbb{L}(A^*)$. Notons le contexte mémoire initial de A^* comme $M_0^* = M_0 \cup M_w \cup M_\Omega$ et son état initial est donc $((q'_0, \emptyset), M_0^*)$. Étant donné les mots $w_1, w_2, \ldots, w_n \in \mathbb{L}(A)$ pour tout $n \in \mathbb{N}\backslash\{0\}$, montrons que $w = w_1 w_2 w_3 \ldots w_n \in \mathbb{L}(A^*)$. Cela signifie qu'il faut trouver un chemin de transitions dans A^* acceptant w. Il est trivial de voir que le mot ε est reconnu par A^* via l'ε-transition $((q'_O, \emptyset), \emptyset, \varepsilon, \emptyset, (q_\varepsilon, \emptyset))$, menant à l'état final $((q_\varepsilon, \emptyset), M_0^*)$. Si $w \neq \varepsilon$, alors le chemin débute nécessairement par la transition $((q'_0, \emptyset), M_0^*) \xrightarrow{\emptyset, \varepsilon, \emptyset} ((q_0, \emptyset), M_0^*)$.

Considérons la relation $\sim_S$ entre les contextes mémoire de A et A^* tel que $M \sim_S M^* \Leftrightarrow \forall X^l \in S, M(X^l) = M^*(X^{l_w}) \wedge \forall Y^k \notin S, M(Y^k) = M^*(Y^k) \cup M^*(Y^{k_w})$. Comme $w_1 = e_1 e_2 \ldots e_k \in \mathbb{L}(A)$, où $\forall i, e_i \in \mathcal{U} \cup \{\varepsilon\}$, alors il existe un chemin de transitions $\delta_1, \delta_2, \ldots, \delta_k \in \Delta$ tel que $(q_0, M_0) \xrightarrow[\delta_1]{e_1} (q_1, M_1) \xrightarrow[\delta_2]{e_2} \ldots \xrightarrow[\delta_k]{e_k} (q_k, M_k)$ où $q_k \in F$. Montrons qu'il existe un chemin dans Δ' allant de l'état $((q_0, S_0), M_0^*)$ à un état final $((q_k, S_k), M_k^*)$, passant par les états intermédiaires $((q_i, S_i), M_i^*)$, tel que $\forall i \in [0, k], M_i \sim_{S_i} M_i^*$. Initialement, par définition de M_0^*, on sait que $M_0 \sim_{S_0} M_0^*$, où $S_0 = \emptyset$. Supposons que l'état $((q_i, S_i), M_i^*)$ est atteint après la lecture des i premières lettres et que $M_i \sim_{S_i} M_i^*$. Posons $\delta_i = (q_i, \nu, \alpha, \overline{\nu}, q_{i+1})$, il existe $\Delta'_{\delta_{i+1}, S_i}$, le sous-ensemble de Δ' des transitions créées à partir de δ_{i+1}

et dont l'origine est (q_i, S_i). Par définition de Δ', il existe $\delta'_{i+1} \in \Delta'_{\delta_{i+1},S_i}$ tel que $((q_i, S_i)M_i^*) \xrightarrow[\delta'_{i+1}]{e_{i+1}} ((q_{i+1}, S_{i+1}), M_{i+1}^*)$ où $S_{i+1} = S_i \cup \overline{\nu}$ et $\forall X^l \in V \cup \{\Omega\} \times L$:

- si X^l est réinitialisée dans δ_{i+1}, $X^l \in \overline{\nu}$, alors c'est la variable X^{l_w} qui est réinitialisée dans δ^*_{i+1}, $X^l \in S_{i+1}$ et $M^*_{i+1}(X^{l_w}) = \emptyset = M^*_{i+1}(X^l)$,
- si e_{i+1} est associée à X^l dans δ_{i+1}, $\alpha(l) = X$ et $X^l \in \nu \backslash \overline{\nu}$, alors elle est associée à X^{l_w} dans δ^*_{i+1}, $M^*_{i+1}(X^{l_w}) = M^*_i(X^{l_w}) \cup \{e_{i+1}\}$, cependant les transitions de δ'_{i+1} vérifient que e_{i+1} est soit fraîche dans la couche l (avec la variable Ω), soit qu'elle est associée à une des variables de l qui a été réinitialisée ($\exists Y^l \in S, e_{i+1} \in M^*_{i+1}(Y^l)$.
- sinon $M^*_{i+1}(X^{l_w}) = M^*_i(X^{l_w})$

On peut en déduire que $M_{i+1} \sim_{S_{i+1}} M^*_{i+1}$ et qu'il existe un chemin de transitions $\delta'_1 \dots \delta'_k \in \Delta'$ allant de l'état final $((q_0, \emptyset), M_0^*)$ à l'état $((q_k, S_k), M_k^*)$. Comme δ'_k mène à une place finale, alors il existe $\delta'_f \in \Delta_f$ tel que

$$((q_{k-1}, S_{k-1}), M^*_{k-1}) \xrightarrow[\delta'_f]{e_{k-1}} ((q_0, \emptyset), M_0^*).$$

On peut en conclure qu'il existe, pour tout mot $w_i \in \mathbb{L}(A)$, des chemins de transitions dans $\Delta' \cup \Delta_f$ permettant d'aller de l'état $((q_0, \emptyset), M_0^*)$ à lui même ou à un état final, et donc que $w_1 w_2 \dots w_n \in \mathbb{L}(A^*)$ et $\mathbb{L}(A)^* \subseteq \mathbb{L}(A^*)$.

Pour prouver que $\mathbb{L}(A^*) = \mathbb{L}(A)^*$ il reste à démontrer que $\mathbb{L}(A^*) \subseteq \mathbb{L}(A)^*$. Soit un mot $w \in \mathbb{L}(A^*)$, il existe un chemin de transitions de $(\Delta' \cup \Delta_f \cup \Delta_\varepsilon)^*$ acceptant le mot w. La place initiale $(q'_0, \emptyset)$ n'est pas finale et les seules transitions en sortant sont celles Δ_ε. Si $((q'_0, \emptyset), \emptyset, \varepsilon, \emptyset, (q_\varepsilon, \emptyset))$ est la première transition du chemin alors $w = \varepsilon$ car il n'existe aucune transition sortante de la place $(q_\varepsilon, \emptyset)$, et $\varepsilon \in \mathbb{L}(A)^*$ par définition de l'étoile de Kleene. Sinon, la première transition du chemin est $((q'_0, \emptyset), \emptyset, \varepsilon, \emptyset, (q_0, \emptyset))$ et les transitions suivantes forment un chemin de $(\Delta' \cup \Delta_f)^*$. Comme toutes les transitions de Δ_f ont pour destination l'état $((q_0, \emptyset), M_0^*)$ alors le chemin acceptant w est de la forme :

$$\begin{array}{c} \delta^1_1 \delta^1_2 \dots \delta^1_{|w_1|-1} \delta^1_f \delta^2_1 \delta^2_2 \dots \delta^2_{|w_2|-1} \delta^2_f \dots \delta^n_{|w_n|}, \\ \text{où } \forall i \in [1,n], \delta^i_f \in \Delta_f, \forall k \in [1, |w_i|], \delta^i_k \in \Delta' \end{array}$$

Le chemin est décomposé en n sous-chemins débutants tous par l'état $((q_0, \emptyset), M_0^*)$ et les $n-1$ premiers chemins se terminent par une transition de Δ_f tandis que le n-ième termine dans un état final. Chaque sous-chemin lit un sous-mot de w, on peut ainsi le décomposer en $w = w_1 w_2 \dots w_n$ où $\forall i \in [1,n], w_i \in (\mathcal{U} \cup \{\varepsilon\})^*$, tel que chaque mot w_i est lu par l'un des sous-chemins. Par définition de Δ_f, pour chaque $\delta^i_f \in \Delta_f$ il existe une transition $\delta^i_{|w_i|} \in \Delta'$ dont la destination est une place finale de A^* et étant déclenchable par les mêmes lettres que $\delta^i_{|w_i|}$. Ainsi, pour chaque sous-mot w_i de w il existe un chemin dans Δ' allant de la place $(q_0, \emptyset)$ à une place finale de $F \times 2^{V \times L}$.

Soit l'un des sous-mots $w_i = e^i_1 e^i_2 \dots e^i_{|w_i|}$ de w et le chemin de transitions de Δ' tel que $((q_0, \emptyset), M^*_0) \xrightarrow[\delta^i_1]{e^i_1} ((q_1, S_1), M^*_1) \xrightarrow[\delta^i_2]{e^i_2} \cdots \xrightarrow[\delta^i_{|w_i|}]{e^i_{|w_i|}} ((q^i_{|w_i|}, S_i), M^*_i)$ où $q_{|w_i|} \in F$. Montrons par récurrence qu'il existe un chemin de transitions dans A acceptant le mot w_i. Le chemin de A débute par son état initial (q_0, M_0), et comme précédemment $M_0 \sim_\emptyset M^*_0$.

Supposons que l'état $((q_k, S_k), M^*_k)$ soit atteint par le chemin de transitions $\delta^i_1 \delta^i_2 \dots \delta^i_k$ après la lecture des k premières lettres de w_i dans A^*, et l'état (q_k, M_k) soit atteint après la lecture des mêmes lettres dans A, tel que $M_k \sim_{S_k} M^*_k$. Dans A^* on sait que $((q_k, S_k), M^*_k) \xrightarrow[\delta^i_{k+1}]{e_{k+1}} ((q_{k+1}, S_{k+1}), M^*_{k+1})$, par définition de Δ' il existe une transition $\delta \in \Delta$ qui est à l'origine de la définition de δ^i_{k+1}. On peut déduire de $\delta^i_{k+1} = ((q_k, S_k), \nu^i, \alpha^i, \overline{\nu}^i, (q_{k+1}, S_{k+1}))$ que $\delta = (q_k, \nu, \alpha, \overline{\nu}, q_k + 1)$ tel que : $\nu = \{X^l \mid X^{l_w} \in \nu^i\}$ (de même pour $\overline{\nu}$) et :

$$\forall l \in L, \alpha(l) = \begin{cases} \alpha(l_w) & \text{si } \alpha(l_w) \neq \sharp \\ \alpha(l) & \text{sinon} \end{cases}$$

Comme $M_k \sim_{S_k} M^*_k$, alors la transition δ vérifie $(q_k, M_k) \xrightarrow[\delta_{k+1}]{e_{k+1}} (q_{k+1}, M_{k+1})$ car :

- si $\alpha(l) = X$ et $X^l \notin \nu$ alors $X^{l_w} \notin \nu$ et $\alpha^i(l) = X$ ou $\alpha^i(l_w) = X$ donc $e_{k+1} \in M^*_k(X^l) \cup M^*_k(X^{l_w}) = M_k(X^l)$
- si $\alpha(l) = X$ et $X^l \in \nu$ alors, par définition de Δ', $\alpha^i(l_w) = X$ et $\alpha^i(l) \in \{X \mid X^l \in \nu\} \cup \{\Omega\}$, donc $e_{k+1} \in \mathcal{U} \backslash (\bigcup_{Y \in V} M^*_k(Y^{l_w})) \cap (\mathcal{U} \backslash \bigcup_{Z^l n^o \in S} M^*_k(Z^l)) = \mathcal{U} \backslash (\bigcup_{Y \in V} M^*_k(Y^{l_w}) \bigcup_{Z^l n^o \in S} M^*_k(Z^l))$ et comme $M_k \sim_{S_k} M^*_k$ alors $\bigcup_{Y \in V} M^*_k(Y^{l_w}) \bigcup_{Z^l n^o \in S} M^*_k(Z^l) = \bigcup_{Y \in V} M_k(Y^l)$ donc e_{k+1} est fraîche pour la couche l de M_k,
- si $\alpha(l) = \sharp$ alors $\alpha^i(l) = \alpha^i(l_w) = \sharp$ donc aucune de ces couches ne rentre en compte pour savoir si la lecture de e_{k+1} permet de franchir cette transition.

De même on peut déterminer la valeur de M_{k+1} où :

- si $X^l \in \overline{\nu}$ alors $M_{k+1}(X^l) = M^*_{k+1}(X^{l_w}) = \emptyset$ et $X^l \in S_{k+1}$,
- si $X^l \in \nu \backslash \overline{\nu}$ alors $M_{k+1}(X^l) = M_k(X^l) \cup \{e_{k+1}\}$, et $M^*_{k+1}(X^l) = M^*_k(X^l) \cup \{e_{k+1}\}$,
- sinon $M_k(X^l) = M_{k+1}(X^l)$ et de même $M^*_k(X^l) = M^*_{k+1}(X^l)$

Ainsi, les contextes mémoire vérifient $M_{k+1} \sim_{S_{k+1}} M^*_{k+1}$. On peut ainsi déduire de cette récurrence qu'il existe pour chaque mot w_i un chemin de transitions dans l'automate A acceptant w_i et donc $\mathbb{L}(A^*) \subseteq \mathbb{L}(A)^*$. Nous pouvons conclure de cette preuve que $\mathbb{L}(A)^* = \mathbb{L}(A^*)$ et donc que l'ensemble des langages exprimables par les ACM est clos par l'étoile de Kleene. □

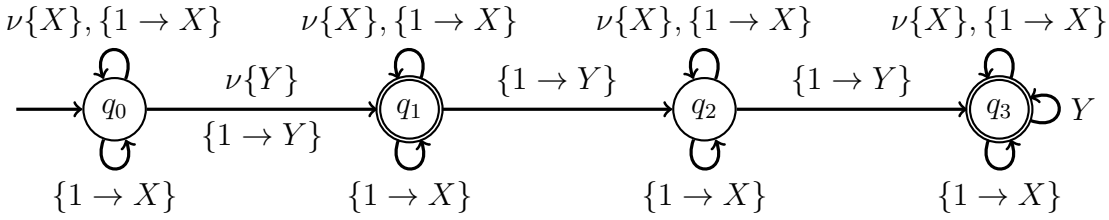


FIGURE 4.2 – ACM reconnaissant le langage $\mathcal{L}_{\neq 2}$ des mots contenant une lettre n'apparaissant pas exactement deux fois

Dans les langages rationnels, l'intersection de langages peut-être dérivé de l'opération de complément. Cependant, dans la plupart des modèles d'automates à mémoire, les ensembles de langages qui sont reconnaissables ne sont pas clos par cette opération. De même, l'ensemble des langages reconnaissables par les ACM n'est pas clos pour le complément, même si il est clos pour l'intersection (Propriété 4.2.2), qui est une opération primitive et non dérivée.

Propriété 4.2.5 (Complément) *L'ensemble des langages représentés par les ACM n'est pas clos par le complément.*

Preuve : Le complément du langage $\mathcal{L}_{\neq 2}$ reconnu par l'ACM présenté dans la Figure 4.2, est le langage $\overline{\mathcal{L}_{\neq 2}}$ des mots dont toutes les lettres apparaissent exactement deux fois. Pour encoder $\overline{\mathcal{L}_{\neq 2}}$, il est nécessaire d'être capable d'énumérer les occurrences de toutes les lettres des mots reconnus par le langage. Cependant, comme le langage est défini sur un alphabet infini, il faudrait que l'ACM puisse compter les occurrences d'une quantité arbitraire de lettres différentes. Cela est impossible pour les ACM car ils disposent d'un nombre fini de place et de variables différentes. □

La dernière propriété à laquelle nous allons nous intéresser ici est le non-déterminisme. Un automate est dit déterministe si à toutes les étapes de la reconnaissance d'un mot, il existe au plus un état atteignable. Les automates finis sont déterminisables, c'est-à-dire qu'à partir de tout automate fini non-déterministe il est possible de construire un automate fini déterministe reconnaissant le même langage. Un automate fini déterministe dispose de propriété algorithmiques très intéressantes, principalement simplifiant le problème de la reconnaissance à un problème linéaire.

Cependant, dans le cas des automates à mémoire, la présence d'un alphabet infini induit des mécaniques non-déterministes, telles que le choix d'une lettre fraîche. Ainsi, la plupart des classes d'automates à mémoire déterministes sont

moins expressives que leurs équivalents non-déterministes, tels que les FMA. Les seuls modèles déterminisables à notre connaissance sont des modèles très contraints et disposant d'une expressivité réduite [47].

Propriété 4.2.6 (Déterminisme) *Les ACM déterministes sont strictement moins expressifs que les ACM non-déterministes.*

Preuve : Nous allons prouver qu'il n'existe pas d'ACM déterministe permettant d'exprimer le langage $\mathcal{L}_{\neq 2}$, de la Figure 4.2. Les transitions $\delta_1 = q_0 \xrightarrow{\nu X, X} q_0$ et $\delta_2 = q_0 \xrightarrow{\nu Y, Y} q_1$ lisent toutes les deux une valeur fraîche et l'associent à des variables différentes. La variable Y représente la lettre qui n'apparaît pas exactement deux fois dans le mot, tandis que la variable X représente toutes les autres lettres. Le non-déterminisme entre δ_1 et δ_2 représente l'identification, de manière non-déterministe, de la lettre qui n'apparaît pas exactement deux fois.

Comme la dynamique des ACM fonctionne en une seule lecture du mot, il n'est pas possible de déterminer quelle lettre serait associée à Y avant que le mot soit entièrement lu. En considérant un alphabet fini, il serait possible d'écrire une version déterministe de ce langage en énumérant les occurrences de chaque lettre du mot. Cependant, comme l'alphabet est infini et que le mot peut contenir un nombre arbitraire de lettres différentes, il faudrait pouvoir énumérer les occurrences d'un nombre arbitraire de lettres. Ainsi il n'existe pas d'ACM déterministe avec un nombre fini de places et de variables pouvant reconnaître ce langage. □

4.2.2 Les comparaisons avec d'autres automates à mémoire

Dans cette section nous allons positionner l'expressivité des ACM par rapport aux autres modèles d'automates à mémoire. La Figure 4.3 reprend en partie celle présentée dans [57] afin de présenter le positionnement des ACM dans la littérature. En positionnant les ACM ainsi, cela nous permet de caractériser les propriétés exprimables à l'aide des ACM, mais aussi d'en voir les limitations. De plus, en montrant que l'on peut reproduire tous les langages exprimables par d'autres classes d'automates on peut déduire des propriétés algorithmiques et de décidabilité.

Sur la Figure 4.3 les flèches représentent une relation d'ordre partiel sur l'expressivité des modèles d'automates à mémoire. La flèche allant des FMA au FRA indique que l'ensemble des langages reconnaissables par un FMA est inclus dans ceux reconnaissables par des FRA. Les flèches continues sont des propriétés déjà prouvés dans la littérature, tandis que les flèches discontinues sont des propriétés que nous prouvons dans cette section. Enfin, les lignes en pointillé indiquent que l'expressivité des modèles est incomparable.

Propriété 4.2.7 *Les ACM sont strictement plus expressifs que les automates à registres frais (FRA).*

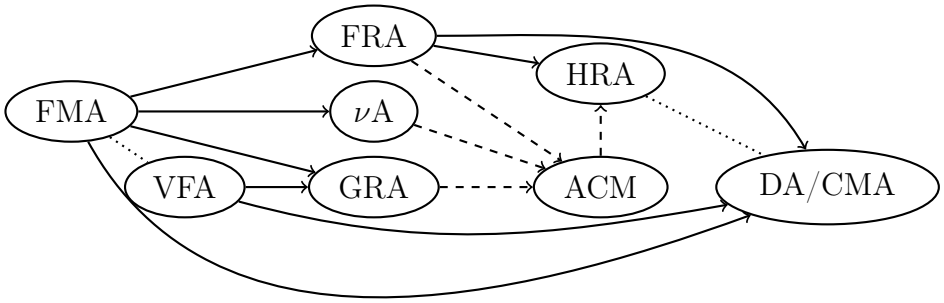


FIGURE 4.3 – Comparaison d'expressivité des modèles d'automates à mémoire. Une flèche indique une inclusion stricte de l'expressivité, celles discontinues sont issues des preuves de cette thèse. Une ligne en pointillée que l'expressivité des modèles est incomparable.

Preuve : Pour prouver que les ACM sont au moins aussi expressifs que les FRA, nous montrons comment simuler un FRA avec un ACM à deux couches. Cette preuve se base sur la définition des FRA donnée dans la Section 2.3.1.

Prouvons d'abord que les ACM sont au moins aussi expressifs que les FRA. Étant donné un FRA $A_{FR} = (Q, q_0, F, \Delta, \sigma_0, \mathcal{C}, \mathcal{U})$, on peut construire un ACM $A_{CM} = (Q \cup Q_r, q_0, F, \Delta', V, L, M_0)$ représentant le même langage. L'ACM est défini sur le même ensemble de places Q et augmenté de quelques places Q_r supplémentaires, et ayant les mêmes places initiale et finales. Le contexte mémoire de A_{CM} est défini sur deux couches $L = \{1, 2\}$, la première simule les registres et constantes, et la seconde simule l'historique afin de d'exprimer la notion de fraîcheur globale. La mémoire de l'ACM est composée d'une variable X_c^1, pour chaque constante $c \in \mathcal{C}$, initialisée comme telle : $M_0(X_c^1) = c$, d'une variable X_i^1 pour chaque registre i du FRA initialisée avec la même lettre initiale que le registre i, et enfin une variable H^2, représentant l'historique, initialisée avec l'ensemble des lettres initialement associées aux registres.

Pour chaque transition δ du FRA allant d'une place q à une place q', A_{CM} vérifie :

- si $\delta = (q, c, q')$, alors il existe une transition $(q, \emptyset, \{1 \to X_c, 2 \to \sharp\}, \emptyset, q') \in \Delta'$.
- si $\delta = (q, i, q')$, alors il existe une transition $(q, \emptyset, \{1 \to X_i, 2 \to \sharp\}, \emptyset, q') \in \Delta'$.
- si $\delta = (q, i^\bullet, q')$, alors il existe une place intermédiaire $q_i \in Q_r$ accessible via $(q, \emptyset, \varepsilon, \{X_i^1\}, q_i) \in \Delta'$ réinitialisant la variable X_i^1 afin de simuler l'écrasement de la valeur de X_i^1 dans l'ACM. Deux transitions sont nécessaires pour simuler l'affectation d'une valeur localement fraîche : $(q_i, \{X_i^1\}, \{1 \to$

$X_i, 2 \to H\}, \emptyset, q') \in \Delta'$ représentant une lettre qui a déjà été lue au moins une fois, et $(q_i, \{X_i^1, H^2\}, \{1 \to X_i, 2 \to H\}, \emptyset, q') \in \Delta'$ qui représente une lettre globalement fraîche.

— si $\delta = (q, i^*, q')$, nécessite aussi la place intermédiaire $q_i \in Q_r$ et l'ε-transition permettant de simuler l'écrasement de la valeur de X_i^1. Il existe une transition $(q_i, \{X_i^1, H^2\}, \{1 \to X_i, 2 \to H\}, \emptyset, q') \in \Delta'$.

Soit la relation $\sim_{A_{FR}}$ entre une valuation mémoire (σ, H) du FRA A_{FR} et un contexte mémoire M de l'ACM A_{CM} correspondant. On a $(\sigma, H) \sim_{A_{FR}} M$ si et seulement si $\forall c \in \mathcal{C}, M(X_c^1) = \{c\}$, $\forall i, M(X_i^1) = \{\sigma(i)\}$ et $M(H^2) = H \cup \{\sigma_0(i) \mid 1 \leq i \leq n\}$. Pour prouver que les deux automates représentent le même langage, prouvons par récurrence que l'état (q, σ, H) de A_{CM} est atteignable par la lecture d'un mot w si et seulement si l'état (q, M) est atteignable dans l'ACM où $(\sigma, H) \sim_A M$ où $q \in Q$. Initialement, le mot ε ne permet pas de franchir de transition dans les FRA, donc seul l'état initial $(q_0, \sigma_0, \emptyset)$ est atteignable dans A_{FR}. De même, par définition $(\sigma_0, \emptyset) \sim_{A_{FR}} M_0$, et (q_0, M_0) est atteignable par la lecture du mot ε. Les seules transitions franchissables sont des ε-transitions, et par définition de A_{CM}, leurs destinations sont uniquement des places de Q_r et ne font pas partie de Q.

Supposons qu'un état (q, σ, H) soit atteignable dans A_{FR} après la lecture d'un préfixe w, et que le même préfixe mène à l'état (q, M) dans A_{CM} où $(\sigma, H) \sim_{A_{FR}} M$. Soit la lecture d'une lettre e, observons quels états sont atteignables :

— Si $e \in \mathcal{C}$, un état est atteignable dans A_{FR} seulement s'il existe une transition (q, c, q') où $e = c$, et l'état atteint serait alors (q', σ, H). Si une telle transition existe, alors par définition de Δ' il existe une transition $(q, \emptyset, \{1 \to X_c, 2 \to \sharp\}, \emptyset, q')$ dans A_{CM} permettant d'aller de l'état (q, M) à l'état (q', M).

— Si $e \in \mathcal{U}$ et qu'il est déjà stocké dans le registre i, alors seule une transition (q, i, q') peut accepter cette lettre et elle mènerait alors à l'état (q', σ, H). Si une telle transition existe dans A_{FR} alors par définition il existe une transition $(q, \emptyset, \{1 \to X_i, 2 \to \sharp\}, \emptyset, q') \in \Delta'$ allant de l'état (q, M) à l'état (q', M), car $M(X_i^1) = e$ car $(\sigma, H) \sim_{A_{FR}} M$.

— Si $e \in \mathcal{U} \cap H$ mais qu'il n'existe aucun registre de σ stockant e, alors seule une transition de la forme $(q, i^\bullet, q')$ peut accepter la lettre et elle mène à l'état $(q', \sigma_{[i \to e]}, H)$. Par définition de Δ', il existe une ε transition allant de q à $q_i \in Q_r$ réinitialisant la variable X_i^1 permettant d'aller de l'état (q, M) à $(q_r, M[X_i^1 \to \emptyset])$. De la place q_i il existe une transition $(q, \{X_i^1\}, \{1 \to X_i, 2 \to H\}, \emptyset, q')$ acceptant la lettre e car elle est associée à H_2 mais à aucune variable de la couche 1. Cette transition permet donc à A_{CM} d'accéder à l'état $(q', M[X_i^1 \to \{e\}])$, et $(\sigma[i \to e], H) \sim_{A_{FR}} M[X_i^1 \to \{e\}]$.

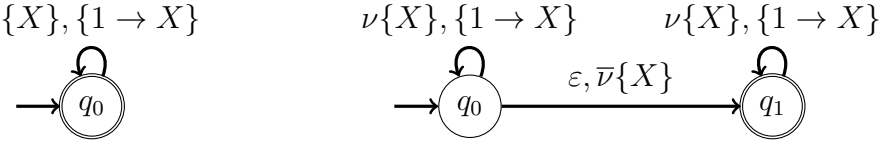


FIGURE 4.4 – À gauche, ACM représentant le langage $\mathcal{L}_{\neq 1}$, des mots dont toutes les lettres sont différentes (gauche). À droite, l'ACM représentant $\mathcal{L}_{\neq 1} \cdot \mathcal{L}_{\neq 1}$.

— Si $e \in \mathcal{U} \backslash H$, la lettre pourra être acceptée par une transition de la forme $(q, i^\bullet, q')$ ou par une transition (q, i^*, q'), dans les deux cas menant à l'état $(q', \sigma[i \to e], H \cup \{e\}$. La présence d'une des ces deux transitions induit, par définition de Δ', l'existence d'une ε-transition menant à la place q_i de laquelle il existera une transition $(q_i, \{X_i^1, H^2\}, \{1 \to X_i, 2 \to H\}, \emptyset, q')$ acceptant une lettre globalement fraîche. Via cette transition, il sera possible dans A_{CM} d'accéder depuis (q, M) successivement aux états $(q_i, M[X_i^1 \to \emptyset])$ puis $(q_i, M[X_i^1 \to \{e\}], H^2 \to H^2 \cup \{e\}])$ où $\sigma[i \to e], H \cup \{e\} \sim_{A_{FR}} M[X_i^1 \to \{e\}], H^2 \to H^2 \cup \{e\}]$.

On peut ainsi voir que l'ACM A_{CM} simule A_{FR} et représente donc le même langage.

Les langages exprimables par les ACM sont clos par la concaténation et l'étoile de Kleene, ce qui n'est pas le cas pour les FRA. On peut donc en conclure que les ACM sont strictement plus expressifs que les FRA. Par exemple, les ACM et FRA peuvent exprimer le langage $\mathcal{L}_{\neq 1}$ des mots dont toutes les lettres sont différentes les unes des autres. Cependant, les ACM peuvent exprimer le langage $\mathcal{L}_{\neq 1} \cdot \mathcal{L}_{\neq 1}$, avec l'automate de la Figure 4.4, tandis qu'il n'existe pas de FRA pouvant le représenter.

□

Propriété 4.2.8 *Les automates à historiques (HRA) sont strictement plus expressifs que les ACM.*

Preuve : Pour montrer que les HRA sont au moins aussi expressifs que les ACM, nous allons présenter une méthode permettant de construire à partir d'un ACM quelconque, un HRA exprimant le même langage. Pour rappel, la définition des HRA est donnée dans la Section 2.3.1. Étant donné l'ACM $A_{CM} = (Q, q_0, F, \Delta, V, L, M_0)$ défini sur l'alphabet $\mathcal{U}$, construisons le HRA $A_{HR} = (Q \cup Q_\delta, q_0, F, \Delta', H = V \times L, \sigma_0)$. L'ensemble de places A_{HR} contient toutes les places de A_{CM} en plus des places $Q_\delta = Q \times \Delta$ utilisées pour la définition de Δ'. Il est défini avec la même place initiale q_0 et les mêmes places finales F. L'automate A_{HR} contient un historique pour chaque variable $X^l \in V \times L$ de A_{CM}, on va ainsi s'autoriser à nommer les historiques de la même manière que les variables correspondantes : $X^l \in H$. Les transitions de Δ' et les places Q_r servent à simuler les

transitions de Δ dans A_{HR}, elles sont définies de la manière suivante :

$$\begin{aligned}\Delta' = \ & \{(q, H_r, H_w, q_\delta) \mid \ (q, \nu, \alpha, \overline{\nu}, q') \in \Delta, \alpha \neq \varepsilon, H_r \in \Gamma(\nu, \alpha), \\ & \qquad\qquad H_w = H_r \cup (\nu \cap \{X^l \mid \alpha(l) = X\})\} \\ & \cup\{(q_\delta, \overline{\nu}, q') \mid (q, \nu, \alpha, \overline{\nu}, q') \in \Delta, \alpha \neq \varepsilon\} \\ & \cup\{(q, \overline{\nu}, q') \mid (q, \nu, \varepsilon, \overline{\nu}, q') \in \Delta\}\end{aligned}$$

où

$$\Gamma(\nu, \alpha) = \left\{ \{X^l \mid \alpha'(l) = X \neq \sharp\} \,\middle|\, \alpha(l) = X \neq \sharp \implies \begin{array}{l}(\alpha'(l) = X \wedge X^l \notin \nu) \\ \vee(\alpha'(l) = \sharp \wedge X^l \in \nu)\end{array} \right\}$$

Pour chaque non-ε transition $\delta \in \Delta$ de A_{CM} allant de q à q', il existe une place intermédiaire $q_\delta \in Q_r$ dans A_{HR}. La place q_δ sert à décomposer δ en deux transitions successives dans A_{HR}, une première qui représente la lettre lue, puis une ε-transition qui réinitialise les lettres de $\overline{\nu}$. Cependant, pour représenter le même ensemble de lettres que δ, plusieurs transitions allant de la place q à q_δ seront nécessaires. Pour représenter cela, $\Gamma(\nu, \alpha)$ indique toutes les valeurs possible de H_r. Si $\alpha(l) = X \neq \sharp$ et que $X^l \notin \nu$, alors la lettre lue pour franchir la transition doit être associée à X^l, et comme les couches sont injectives, on sait qu'aucune autre variable de la couche l n'y est associée, et donc X^l est la seule variable de la couche l dans H_r. Si $\alpha(l) = X \neq \sharp$ et que $X^l \in \nu$, alors la transition reconnaît une lettre fraîche pour la couche l, donc aucune variable de la couche l ne doit être présente dans H_r. Si $\alpha(l) = \sharp$, la lettre lue peut être associée ou non à une variable de la couche l, or dans les HRA il est nécessaire d'indiquer l'ensemble exact d'historiques auxquelles la lettre est associée. Il faudra donc une transition dans A_{HR} allant de q à q_δ pour chaque combinaison possible d'historiques dans les couches où $\alpha(l) = \sharp$. Enfin, les ε-transitions de A_{CM} sont directement traduites en ε-transitions dans A_{HR} sans utiliser de place intermédiaire.

Soit la relation $\sim_{V \times L}$ entre une valuation mémoire σ d'un HRA et une valuation mémoire M d'un ACM telle que :

$$\sigma \sim_{V \times L} M \Leftrightarrow \forall X^l \in V \times L, \sigma(X^l) = M(X^l).$$

Cette relation permet, entre autre, d'indiquer que les historiques représentant des variables d'une même couche ne sont pas associées aux mêmes valeurs, transposant donc l'injectivité des couches des ACM aux HRA. Pour prouver que l'automate A_{HR} représente le même langage que A_{CM}, nous procédons par induction sur la structure de l'automate. Cela revient à prouver qu'il existe une transition $\delta \in \Delta$ permettant d'aller de l'état (q, M) à (q', M') lors de la lecture de $e \in \mathcal{U} \cup \{\varepsilon\}$ dans A_{CM} si et seulement s'il existe une transition $\delta' \in \Delta'$ permettant d'aller de l'état (q, σ) à (q', σ') lors de la lecture de e dans A_{HR} où $\sigma \sim_{V \times L} M$ et $\sigma' \sim_{V \times L} M'$.

Si $e = \varepsilon$ alors δ est une ε-transition et $M' = M[\overline{\nu} \to \emptyset]$, et par définition de A_{HR}, il existe une ε-transition $(q, \overline{\nu}, q')$ réinitialisant les historiques de $\overline{\nu}$, permettant d'aller de l'état (q, σ) à (q, σ') où $\sigma' = \sigma[\overline{\nu} \to \emptyset]$ et donc $\sigma' \sim_{V \times L} M'$.

Si $e \neq \varepsilon$, alors $\delta = (q, \nu, \alpha, \overline{\nu}, q')$ où $\alpha \neq \varepsilon$, alors il existe dans A_{HR} l'ensemble de transitions $\Delta'_\delta \subset \Delta'$ menant à la place q_δ qui ont été construites à partir de δ. La seule différence entre les transitions de Δ_δ est la valeur de α, pour chaque couche $l \in L$:

- si $\alpha(l) = X \in V$ et $X^l \notin \nu$, toutes les transitions $(q, H_r, H_w, q') \in \Delta'_\delta$ vérifient que $X^l \in H_r \cap H_w$ et aucun autre historique de la couche l ne fait partie de ces ensembles.
- si $\alpha(l) = X \in V$ et $X^l \in \nu$, toutes les transitions $(q, H_r, H_w, q') \in \Delta'_\delta$ vérifient qu'aucune variable de la couche l n'est présente dans H_r, car e doit être fraîche pour la couche l, mais $X^l \in H_w$ afin d'associer e à X^l.
- si $\alpha(l) = \sharp$, la transition δ ne *consulte* aucune variable de la couche, alors que la lettre pourrait y être présente. Ainsi, chaque transition $\delta' \in \Delta'_\delta$ est annotée avec une combinaison d'historiques de ces couches afin de *rechercher* les historiques qui pourraient contenir la lettre lue.

Si δ permet d'aller de l'état (q, M) à (q', M') lors de la lecture d'une lettre e, il existe une transition $\delta' \in \Delta'_\delta$ acceptant la lettre et permettant d'aller à l'état $(q_\delta, \sigma[H_w \to H_w \cup \{e\}])$ et il existe une unique ε-transition quittant cet état menant à l'état (q', σ') où $\sigma' = \sigma[H_w \to H_w \cup \{e\}][\overline{\nu} \to \emptyset]$, et donc $\sigma' \sim_{V \times L} M$.

Par définition de A_{HR}, on sait que $\sigma_0 \sim_{V \times L} M_0$. Donc, il existe un chemin de transitions acceptant le mot w dans A_{CM} si et seulement s'il existe un chemin de transitions acceptant w dans A_{HR}. On peut donc en conclure que pour tout ACM, il est possible de construire un HRA acceptant les mêmes mots, représentant donc le même langage.

Les HRA sont strictement plus expressifs que les ACM car il existe des langages que seuls les HRA peuvent exprimer. Par exemple, le langage $\mathcal{L}_{ex}$, représenté par le HRA de la Figure 4.5, n'est pas représentable par un ACM. Le langage $\mathcal{L}_{ex}$ est l'ensemble des mots $w = uv$ dont le préfixe $u \in \mathcal{L}_{\neq 1}$ est un mot de longueur *paire* et dont toutes les lettres sont différentes, et le suffixe $v = v_1 v_2 v_3 \dots v_n$ est un mot dont toutes les lettres v_i respectent la propriété : si i est impair alors la précédente occurrence de la lettre v_i était à une position paire de w, et si i est pair alors la précédente occurrence de v_i était à une position impaire de w. Dans les HRA, $\mathcal{L}_{ex}$ est exprimé en utilisant la fonctionnalité de *transfert*, permettant de déplacer la lettre lue entre des historiques, ce qui permet entre autre de supprimer la lettre lue de certains historiques sans totalement les réinitialiser. Cette fonctionnalité n'existe pas dans les ACM, et il est donc impossible d'exprimer $\mathcal{L}_{ex}$ avec un ACM utilisant un nombre fini de places et de variables. □

Dans [46], il est prouvé que l'expressivité des HRA est incomparable à celle des

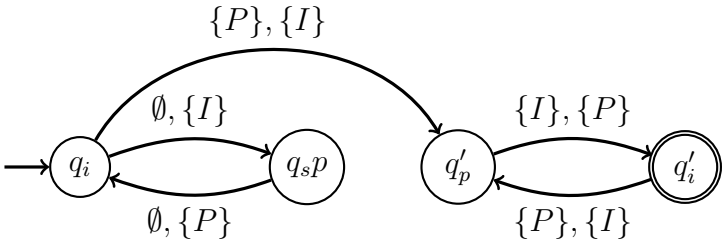


FIGURE 4.5 – Automates à Historique représentant le langage $\mathcal{L}_{ex}$

Class Memory Automata (CMA) [20], et donc aussi des *Data Automata* (DA) [23], à cause de l'opération de réinitialisation d'historique. En effet, il y est démontré que la sous-classe de HRA n'utilisant aucune réinitialisation est strictement moins expressive que les CMA. Comme les ACM sont moins expressifs que les HRA mais disposent aussi de cette possibilité de réinitialiser des variables, on peut conjecturer que l'expressivité des ACM est incomparable à celle des CMA. Mais on peut aussi conjecturer que la sous classe des ACM n'utilisant pas de réinitialisation est strictement moins expressive que les CMA.

Le modèle des ν-automates [31] est l'inspiration principale des ACM. Les ν-automates sont un intermédiaire aux HRA et au FMA car chaque variable/historique peut être associé à plusieurs lettres simultanément, mais elles sont soumises à la contrainte d'injectivité interdisant une lettre à être associée simultanément à deux variables. Les ACM sont une généralisation de ce modèle, avec l'introduction des couches permettant de relâcher partiellement la contrainte d'injectivité.

Propriété 4.2.9 *Les ACM sont strictement plus expressifs que les ν-automates.*

Preuve : Les ACM peuvent être vus comme une généralisation des ν-automates, où ces derniers sont des ACM à une couche. Ainsi, l'ACM représentant le même langage qu'un ν-automate est le ν-automate lui-même. De plus, l'ensemble des langages reconnaissables par les ν-automates ne sont pas clos par l'intersection, ce qui est le cas dans les ACM car ils peuvent empiler les couches. Cela nous permet donc de conclure que les ACM sont bien strictement plus expressifs que les ν-automates. □

Propriété 4.2.10 *Les ACM sont strictement plus expressifs que les automates à prédictions (GRA).*

Preuve : La définition des GRA est indiquée dans la Section 2.3.1. La mécanique d'affectation non-déterministe n'existe pas dans les ACM. Elle permet d'affecter une lettre choisie de manière non-déterministe à un registre lors d'une ε-transition.

La particularité de cette mécanique vient de la propriété d'injectivité des registres. Au moment de l'affectation d'une lettre à un registre, elle est différente de toutes celles déjà associées aux autres registres. De plus, la lettre est différente de toutes les lettres qui seront associées aux autres registres avant que sa valeur ne soit déterminée.

Pour montrer que les ACM sont au moins aussi expressifs que les GRA, nous montrons qu'il est possible de construire un ACM représentant le même langage. Pour simuler l'affectation non-déterministe dans les ACM, nous proposons une construction qui va contenir un historique de toutes les valeurs qui sont ou seront stockées entre une affectation d'un registre et sa prochaine utilisation. Ainsi, lors du franchissement de la transition déterminant la valeur de la lettre, il sera possible d'utiliser la propriété d'injectivité des couches pour contraindre la lettre à être différente de celles de l'historique. Bien qu'en théorie la création de tels historiques devrait être simple via l'utilisation des variables des ACM, la construction est en réalité compliquée car il est difficile d'initialiser correctement ces historiques.

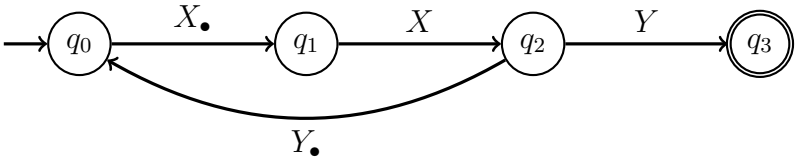


FIGURE 4.6 – Un GRA défini sur les registres X et Y.

Présentons un exemple afin d'illustrer l'intuition de la construction d'un ACM équivalent à un GRA donné. La Figure 4.6 représente un GRA relativement simple qui reconnaît le langage de tous les mots dont la dernière lettre est différente des deux précédentes. L'automate utilise deux registres, X qui représente toutes les lettres du mot sauf la dernière, et Y qui représente la dernière lettre du mot. La transition allant de q_0 à q_1 est une ε-transition qui affecte une lettre à X de manière non-déterministe. Puis la transition allant q_1 à q_2 est franchissable lors de la lecture de la lettre associée au registre X. Deux transitions sortent de la place q_2 : la transition allant vers la place finale q_3 franchissable lors de la lecture de la dernière lettre du mot, et une ε-transition revenant à q_0 affectant une lettre au registre Y. Ainsi, lors d'une reconnaissance, la première fois que la place q_2 est atteinte, il n'est pas possible de franchir la transition allant à q_3 car aucune lettre n'est affectée à Y.

La Figure 4.7 représente un ACM simulant le GRA de la Figure 4.6 permettant d'illustrer les mécaniques mise en place pour simuler l'affectation non-déterministe avec des ACM. Cet automate est en réalité une version réduite de l'automate qui serait réellement construit. Comme, dans la Figure 4.6, les affectations de X sont directement suivies de la transition déterminant sa valeur, cela consiste simplement

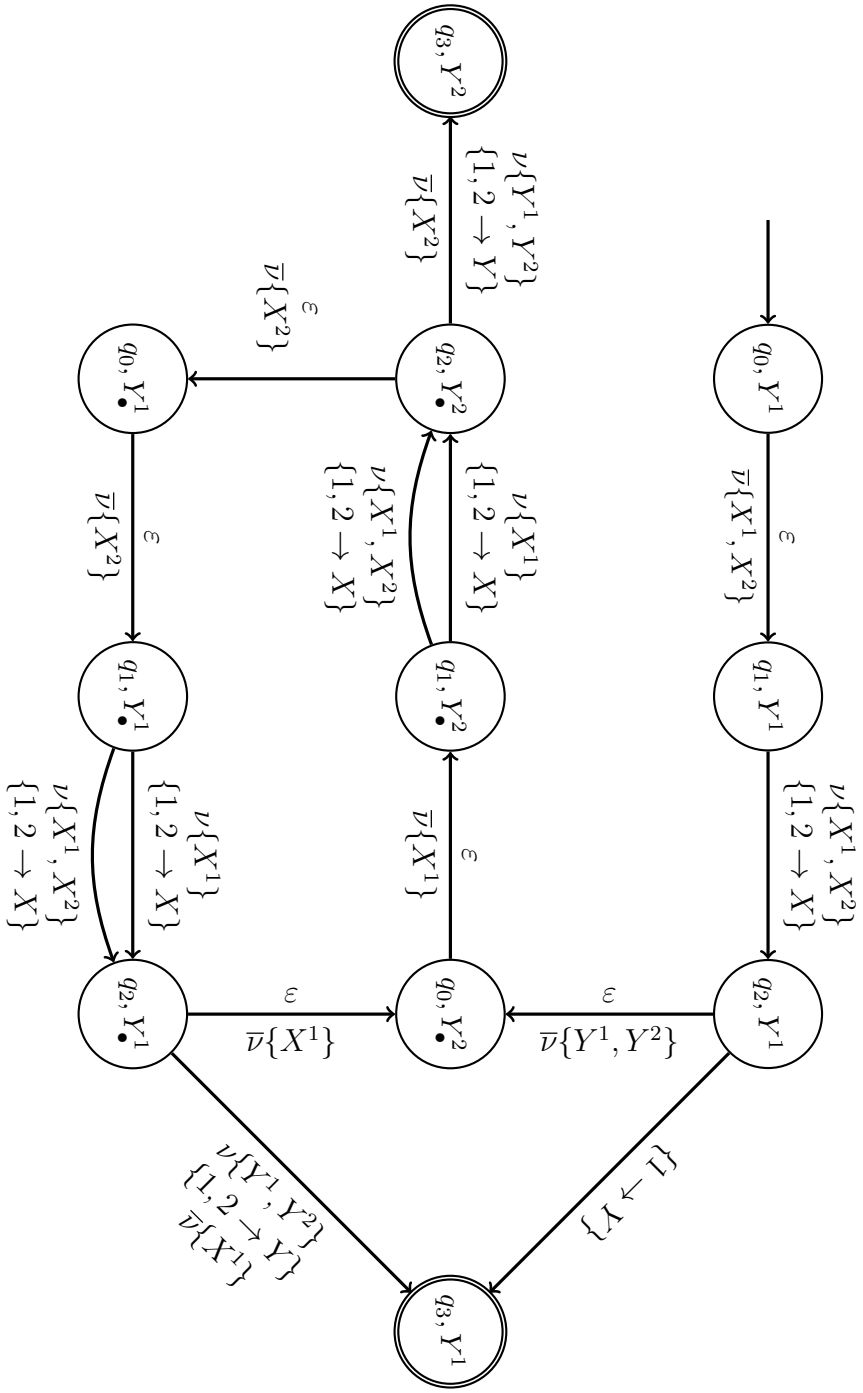


FIGURE 4.7 – L'ACM généré pour reconnaître le langage du GRA de la Figure 4.6.

à y affecter une valeur fraîche. Ainsi, la construction détaille seulement l'historique utilisé pour simuler les affectations de Y.

L'ACM est défini avec 4 variables X^1, X^2, Y^1, Y^2 réparties sur deux couches $L = \{1, 2\}$. Les deux couches sont utilisées alternativement pour représenter l'historique de Y. On peut voir dans les places de l'ACM, en plus de leur identifiant $(q_0, q_1, \dots)$, des symboles $(Y^1_\bullet)$ qui sont uniquement utilisés lors de la construction. Si le symbole est Y^1 alors il indique que c'est la couche 1 qui est actuellement utilisée pour représenter l'historique de Y. Le symbole $\bullet$ en indice indique si le registre a été assigné précédemment et que l'on cherche à déterminer la valeur qui y est associée. Ainsi, pour chaque place dans le GRA, il existe plusieurs places dans l'ACM généré, une pour chaque valeur possible de l'annotation. Dans la Figure 4.7, seules les places atteignables sont représentées.

L'ε-transition $q_2 \xrightarrow{Y_\bullet} q_0$ représentant l'affectation dans le GRA, est traduite dans l'ACM par des ε-transitions réinitialisant les variables Y dans toutes les couches. À partir de l'état suivant cette transition, il est nécessaire de mémoriser les lettres qui sont et seront associées aux autres registres, jusqu'à ce qu'on franchisse une transition permettant de déterminer la valeur de Y. Ainsi, dans l'ACM, trois transitions résultent de la traduction de la transition de $q_2 \xrightarrow{Y_\bullet} q_0$:

— La transition allant de (q_2, Y^1) à $(q_0, Y^2_\bullet)$, où on commence la recherche de la lettre assignée à Y. Comme la place d'origine est annotée par Y^1, alors la couche 2 est utilisée comme initialisation et elle devient l'historique dans l'état d'arrivée.

— La transition allant de $(q_2, Y^2_\bullet)$ à $(q_0, Y^1_\bullet)$ ainsi que celle allant de $(q_2, Y^1_\bullet)$ à $(q_0, Y^2_\bullet)$. Ces deux transitions sont le cas particulier d'un registre assigné de nouveau alors que sa lettre n'a pas encore été déterminée. Dans ce cas, l'historique est mis à jour afin de ne contenir que les lettres actuellement associées aux autres registres. Les autres lettres qui ont été mémorisées sont donc oubliées. Pour ce faire, la couche servant de valeur initiale à l'historique devient l'historique et les variables de l'ancienne couche d'historique sont réinitialiées.

L'ε-transition $q_0 \xrightarrow{X_\bullet}$ du GRA assigne une nouvelle lettre au registre X. Dans l'ACM équivalent, il n'est pas nécessaire d'utiliser un historique pour simuler l'affectation de X car sa valeur est déterminée par la transition suivante, nous expliquons seulement comment cette affectation ce traduit par rapport à l'historique de Y. Dans l'ACM, trois transitions sont issus de la traduction de cette transition :

— l'ε-transition $(q_0, Y^1) \xrightarrow[\overline{\nu}\{X^1, X^2\}]{\varepsilon} (q_1, Y^1)$ est dans le cas où aucun historique ne mémorise les lettres assignées aux registres. Ainsi, la variable X est réinitialisée dans toutes les couches.

— les deux ε-transitions $(q_0, Y^2_\bullet) \xrightarrow[\overline{\nu}\{X^1\}]{\varepsilon} (q_1, Y^2_\bullet)$ et $(q_0, Y^1_\bullet) \xrightarrow[\overline{\nu}\{X^2\}]{\varepsilon} (q_1, Y^1_\bullet)$ qui

représentent les situations où l'on cherche à déterminer la lettre assigner à Y et où les lettres assignées aux autres valeurs sont mémorisées dans la couche servant d'historique. Ainsi la variable X est réinitialisée dans toutes les couches sauf celle utilisée comme historique.

La transition $q_1 \xrightarrow{X} q_2$ détermine la lettre assignée à X. Dans l'ACM, 5 transitions sont issues de sa traduction :

— La transition $(q_1, Y^1) \xrightarrow[\{1,2\to X\}]{\nu\{X^1,X^2\}} (q_2, Y^1)$ associe la lettre aux variables X dans toutes les couches. Aucun traitement particulier n'est mis en place car l'historique ne mémorise pas les lettres dans ces places.

— les deux transitions allant de $(q_1, Y^2_\bullet)$ à $(q_2, Y^2_\bullet)$. Ici, la couche 2 est l'historique mémorisant les lettres assignées aux registres. Deux transitions sont nécessaires car lors de l'affectation d'une lettre, il est possible de réassigner une lettre qui avait déjà été précédemment associée à un registre et la lettre serait donc déjà stockée dans l'historique. Ainsi, la transition annotée par $\nu\{X^1\}, \{1, 2 \to X\}$ représente la situation où la lettre est déjà stockée dans l'historique, et celle annotée par $\nu\{X^1, X^2\}, \{1, 2 \to X\}$ correspond à la situation où la lettre assignée est différente des précédentes.

— les deux transitions allant de $(q_1, Y^1_\bullet)$ à $(q_2, Y^1_\bullet)$ représentent le même scénario que les deux précédentes mais où l'historique est la couche 1.

Enfin, la transition finale $q_2 \xrightarrow{Y} q_3$ détermine la lettre qui est assignée Y. Dans l'ACM, à nouveau trois transitions sont issues de sa traduction :

— $(q_2, Y^1) \xrightarrow{\{1\to Y\}} (q_3, Y^1)$, cette transition représente la situation où Y n'a pas été assigné précédemment, ainsi elle est franchissable seulement lors de la lecture d'une lettre déjà associée à Y. Comme la place indique que l'historique précédent de Y est la couche 1 alors c'est la variable Y^1 qui annote la transition.

— Les deux transitions, allant respectivement de $(q_2, Y^1_\bullet)$à (q_3, Y^1) et de $(q_2, Y^2_\bullet)$ à (q_3, Y^2), elles suivent une affectation et détermine la valeur courante de la lettre. Lors du franchissement de ces transitions, la lettre lue est associée aux variables Y de toutes les couches, et par la propriété d'injectivité des couches, la lettre doit être différente de toutes celles mémorisées dans l'historique. La variable ayant accumulée toutes les lettres est ensuite réinitialisée.

C'est par une telle construction que l'on peut simuler un GRA avec les ACM. Ainsi les ACM peuvent reconnaître tous les langages exprimables par des GRA.

Cependant, le langage $\mathcal{L}_{\neq 1}$, représenté par l'ACM de la Figure 4.4, est l'ensemble des mots dont toutes les lettres sont différentes. Les GRA ne peuvent pas exprimer le langage $\mathcal{L}_{\neq 1}$, car une fois que tous les registres du GRA contiennent une lettre, la reconnaissance d'une lettre fraîche nécessitera l'écrasement d'une des

lettres précédemment stockées dans un registre. Une fois une lettre écrasée, le GRA ne sera pas capable de la différencier d'une autre lettre fraîche, comme pour les FRA. On peut donc en conclure que les ACM sont strictement plus expressifs que les GRA. □

4.2.3 Les problèmes de décision et les complexités

Étant donné deux ACM A_1 et A_2, le problème de l'inclusion de langages consiste à déterminer si le langage représenté par un automate A_1 est inclus dans celui de l'automate A_2 : $\mathcal{L}(A_1) \subseteq \mathcal{L}(A_2)$. Il est connu que le problème de l'inclusion de langages est indécidable pour les Automates à registres [69]. Comme, les ACM sont capables de simuler les automates à registres (Propriété 4.2.7), alors le problème est aussi indécidable pour les ACM.

Propriété 4.2.11 *Le problème de l'inclusion de langage exprimé par des ACM est indécidable.*

Le problème de l'universalité consiste à déterminer si un automate représente le langage de tous les mots existant sur un alphabet donné. Plus formellement, étant donné un ACM A défini sur l'alphabet infini $\mathcal{U}$, le problème consiste à déterminer si $\mathcal{L}(A) = \mathcal{U}^*$. Le problème de l'universalité est indécidable pour les automates à registres [69], et donc pour les ACM.

Propriété 4.2.12 *Le problème de l'universalité est indécidable pour les ACM.*

Le problème du langage vide (*emptiness checking*) consiste à détecter si le langage exprimé par un automate est le langage vide $\emptyset$. Même si c'est un des problèmes les plus largement étudiés dans la théorie des automates, nous l'avons peu étudié dans le cadre de cette thèse. Aussi, nous n'avons pas déterminé la complexité de la résolution de ce problème pour les ACM. Cependant, comme tous les FMA et FRA peuvent être simulés par un ACM, qui eux mêmes peuvent être simulés par des HRA, il est possible d'estimer une borne inférieure et une supérieure à la complexité du problème. Le problème est décidable pour les HRA et est dans la classe de complexité Ackermann-complet, ce qui est une borne supérieure de la complexité pour le problème dans les ACM. De même, le problème est NP-complet pour les FMA, ce qui donne une borne de complexité inférieure.

Propriété 4.2.13 *Le problème du langage vide est décidable et NP-difficile pour les ACM.*

Le problème de l'appartenance au langage, *Membership Problem*, consiste à déterminer si un mot w donné appartient au langage reconnu par un automate A

donné. Dans le cadre des automates, ce problème est aussi appelé le problème de la reconnaissance et il consiste à déterminer un *run* acceptant dans A, c'est-à-dire, un chemin de transitions dont le franchissement lors de la lecture de w va de l'état initial à un état final. C'est le principal problème qui nous intéresse dans cette thèse. Le Chapitre 5 présente un algorithme résolvant ce problème.

C'est un problème NP-complet pour les FMA, [75], il appartient donc à la même classe complexité que le problème du langage vide. Le *membership problem* n'a pas été étudié à notre connaissance pour les HRA et les FRA. Cependant, comme l'*emptiness checking* est Ackermann-complet pour les HRA, on peut en déduire que la complexité du *membership problem* est aussi Ackermann pour les ACM. En effet, il est trivial à partir d'un mot w de construire un HRA reconnaissant exactement ce mot, en utilisant l'affectation initiale des historiques, puis comme les HRA sont clos pour l'intersection de langages [46], de faire l'intersection de l'automate ainsi construit avec celui dont on souhaite tester l'appartenance et ensuite de résoudre le *emptiness checking*. Comme les ACM sont simulables par les HRA, alors on peut en déduire que :

Propriété 4.2.14 *Le problème de l'appartenance au langage,* membership problem*, est décidable pour les ACM et est NP-difficile.*

Nous disposons ainsi des bornes supérieure et inférieure pour la complexité du *Membership Problem* pour les ACM. Dans la Section 5.1 nous proposons un algorithme permettant la reconnaissance d'un mot dans les ATCM, la version temporisée des ACM.

4.3 L'équivalence avec les ECM

Les ACM sont développés comme un formalisme de reconnaissance pour les expressions temporisées à couches mémoire (ETCM). Dans cette section nous souhaitons présenter l'équivalence entre les ETCM et les automates temporisés à couches mémoire (ATCM), la version temporisée des ACM. Dans un premier temps nous allons prouver l'équivalence entre les expressions à couches mémoire (ECM), la version non-temporisée des ETCM, et les ACM. Puis nous étendrons cette preuve aux versions temporisées du langage de spécification et du modèle d'automate.

La syntaxe des ECM est un sous-ensemble de la syntaxe des ETCM de laquelle ont été retirés tous les opérateurs spécifiques à l'expression de contraintes de temps. On peut ainsi résumer la syntaxe des ECM à :

$$\begin{array}{lll} \textit{Terminal } t & ::= & \emptyset \mid a \mid X^l \mid \sharp X^l \\ \textit{Expression } e, e_1 e_2 & ::= & t \mid e_1 \cdot e_2 \mid e_1 + e_2 \mid e_1 \& e_2 \mid e^*/\bar{l}/ \mid e\overline{X^l}! \mid e \uparrow_n \mid e_{\backslash a} \end{array}$$

Théorème 4.3.1 *Pour tout langage $\mathcal{L}$, il existe une ECM e et un contexte mémoire M tel que $\mathbb{L}_M(e) = \mathcal{L}$ si et seulement si il existe un ACM tel que $\mathbb{L}(A) = \mathcal{L}$*

Les deux sections suivantes sont les preuves du Théorème 4.3.1. La Section 4.3.1 présente une preuve que pour toute ECM, il est possible de construire un ACM reconnaissant le même langage. Ensuite, la Section 4.3.2 présente un algorithme permettant de créer à partir d'un ACM une ECM représentant le même langage.

4.3.1 ECM vers ACM

Dans cette première section nous allons démontrer que les langages exprimés par les ECM sont inclus dans ceux exprimés par les ACM. Pour cela nous allons présenter une méthode permettant de construire un ACM équivalent à une ECM au contexte mémoire initial près. Notons $\mathbb{L}_M(A)$ le langage reconnu par l'automate A en remplaçant son contexte mémoire initial par M. Étant donné une ECM e, on peut construire, indépendamment de son contexte mémoire initial, un ACM A, tel que pour tout contexte mémoire M_e on est capable de définir un contexte mémoire M_A où $\mathbb{L}_{M_e}(e) = \mathbb{L}_{M_A}(A)$. En effet, les contextes M_e et M_A sont très rarement identiques car, lors de la construction de l'automate, il peut être nécessaire d'introduire de nouvelles variables, autres que celles sur lesquelles est définie e.

La méthode de construction présentée ici va décomposer l'expression e en sous-expressions, par rapport à l'opérateur de plus haut niveau, et créer récursivement les automates pour chacune de ces sous-expressions, puis combiner les automates ainsi obtenus en fonction de l'opérateur pour obtenir l'automate A reconnaissant langage de e. Étant donné l'ECM e définie sur les ensembles de variables V et de couches L, l'automate construit A est défini sur les ensembles de variables V' et de couches L' contenant toutes celles utilisées par e : $V \subseteq V'$ et $L \subseteq L'$. Les variables et couches supplémentaires permettent de simuler certaines mécaniques des ECM, telles les constantes et l'itération $(*)$.

L'automate A est généré uniquement en fonction de l'expression e, et le contexte mémoire initial M_e de e est uniquement utilisé pour initialiser les variables de $V \times L$ du contexte mémoire initial de A. Les valeurs des variables de $(V' \times L') \backslash (V \times L)$, dans M_A, dépendent uniquement de e, indépendamment des valeurs de M_e. Lors de la reconnaissance, l'automate va reproduire l'évolution du contexte mémoire de e sur les variables de $V \times L$. Ainsi, si $w \in \mathbb{L}_{M_e}^{M_f}(e)$ alors il existe un chemin de transitions dans A acceptant w et dont l'état final atteint (q_f, M) vérifie $\forall X^l \in V \times L, M_f(X^l) = M(X^l)$.

Propriété 4.3.1 *Soit l'expression e définie sur l'ensemble de variables V et de couches L et l'automate correspondant $A = (Q, q_0, F, \Delta, V', L', M_A)$ construit par la méthode mentionnée ci-dessus. Étant donnés les contextes mémoire M_i et M_f,*

pour tout mot $w \in \mathbb{L}_{M_i}^{M_f}(e)$ *il existe un chemin de transitions de* Δ^* *acceptant* w *en allant de l'état* (q_0, M_i') *à* (q_f, M_f')*, où* $q_f \in F$*, tel que :*

$$\begin{array}{rl} & \forall X^l \ \in \ V \times L, M_i(X^l) \ = \ M_i'(X^l), M_f(X^l) = M_f'(X^l) \\ \wedge & \forall Y^k \in V' \times L' \backslash V \times L, M_i'(Y^k) \ = \ M_A(Y^k) \end{array}$$

.

Les expressions terminales

Les expressions terminales sont directement converties en un ACM représentant le même langage. Les preuves d'équivalence de langage y sont triviales, nous ne les détaillons pas.

Comme les ACM sont définis sur un alphabet infini $\mathcal{U}$, il n'y existe pas de notions de lettres constantes. Nous considérons donc que Σ est inclus dans $\mathcal{U}$ et utilisons le contexte mémoire initial de l'automate pour simuler la présence des constantes. Dans les ECM, elles ne peuvent pas être associées aux variables $V \times L$. Pour simuler cela dans les ACM, une variable est créée pour chaque constante $V_\Sigma = V \cup \Sigma$. Chaque constante est associée à la variable correspondante dans chaque couche, la rendant non-fraîche pour toutes les couches de l'automate : $M_\Sigma = \{c^l \to \{c\} \mid \forall c \in \Sigma\}$. Ainsi, l'ACM correspondant à l'ECM a, où $a \in \Sigma$ est $A_a = (\{q_0, q_f\}, q_0, \{q_f\}, \{(q_0, \emptyset, \alpha, \emptyset, q_f)\}, V_\Sigma, L, M_e \cup M_\Sigma)$ où $\forall l \in L, \alpha(l) = a$. Cet automate est composé de deux places, q_0 initiale et q_f finales, reliées par une unique transition franchissable lors de la lecture de la lettre a, qui est la seule lettre associée aux variables a de chaque couche de L.

Le langage vide, représenté par l'ECM $\emptyset$, est exprimé par l'automate $A_\emptyset = (\{q_0\}, q_0, \emptyset, \emptyset, V, L, M_e)$. Il est composé d'une unique place q_0 qui est sa place initiale, mais ne dispose d'aucune place finale, ni transition.

Le langage de l'ECM X^l représente l'ensemble des mots d'une lettre, qui est associée à la variable X de la couche l. Ce langage est représentable par l'automate $A_{X^l} = (\{q_0, q_f\}, q_0, \{q_f\}, \{(q_0, \emptyset, \alpha, \emptyset, q_f)\}, V_\Sigma, L, M_e \cup M_\Sigma)$, où $\alpha(l) = X$ et $\forall k \in L \backslash \{l\}, \alpha(k) = \sharp$. Cet automate est composé d'une unique transition allant de la place initiale à la place finale, elle est franchissable uniquement lors de la lecture d'une des lettres associées à X^l.

Le langage de l'ECM $\sharp X^l$ représente l'ensemble des mots d'une lettre, fraîche pour la couche l, et l'associe à la variable X^l. Ce langage est représentable par l'automate $A_\sharp = (\{q_0, q_f\}, q_0, \{q_f\}, \{(q_0, \{X^l\}, \alpha, \emptyset, q_f)\}, V_\Sigma, L, M_e \cup M_\Sigma)$, où $\alpha(l) = X$ et $\forall k \in L \backslash \{l\}, \alpha(k) = \sharp$. Cet automate est composé d'une unique transition allant de la place initiale à la place finale, franchissable en associant la lettre lue à X^l, cette lettre doit donc être fraîche pour la couche l. Comme le contexte mémoire initial contient les valeurs de M_Σ, la lettre est différente des constantes de Σ car elles sont présentes dans toutes les couches de L.

Les opérateurs du langage

Une expression non-terminale est composée d'une ou plusieurs sous-expressions sur lesquelles est appliqué un opérateur. Pour construire l'ACM correspondant à une ECM e non-terminale, nous construisons inductivement les automates correspondant à ses sous-expressions, puis nous construisons à partir d'eux l'ACM correspondant à e. Dans cette section nous présentons les différentes constructions utilisées pour construire cet ACM en fonction des différents opérateurs des ECM. Nous notons par e_1 et e_2 les sous-expressions et les automates correspondant sont respectivement : $A_1 = (Q_1, q_{0_1}, F_1, \Delta_1, V_1, L_1, M_1)$ et $A_2 = (Q_2, q_{0_2}, F_2, \Delta_2, V_2, L_2, M_2)$ et supposons qu'ils respectent la Propriété 4.3.1.

Le langage de l'ECM $e_1 \cdot e_2$ représentant la concaténation des langages de e_1 et e_2 est représenté par l'automate $A_{1\cdot 2} = (Q_1 \cup Q_2, q_{0_1}, F_2, \Delta_1 \cup \Delta_2 \cup \Delta_{1\cdot 2}, V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2)$, où $\Delta_{1\cdot 2} = \{(q, \nu, \alpha, \overline{\nu}, q_{0_2}) \mid (q, \nu, \alpha, \overline{\nu}, q_f) \in \Delta_1, q_f \in F_1\}$. L'automate est composé de toutes les places et transitions des automates A_1 et A_2, plus les transitions de $\Delta_{1\cdot 2}$ qui sont des copies de toutes les transitions finales de A_1 dont la destination a été changée pour la place initiale de A_2. La place initiale de $A_{1\cdot 2}$ est la place initiale de A_1 et ses places finales sont celles de A_2.

La sémantique de la concaténation (Table 3.3) indique que étant donné M, $\mathbb{L}_M^{M_f}(e_1 \cdot e_2) = \bigcup_{M'} \mathbb{L}_M^{M'}(e_1) \cdot \mathbb{L}_{M'}^{M_f}(e_2)$. Pour tous les mots $w \in \mathbb{L}_M(e_1 \cdot e_2)$ il existe M' tel que $w \in \mathbb{L}_M^{M'}(e_1) \cdot \mathbb{L}_{M'}(e_2)$. Ainsi, il est possible de décomposer $w = uv$ tel que $u \in \mathbb{L}_M^{M'}(e_1)$ et $v \in \mathbb{L}_{M'}(e_2)$. Par hypothèse d'induction sur A_1, il existe le chemin de transitions $\delta_1, \delta_2, \ldots, \delta_n \in \Delta_1$ tel qu'en lisant u il est possible d'aller de l'état (q_{0_1}, M_A) à (q_{f_1}, M_1'), où $q_{f_1} \in F_1$ et les contextes mémoire M_A et M_A' vérifient la Propriété 4.3.1. De même, par hypothèse d'induction sur A_2, il existe $\delta_1', \delta_2', \ldots, \delta_m' \in \Delta_2$ permettant en lisant v d'aller de l'état (q_{0_2}, M_A') à (q_{f_2}, M_{f_A}), où $q_{f_2} \in F_2$ et les contextes mémoire M_A' et M_{f_A} vérifient la Propriété 4.3.1. Par définition de $A_{1\cdot 2}$, il existe $\delta \in \Delta_{1\cdot 2}$ qui est une copie de δ_n dont la destination est la place q_{0_2}. En substituant δ_n par δ on obtient dans $A_{1\cdot 2}$ le chemin de transitions $\delta_1, \ldots, \delta_{n-1}, \delta$ allant de l'état (q_{0_1}, M_A) à (q_{0_2}, M_A') puis $\delta_1', \ldots \delta_m'$ allant jusqu'à l'état final (q_{f_2}, M_{f_A}) en lisant w. Nous pouvons en conclure que $\mathbb{L}_M(e_1 \cdot e_2) \subseteq \mathbb{L}_{M_A}(A_{1\cdot 2})$ et que la Propriété 4.3.1 est vérifiée entre $A_{1\cdot 2}$ et $e_1 \cdot e_2$. La preuve que $\mathbb{L}_{M_A}(A_{1\cdot 2} \subseteq \mathbb{L}_M(e_1 \cdot e_2)$ est similaire à celle ci, aussi est elle omise ici.

Le langage de l'ECM $e_1 + e_2$ représentant l'union des langages de e_1 et e_2 est représenté par l'automate $A_{1+2} = (Q_1 \cup Q_2 \cup \{q_0\}, q_0, F_1 \cup F_2, \Delta_1 \cup \Delta_2 \cup \Delta_{1+2}, V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2)$, où $\Delta_{1+2} = \{(q_0, \emptyset, \varepsilon, \emptyset, q_{0_i}) \mid i \in \{1, 2\}\}$. Les automates A_1 et A_2 sont intégralement conservés dans A_{1+2}, et leurs places finales le sont toujours dans l'automate créé. La place initiale q_0 et une nouvelle place, connectée par les ε-transitions de Δ_{1+2} aux places q_{0_1} et q_{0_2} afin d'accéder de manière non-déterministe aux automates A_1 et A_2.

La sémantique de l'union (Table 3.3) indique que pour tous M et M_f, $\mathbb{L}_M^{M_f}(e_1 + e_2) = \mathbb{L}_M^{M_f}(e_1) \cup \mathbb{L}_M^{M_f}(e_2)$. Pour tout mot $w \in \mathbb{L}_M^{M_f}(e_1)$, il existe un chemin de transition $\delta_1, \delta_2, \ldots, \delta_n \in \Delta_1$ acceptant le mot w en allant de l'état (q_{0_1}, M_A) à l'état final (q_f, M_{f_A}) dans A_1, où les contextes mémoire M_A et M_{f_A} vérifient la Propriété 4.3.1. Par définition de Δ_{1+2}, il existe l'ε-transition δ_{ε_1} allant de (q_0, M_A) à l'état (q_{0_1}, M_A) dans l'automate A_{1+2}, formant ainsi le chemin $\delta_{\varepsilon_1}, \delta_1, \delta_2, \ldots, \delta_n$ acceptant le mot w dans A_{1+2}. Le même processus peut être répété avec l'automate A_2 permettant ainsi de prouver que $\mathbb{L}_M(e_1 + e_2) \subseteq \mathbb{L}_{M_A}(A_{1+2})$. Pour tout mot $w \in \mathbb{L}_{M_A}(A_{1+2}$, il existe un chemin de transitions allant de la place q_0 à une place finale. Ce chemin débute forcément par une ε-transition de Δ_{1+2}, car ce sont les seules sortantes de q_0, puis une fois l'état q_{0_1} (resp. q_{0_2}) atteint, toutes les transitions suivantes du chemin peuvent uniquement faire partie de Δ_1 (resp. Δ_2) et forme donc un chemin acceptant w dans A_1 (resp. A_2). On peut donc en déduire que étant donné M vérifiant la Propriété 4.3.1 :

$$\mathbb{L}_{M_A}(A_{1+2}) \subseteq \mathbb{L}_{M_A}(A_1) \cup \mathbb{L}_{M_A}(A_2) = \mathbb{L}_M(e_1) \cup \mathbb{L}_M(e_2) = \mathbb{L}_M(e_1 + e_2).$$

Le langage de l'ECM $e_1 \& e_2$ est l'intersection des langages des expressions e_1 et e_2. L'automate est construit en se basant sur la construction de la preuve de la Propriété 4.2.2. L'opérateur & contraint les sous-expressions à ne pas utiliser les même couches, cependant ce n'est pas le cas des automates A_1 et A_2 qui les représentent car toutes les transitions issues des expressions de constantes utilisent toutes les couches de L. Mais, comme les transitions des constantes sont les seules à utiliser les variables des Σ, il est possible de les traiter séparément des autres transitions. L'automate correspondant à l'expression $e_1 \& e_2$ est $A_{1\&2}$ le produit synchrone des automates A_1 et A_2 : $A_{1\&2} = (Q_1 \times Q_2, (q_{0_1}, q_{0_2}), F_1 \times F_2, \Delta_X \cup \Delta_\Sigma \cup \Delta_{\varepsilon_1} \cup \Delta_{\varepsilon_2}, V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2)$ où :

- Les transitions de Δ' sont le résultat de la synchronisation des transitions représentant les événements de $\mathcal{U}$: $\Delta' = \{(q_1, q_2), \nu_1 \cup \nu_2, \alpha, \overline{\nu}_1 \cup \overline{\nu}_2, (q_1', q_2') \mid \forall i \in \{1, 2\}, (q_i, \nu_i, \alpha_i, \overline{\nu}_i, q_i') \in \Delta_i, \exists l \in L_i, \alpha_i(l) \notin \Sigma \cup \{\sharp\}\}$ où $\alpha(l) = \begin{cases} \alpha_1(l) & \text{si } l \in L_1 \\ \alpha_2(l) & \text{si } l \in L_2 \end{cases}$
- Les transitions de Δ_Σ sont le résultat de la synchronisation des transitions représentant les lettres de Σ : $\Delta_\Sigma = \{(q_1, q_2), \emptyset, \alpha, \overline{\nu}_1 \cup \overline{\nu}_2, (q_1', q_2') \mid \forall i \in \{1, 2\}, (q_i, \nu_i, \alpha, \overline{\nu}_i, q_i') \in \Delta_i, \exists c \in \Sigma, \forall l \in L_i, \alpha_i(l) = c\}$
- Les Δ_{ε_i}, où $i \in \{1, 2\}$, sont les ensembles de ε-transitions des automates. Elles ne sont pas synchronisées dans $A_{1\&2}$:

$$\Delta_{\varepsilon_1} = \{((q_1, q_2), \nu, \alpha, \overline{\nu}, (q_1', q_2)) \mid (q_1, \nu, \alpha, \overline{\nu}, q_1') \in \Delta_1\}$$
$$\Delta_{\varepsilon_2} = \{((q_1, q_2), \nu, \alpha, \overline{\nu}, (q_1, q_2')) \mid (q_2, \nu, \alpha, \overline{\nu}, q_2') \in \Delta_2\}$$

Cette construction reprend celle de la preuve de la Propriété 4.2.2, pour prouver que le produit synchrone de deux ACM représente l'intersection de leurs langages.

Nous n'allons donc pas détailler de nouveau l'intégralité de la preuve, car les transitions de Δ_Σ ne peuvent pas modifier le contexte mémoire, et aucune des transitions de Δ' et des Δ_{ε_1} et Δ_{ε_2} n'interagissent avec les variables de $\Sigma \times L$. De même, Δ_Σ représente la synchronisation des transitions représentant les constantes, et ces transitions sont synchronisées entre elles car elles seules permettent de représenter les lettres de Σ. Ainsi il est facile d'en conclure que $\mathbb{L}_M(e_1\&e_2) = \mathbb{L}_{M_A}(A_{1\&2})$, où les contextes mémoire vérifient la Propriété 4.3.1.

Pour montrer que la construction vérifie la Propriété 4.3.1, il faut se rappeler que les couches mémoire ont un fonctionnement indépendant les une des autres et que les couches utilisées dans chaque expressions sont disjointes, c'est-à-dire que si e_1 est composée avec une sous-expression terminale X^l ou $\sharp X^l$, alors il n'existe aucune sous-expression terminale utilisant la couche l dans e_2, et ce peu importe la variable. Nous savons que les transitions de Δ_1 et Δ_2 préservent la propriété, et que la synchronisation des transitions consiste à effectuer l'intégralité des opérations des deux transitions dans la transition créée. Ces opérations n'agissant pas sur les mêmes couches et étant donc indépendantes entre les deux transitions, alors la Propriété 4.3.1 est bien vérifiée par $A_{1\&2}$.

Le langage de l'ECM $e_1^*/\bar{l}/$ représente l'ensemble des mots formés de la concaténation d'un nombre arbitraire de mots du langage de e_1. À la différence de l'étoile de Kleene, la concaténation de langages à mémoire laisse la possibilité d'effectuer un effet de bord entre les contextes mémoire des langages concaténés. Le paramètre $/\bar{l}/$ représente l'ensemble des couches mémoires subissant un effet de bord entre chaque itération, ce qui signifie que les lettres associées à ces variables à la fin d'une itération y sont toujours associées au début de la suivante. Le début de la preuve de la propriété 4.2.4 présente une construction permettant, à partir d'un automate A, d'obtenir l'automate représentant le langage $\mathbb{L}(A)^*$. Cette construction présente le mécanisme permettant de réinitialiser les variables à la fin de chaque itération en gardant en mémoire les valeurs initiales des variables et en effectuant les modifications qui devraient y avoir lieu sur les nouvelles couches L_w. Dans le cas de l'expression $e_1^*/\bar{l}/$, les variables des couches $\bar{l}$ ne sont pas réinitialisées à la fin d'une itération, ainsi il n'est pas nécessaire de préserver leurs valeurs initiales au cours de l'itération et donc de créer des couches correspondantes dans L_w.

Pour éviter d'être redondant avec la preuve de la Propriété 4.2.4, nous ne détaillerons pas formellement cette construction. L'idée de la construction consiste de modifier A_1, l'automate correspondant à l'expression e_1, en ajoutant des couches supplémentaire (L_w) mais uniquement pour les couches $l \notin \bar{l}$. Ainsi, les places et transitions sont modifiées, de la même manière que dans la preuve de la propriété 4.2.4, mais uniquement vis-à-vis des variables appartenant à ces couches. Tandis que les variables des couches de $\bar{l}$ peuvent être utilisées de la même manière que dans A_1, ainsi les annotations correspondantes dans les transitions ne sont pas

modifiées. Ainsi, la Propriété 4.3.1 est vérifiée par la construction de l'itération, car toutes les variables X^l, telles que $l \in \bar{l}$, sont utilisées de la même manière que dans les transitions de Δ_1 et subisse des effets de bord entre chaque itération, comme indiqué par la sémantique. De même les valeurs des variables X^l où $l \in L \backslash \bar{l}$ ne sont pas modifiées lors du franchissement des transitions de l'automate construit, ce qui vérifie bien la sémantique de l'itération.

L'opérateur de réinitialisation $e_1\overline{X^l}!$ qui s'applique en suffixe d'une expression e_1, représente le même ensemble de mots que e_1, mais les variables de $\overline{X^l}$ sont réinitialisées dans le contexte mémoire final. Étant donné l'automate A_1 équivalent à l'expression e_1, on peut construire l'automate $A_{1!} = (Q_1 \cup \{q_f\}, q_{0_1}, \{q_f\}, \Delta_1 \cup \Delta_!, V_1, L_1, M_1)$ où :

$$\Delta_! = \{(q, \nu, \alpha, \overline{\nu} \cup \overline{X^l}, q_f) \mid (q, \nu, \alpha, \overline{\nu}, q') \in \Delta_1, q' \in F\}$$

Cet automate est créé en ajoutant une nouvelle place q_f qui est désignée comme l'unique place finale. Une copie de chaque transition finale de A_1 est créée dont la destination est q_f. Ces transitions réinitialisent toutes les variables de $\overline{X^l}$. Il est facile de voir que l'automate $A_{1!}$ vérifie la Propriété 4.3.1 par rapport à l'expression $e_1\overline{X^l}!$. En effet, par définition de A_1, il reconnaît l'ensemble des mots représentés par e_1. Cela signifie qu'il existe un chemin de transitions dans Δ_1 menant à une place finale qui peut être franchi lors de la lecture de tout mot de e_1. Ce même chemin de transition existe dans $A_{1!}$ et il est possible d'intervertir sa transition finale par la transition qui en est une copie dans $\Delta_!$. Ce nouveau chemin est aussi franchissable, mais dans son état final, toutes les variables de $\overline{X^l}$ sont réinitialisées.

L'opérateur de décalage de couche $e_1 \uparrow_n$ est une réécriture dynamique des identifiants des couches utilisées dans l'expression. Modifier le nom des couches modifie la manière dont est manipulé le contexte mémoire par l'expression. Cela se traduit naturellement dans les automates en renommant de la même manière les couches dans les annotations des transitions. Ainsi, étant donné l'automate A_1 correspondant à l'expression e_1, on peut construire l'automate correspondant à l'expression $e_1 \uparrow_n : A_\uparrow = (Q_1, q_{0_1}, F_1, \Delta_\uparrow, V_1, \{l + n \mid l \in L_1\}, M_1)$ où :

$$\Delta_\uparrow = \{q, \{X^{l+n} \mid X^l \in \nu\}, \alpha_\uparrow, \{X^{l+n} \mid X^l \in \overline{\nu}\}, q' \mid (q, \nu, \alpha, \overline{\nu}, q') \in \Delta_1\}$$

avec $\alpha_\uparrow(l + n) = \alpha(l)$. La réécriture des indentifiants des couches ne modifie pas les propriétés de l'automate, et les couches de l'expression étant renommées dynamiquement alors la Propriété 4.3.1 est bien vérifiée.

L'opérateur de masquage de lettre $(e_1)_{\backslash a}$ permet de cacher la lettre $a \in \Sigma$ des mots représentés par l'expression e_1. Cet opérateur est principalement utilisé avec les différents opérateurs de temps des ETCM, mais il est aussi utilisé afin d'exprimer le mot vide ε. Étant donné l'automate A_1 correspondant à l'expression e_1, on peut construire l'automate $A_{\backslash a}$ correspondant à l'expression $(e_1)_{\backslash a}$ en transformant

en ε-transitions toutes les transitions de A_1 franchissables lors de la lecture de a. Ainsi, $A_{\backslash a} = (Q_1, q_{0_1}, F_1, \Delta_{\backslash a}, V_1, L_1, M_1)$ où :

$$\begin{aligned}\Delta_{\backslash a} = \ & (\Delta_1 \backslash \{(q, \nu, \alpha, \overline{\nu}, q') \in \Delta_1 \mid \forall l \in L, \alpha(l) = a\}) \\ & \cup \{(q, \nu, \varepsilon, \overline{\nu}, q') \in \Delta_1 \mid (q, \nu, \alpha, \overline{\nu}, q') \in \Delta_1, \forall l \in L, \alpha(l) = a\}\end{aligned}$$

Pour rappel, L est l'ensemble des couches sur lesquelles l'expression est définie. Dans la construction correspondant à la traduction de la sous-expression d'une constante a, la transition générée associe avec α toute les couches à la variable a.

Prouvons que cette construction vérifie la propriété 4.3.1 avec l'expression $(e_1)_{\backslash a}$ sachant que par hypothèse d'induction l'automate A_1 vérifie la propriété avec l'expression e_1. Étant donné un chemin de transitions $\delta_1, \delta_2, \ldots, \delta_n \in \Delta_1$ franchissable lors de la lecture d'un mot $w = w_1 w_2 \ldots w_n \in \mathcal{U} \cup \{\varepsilon\}$ menant de l'état (q_{0_1}, M_1) à l'état (q_f, M_f) où $q_f \in F_1$, il existe un chemin de transitions $\delta'_1, \ldots \delta'_n \in \Delta_{\backslash a}$ tel que pour tout $i \in [1, n]$, si $\delta_i = (q, \nu, \alpha, \overline{\nu}, q')$ où $\forall l \in L, \alpha(l) = a$ alors $\delta'_i = (q, \nu, \varepsilon, \overline{\nu}, q')$ sinon $\delta_i = \delta'_i$ par définition de $\Delta_{\backslash a}$. Le chemin de transitions $\delta'_1 \ldots \delta'_n$ est franchissable lors de la lecture du mot $w'_1 w'_2 \ldots w'_n$ tel que pour tout $i \in [1, n]$ si $w_i = a$ alors $w'_i = \varepsilon$ sinon $w_i = w'_i$, et mène de l'état (q_{0_1}, M_1) à l'état (q_f, M_f) car la lecture d'une lettre constante comme d'une variable ne modifie pas le contexte mémoire.

4.3.2 ACM vers ECM

Pour montrer l'équivalence entre ACM et ECM, il faut également montrer que pour tout ACM A, il existe une ECM e et un contexte mémoire M tel que $\mathbb{L}(A) = \mathbb{L}_M(e)$. Pour ce faire nous allons présenter un algorithme permettant de générer une ECM à partir d'un ACM. À partir de cet algorithme, nous allons pouvoir présenter la propriété suivante :

Propriété 4.3.2 *Pour tout ACM A, il existe une ECM e telle que pour tout contexte mémoire M, $\mathbb{L}_M(A) = \mathbb{L}_M(e)$.*

Nous utilisons l'algorithme dit de *suppression des positions* pour générer une expression équivalente à un automate donné par des itérations successives. Cet algorithme est inspiré de celui de conversion d'un automate fini en expression régulière présenté dans [53, 28]. À partir de l'ACM, la première étape consiste à traduire d'abord toutes les annotations des transitions en expressions équivalentes [1]. Ensuite, les places de l'automate sont successivement supprimées, et remplacées par de nouvelles transitions annotées par la composition des expressions annotant

1. Ces expressions représentent l'ensemble des mots d'une lettre permettant de franchir la transition.

les transitions passant par ces places. Cette contraction se poursuit jusqu'à ce qu'il ne reste plus qu'une unique transition annotée par une expression équivalente à l'automate de départ.

L'automate annoté avec des ECM

Le modèle d'automate intermédiaire utilisé dans l'algorithme est celui dont les transitions sont annotées directement par des ECM. Ainsi nous définissons les AECM comme une variante des ACM dont la seule différence est que ses transitions sont des triplets de la forme :

$$(q, e, q') \in \Delta$$

tels que q est la place d'origine, q' la place de destination de la transition, et e est une ECM.

Les états dans les AECM sont aussi des paires (q, M) où $q \in Q$ est une place de l'automate et M est un contexte mémoire défini sur les ensembles de variables V et de couches L. Il existe une transition d'un état (q, M) à un état (q', M') si et seulement s'il existe une transition $(q, e, q') \in \Delta$. La transition est franchissable lors de la lecture d'un mot $w \in \mathbb{L}_M^{M'}(e)$.

Soit un AECM $A_E = (Q, q_0, F, \Delta, V, L, M_0)$ et un mot $w \in \mathcal{U}^*$ décomposable en $w = w_1 w_2 \dots w_n$. Le mot w appartient au langage de A_E (noté $w \in \mathbb{L}(A_E)$) si il existe un chemin de transition $\delta_1, \delta_2, \dots \delta_n \in \Delta$ tel que :

$$(q_0, M_0) \xrightarrow[\delta_1]{w_1} (q_1, M_1) \xrightarrow[\delta_2]{w_2} \cdots \xrightarrow[\delta_n]{w_n} (q_n, M_n)$$

où l'état (q_n, M_n) est final, c'est-à-dire que $q_n \in F$.

L'algorithme de traduction d'ACM en ECM

L'algorithme 1 présente le pseudo code de l'algorithme de suppression des places utilisé pour convertir un ACM en ECM. Il est composé de trois étapes successive, l'étape de traduction (lignes 1 à 3) convertissant l'ACM en AECM, l'étape de simplification (lignes 5 à 8) et l'étape de suppression de place (lignes 9 à 17). Les deux dernières étapes permettent de *fusionner* les transitions et sont répétées tant qu'il reste des places dans l'automate d'origine. Lorsque toute les places de l'automate ont été supprimées, il reste une unique transition dans Δ_E qui est annoté avec une expression représentant le même langage que celui reconnu par l'ACM. L'expression créée est définie sur les mêmes ensemble de couches L et de variables V que l'automate et sur l'alphabet fini de constante $\Sigma = \{\epsilon\}$ contenant la lettre $\epsilon \notin \mathcal{U}$ qui est masquée dans l'expression résultant de l'algorithme. La lettre ϵ permet de représenter les ε-transitions et donc le mot vide dans e.

Algorithme 1 Créer une ECM équivalente à un ACM donné

Entrée: Un ACM $A = (Q, q_0, F, \Delta, V, L, M_0)$
Sortie: $\Delta_E = \{(q'_0, e, q_f)\}, \forall M, \mathbb{L}_M(e_{\setminus\epsilon}) = \mathbb{L}_M(A)$

1: $\Delta_E \leftarrow$ traduction des transitions de Δ en transitions d'AECM
2: $\Delta_E \leftarrow \Delta_E \cup \{(q'_0, \epsilon, q_0)\} \cup \{(q, \epsilon, q_f) \mid q \in F\}$
3: $A_E \leftarrow (Q \cup \{q'_0, q_f\}, q'_0, \{q_f\}, \Delta_E, V, L, M_0)$
4: **tant que** $Q \neq \emptyset$ **faire**
5: **tant que** $\exists (q, e_1, q'), (q, e_2, q') \in \Delta_E$ où $e_1 \neq e_2$ **faire**
6: $\Delta_E \leftarrow \Delta_E \backslash \{(q, e_1, q'), (q, e_2, q')\}$
7: $\Delta_E \leftarrow \Delta_E \cup \{(q, e_1 + e_2, q')\}$
8: **fin tant que**
9: Choisir arbitrairement $q \in Q$
10: $Q \leftarrow Q \backslash \{q\}$
11: **pour tout** $(q_i, e_i, q), (q, e_o, q_o) \in \Delta_E$ où $q_i \neq q$ et $q_o \neq q$ **faire**
12: **si** $\nexists (q, e_r, q) \in \Delta_E$ **alors**
13: $e_r \leftarrow \emptyset$
14: **fin si**
15: $\Delta_E \leftarrow \Delta_E \cup \{(q_i, e_i \cdot e_r^* / L / \cdot e_o)\}$
16: **fin pour**
17: $\Delta_E \leftarrow \Delta_E \backslash \{(q_1, e, q_2) \mid q = q_1 \vee q = q_2\}$
18: **fin tant que**

Étant donné un ACM $A = (Q, q_0, F, \Delta, V, L, M_0)$ pour lequel on souhaite trouver l'expression équivalente. L'algorithme débute par la traduction de A en l'AECM A_E équivalent, représenté par les lignes 1 à 3 de l'algorithme 1. On crée l'AECM $A_E = (Q \cup \{q_0, q_f\}, q'_0, \{q_f\}, \Delta_E, V, L, M_0)$ où $\Delta_E = \Delta' \cup \Delta_\varepsilon$ est l'ensemble de transitions de la forme (q, e, q') où e est une ECM. Les transitions de Δ' sont la traduction des transitions de A et Δ_ε sont les ε-transitions reliant les deux nouvelles places q_0 et q_f aux places initiales et finales de A.

$$\begin{aligned}\Delta' = \; & \{(q, \epsilon\{\overline{\nu}\}!, q') \mid (q, \nu, \varepsilon, \overline{\nu}, q') \in \Delta\} \cup \\ & \{(q, C\{\overline{\nu}\}!, q') \mid (q, \nu, \alpha, \overline{\nu}, q') \in \Delta, \alpha \neq \varepsilon, \\ & \qquad C = \Big(\mathop{\&}_{\alpha(l) = X} X^l\Big) \& \Big(\mathop{\&}_{\substack{\alpha(l) = X \\ X^l \in \nu}} \sharp X^l\Big)\}\end{aligned}$$

Chaque transition $(q, e, q') \in \Delta'$ est la traduction d'une transition $(q, \nu, \alpha, \overline{\nu}, q')$ de A. Les expressions sont de la forme $e = C\{\overline{\nu}\}!$ où C est la conjonction des variables utilisées pour chaque couche, par exemple $C = X^i \& Y^j \& \sharp Z^k \ldots$ où $X^i, Y^j, \ldots$ sont les variables utilisées par α et absentes de ν et Z^k est utilisée par α et fait partie de ν. Ainsi, C est seulement composée de variables de couches différentes, et pour

toute couche l telle que $\alpha(l) = \sharp$ il n'existe pas de variable de la couche l dans C. Les ε-transitions sont un cas particulier où la transition est franchissable lors de la lecture du mot vide. Cela est représenté dans les ECM par l'utilisation de l'opérateur de masquage, ainsi l'annotation de la transition est traduite par $\epsilon\{\overline{\nu}\}!$ où la lettre ϵ est masquée dans l'expression finale. Le suffixe $\overline{\nu}!$ de l'expression exprime la réinitialisation des variables de $\overline{\nu}$.

En plus des places de Q, deux nouvelles places sont ajoutées à A_E, la place initiale q'_0 et l'unique place finale q_f. Ces deux places sont reliées via des ε-transitions de Δ_ε aux anciennes places initiale q_0 et finales F.

$$\Delta_\varepsilon = \{(q'_0, \epsilon, q_0)\} \cup \{(q, \epsilon, q_f) \mid q \in F\}$$

Il contient une transition de q'_0 à q_0 et une transitions de chaque position de F vers la nouvelle position finale q_f. Toutes ces transitions sont annoté par ϵ qui est la lettre masquée dans le langage finale. Ces ajouts permettent de s'assurer qu'aucune transition n'entre dans la place initiale et ne sorte de la place finale de l'automate. Cela nous assure qu'à la fin de l'algorithme l'automate sera composé d'une unique transition, allant de q'_0 à q_f, qui sera annotée par une ECM représentant le même langage que A.

Les deux étapes suivantes de l'algorithme sont répétées tant qu'il reste des places dans A_E autre que q'_0 et q_f.

La première sous-étape consiste à simplifier l'automate A_E en fusionnant les transitions qui partagent les mêmes origine et destination, représenté par les lignes 5 à 8 de l'algorithme 1. Pour toutes paires de places $q, q' \in Q$ (potentiellement les mêmes) et l'ensemble $\Delta_{q,q'} \subseteq \Delta'$ des transitions allant de q à q' et leurs expressions $e_1, e_2, \ldots$, on crée la transition $\delta_{q,q'} = (q, e_1 + e_2 + \cdots, q')$ dont l'expression est la disjonction des expressions des transitions de $\Delta_{q,q'}$. On supprime de l'automate toutes les transitions de $\Delta_{q,q'}$ qui sont remplacées par $\delta_{q,q'} : \Delta' ::= (\Delta' \backslash \Delta_{q,q'}) \cup \{\delta_{q,q'}\}$.

La seconde sous-étape, représentée par les lignes 9 à 17 de l'algorithme 1, consiste à sélectionner arbitrairement une place $q \in Q$ qui sera éliminée de l'automate. La suppression de q de l'automate, entraîne alors la création de nouvelles transitions en composant ses transitions entrantes avec ses sortantes. Pour chaque paire de places $q_i, q_o \in Q$ différentes de q telles qu'il existe des transitions (q_i, e_i, q) et (q, e_o, q_o), nous souhaitons créer une transition allant directement de q_i à q_o sans passer par q. S'il existe une transition qui boucle sur q, $(q, e_r, q) \in \Delta'$, il est nécessaire de la prendre en compte dans la création de la nouvelle transition, car tout chemin passant par q peut emprunter cette transition plusieurs fois. Sans perte de généralité, on peut supposer qu'il existe toujours une transition qui boucle pour toutes places de Q dont l'expression serait $e_r = \varepsilon$.

La nouvelle transition $(q_i, e_i \cdot e_r^*/L/ \cdot e_o, q_o)$ remplace ces deux transitions et simule les chemins allant de q_i à q_o en passant par q, et pouvant boucler sur q.

L'itération $(*)$ appliquée sur l'expression e_r est paramétrée par toutes les couches, car toutes les couches subissent un effet de bord d'un pas d'itération à l'autre. Une fois toutes les nouvelles transitions créées pour chaque paire d'états, la place q et toutes les transitions entrantes et sortantes de q sont supprimées de Q et Δ'.

Tant qu'il reste des places dans A_E autre que q'_0 et q_f alors on répète les deux étapes précédentes de simplification et de suppression des places. Si toutes les places de Q ont été supprimées de l'automate, alors il reste une unique transition (q'_0, e, q_f) dans A_E telle que pour tout contexte mémoire M alors $\mathbb{L}_M(e) = \mathbb{L}_M(A)$.

La preuve de correction de l'algorithme

La preuve consiste à montrer qu'après chaque étape de l'algorithme $\mathcal{L}_M(A) = \mathcal{L}_M(A_E)_{\setminus\epsilon}$ pour tout contexte mémoire M. Si cela est vrai alors à la fin de l'algorithme, cela signifie que l'expression obtenue est bien équivalente au langage de A.

Initialement, la première version de A_E est construite en traduisant les transitions de A et garde la même structure, à l'exception de l'ajout des places q'_0 et q_f et des transitions de Δ_ε. L'ajout de ces places et de Δ_ε ne modifie pas le langage de A_E car Δ_ε contient uniquement des transitions annotées par ϵ et il n'existe aucune transition entrante dans q'_0 et sortante de q_f. Ainsi, tout chemin acceptant dans A_E débute par la transition (q'_0, ϵ, q_0), suivi d'un chemin de transition de Δ' allant de q_0 à un $q \in F$ et se termine par une des transitions (q, ϵ, q_f). Il suffit donc de montrer que la traduction des transitions de Δ en Δ' est correcte, c'est-à-dire représente le même espace d'états et que les transitions sont franchissables lors de la lecture des mêmes lettres ou d'ϵ dans le cas des ε-transitions.

Étant donné une transition $\delta = (q, \nu, \alpha, \overline{\nu}, q') \in \Delta$ de l'ACM, elle permet d'aller d'un état (q, M) à un état (q', M') lors de la lecture d'une lettre c si et seulement si c est associée à toute variable X^l tel que $\alpha(l) = X \neq \sharp$ et $X^l \notin \nu$ et si c n'est associée à aucune des variables des couches k tel que $\alpha(k) = X$ et $X^k \in \nu$, soit formellement :

$$c \in \bigcap_{\substack{l \in L \\ \alpha(l) = X \neq \sharp \\ X^l \notin \nu}} M(X^l) \bigcap_{\substack{k \in L \\ \alpha(k) = X \neq \sharp \\ X^k \in \nu}} (\mathcal{U} \setminus \bigcup_{Y \in V} (Y^k)) \tag{4.1}$$

De plus, les variables de $\overline{\nu}$ ont été vidées dans le contexte mémoire M' de l'état de destination, $\forall X^l \in \overline{\nu}, M'(X^l) = \emptyset$ et les variables qui n'ont pas été réinitialisées sont associées aux mêmes lettres que dans l'état d'origine, saufs celles auxquelles c a aussi été associée : $\forall X^l \in \nu, \alpha(l) = X, M'(X^l) = M(X^l) \cup \{c\}$.

La transition correspondante à δ dans Δ' est $\delta' = (q, e, q')$, annotée par l'ECM $e = C\overline{\nu}!$ où C est formée par la conjonction des expressions terminales :

— X^l pour chaque $\alpha(l) = X \neq \sharp$ où $X^l \notin \nu$, représentant l'ensemble de lettres $M(X^l)$
— $\sharp X^l$ pour chaque $\alpha(l) = X \neq \sharp$ où $X^l \in \nu$, représentant l'ensemble de lettres $\mathcal{U} \backslash \bigcup_{Y \in V} M(Y^l)$

Comme la conjonction exprime l'intersection des langages des sous-expressions, alors e représente l'intersection des langages des expressions terminales décrite ci-dessus, ce qui correspond à l'ensemble des mots de une lettre décrite par 4.1. Ainsi Δ' permet d'aller d'un état (q, M) à un état (q', M') lors de la lecture d'une lettre c vérifiant l'équation 4.1, où M' est un contexte mémoire final de $\mathbb{L}_M(e)$. Par la sémantique de la réinitialisation (Table 3.4), aucune lettre n'est associée aux variables de $\overline{\nu}$, $\forall X^l \in \overline{\nu}, M'(X^l) = \emptyset$. De même, par celle de la conjonction, les valeurs de chaque couche du contexte mémoire final sont égales aux valeurs de cette couche pour la sous-expression qui l'utilise. Ainsi, pour toutes les sous-expressions X^l, correspondantes aux variables telles que $\alpha(l) = X$ et $X^l \notin \nu$, toutes les variables de la couche l sont associées aux mêmes lettres que dans M, sauf pour celles ayant été réinitialisées : $\forall Y \in V, Y^l \notin \overline{\nu} \implies M'(Y^l) = M(Y^l)$. Pour toutes les sous-expressions $\sharp X^l$, correspondantes aux variables telles que $\alpha(l) = X$ et $X^l \in \nu$, seule X^l a sa valeur modifiée, où c y est associée, sauf pour les lettres réinitialisées : $X^l \notin \overline{\nu} \implies M'(X^l) = M(X^l) \cup \{c\}$ et $\forall Y \in V \backslash \{X\}, Y^l \notin \overline{\nu} \implies M'(Y^l) = M(Y^l)$.

On peut donc en conclure que Δ' est une traduction correcte de Δ, modulo ϵ. Il existe un chemin de transition dans Δ acceptant un mot w si et seulement si il existe un chemin de transitions dans Δ' traversant les mêmes états lors de la lecture d'un même mot w' tel que $w'_{\backslash \epsilon} = w$. Ainsi, pour tout M, la propriété $\mathcal{L}_M(A) = \mathcal{L}_M(A_E)_{\backslash \epsilon}$ est vérifiée.

La seconde étape de l'algorithme est la simplification, la première des deux sous-étapes qui sont répétées jusqu'à la fin de l'algorithme. L'étape de simplification consiste à modifier l'automate afin que pour toutes paires de places $(q, q') \in Q^2$ il y ait au plus une transition de la forme $(q, e, q') \in \Delta'$. Cette étape permet d'assurer l'unicité de la dernière transition à la fin de l'algorithme et d'éviter une explosion du nombre de transitions dans l'étape suivante.

Pour rappel, la simplification consiste, pour toutes paires de places $(q, q') \in Q^2$, de supprimer de Δ' le sous-ensemble des transitions $\Delta'_{q,q'}$ allant de la place q à q'. Il est remplacé par une nouvelle transition (q, e_+, q') où e_+ est une expression composée de la disjonction $(+)$ des expressions des transitions de $\Delta'_{q,q'}$. Pour prouver que l'automate A_E représente le même langage après l'étape de simplification, il faut démontrer que les nouvelles transitions, celles qui remplacent les ensembles $\Delta_{q,q'}$, représentent les mêmes transitions dans l'espace d'états.

Soit la transition (q, e_+, q') remplaçant l'ensemble de transitions $\Delta'_{q,q'}$, et soit les états (q, M) et (q', M') tels qu'il existe un mot w vérifiant $(q, M) \xrightarrow[(q,e_+,q')]{w} (q', M')$.

Cela signifie que $w \in \mathbb{L}_M^{M'}(e_+)$, par la sémantique de la disjonction (Table 3.3) nous pouvons déduire qu'il existe une sous-expression e de e_+ telle que $w \in \mathbb{L}_M^{M'}(e)$ où e est une expression annotant $(q, e, q') \in \Delta'_{q,q'}$, par définition de e_+. Nous pouvons donc en déduire que $(q, M) \xrightarrow[(q,e,q')]{w} (q', M')$ et donc $w \in \mathbb{L}_M^{M'}(e_+) \implies \exists (q, e, q') \in \Delta'_{q,q'}, w \in \mathbb{L}_M^{M'}(e)$.

De même dans l'autre sens, supposons qu'il existe un transition $(q, e, q') \in \Delta'_{q,q'}$ vérifiant pour un mot w' : $(q, M) \xrightarrow[(q,e,q')]{w'} (q', M')$. Cela signifie que $w' \in \mathbb{L}_M^{M'}(e)$, donc par la sémantique de la disjonction, $w' \in \mathbb{L}_M^{M'}(e_+)$ et donc $(q, M) \xrightarrow[(q,e_+,q')]{w'} (q', M')$. Nous pouvons donc en conclure que les transitions (q, e_+, q') reproduisent le comportement des ensembles de transitions $\Delta'_{q,q'}$ sur l'espace d'états. Ainsi, l'automate A_E représente le même langage avant et après l'étape de simplification.

La troisième étape est la suppression de place, qui est l'autre sous-étape répétée par l'algorithme. Elle consiste à choisir arbitrairement et supprimer une place $q \in Q$ de A_E et de fusionner ses transitions entrantes et sortantes afin de créer des transitions plus directes (ne passant pas par q) mais représentant le même langage. Ainsi, pour toute paire de places différentes de q, $(q_i, q_o) \in (Q \backslash \{q\})^2$, telle qu'il existe des transitions (q_i, e_i, q) et (q, e_o, q') dans Δ', une nouvelle transition $\delta_{q_i,q_o} = (q_i, e_i \cdot e_l^*/L/ \cdot e_o, q_o)$ est ajoutée dans Δ', où e_l est l'expression de la transition qui boucle (q, e_l, q) (pour généraliser : $e_l = \epsilon$ s'il n'en existe pas). Une fois les transitions δ_{q_i,q_o} créées pour toutes paires d'états, la place q est supprimée de Q ainsi que toutes les transitions dont l'origine ou la destination est q.

Pour prouver que l'automate A_E représente le même langage au début et à la fin de l'étape nous allons démontrer que pour tout chemin de transitions passant par q, il est possible de substituer chaque sous chemin passant par q par l'une des nouvelles transitions. Nous souhaitons donc prouver qu'étant donné le couple de places $(q_i, q_o) \in (Q \backslash \{q\})^2$ et la transition δ_{q_i,q_o} créée lors de la suppression de la place q, alors pour tous contextes mémoire M_i, M_o et pour tout mot $w_i w_1 w_2 \dots w_n w_o$ où $n \in \mathbb{N}$:

$$\begin{aligned}&(q_i, M_i) \xrightarrow[(q_i,e_i,q)]{w_i} (q, M_0) \xrightarrow[(q,e_l,q)]{w_1} (q, M_1) \xrightarrow[(q,e_l,q)]{w_2} \cdots \xrightarrow[(q,e_l,q)]{w_n} (q, M_n) \xrightarrow[(q,e_o,q_o)]{w_o} (q_o, M_o) \\ &\Leftrightarrow (q_i, M_i) \xrightarrow[\delta_{q_i,q_o}]{w_i w_1 \dots w_n w_o} (q_o, M_o)\end{aligned} \tag{4.2}$$

Prouvons d'abord le sens $\implies$ de l'équivalence 4.2. Par l'existence du chemin de transitions acceptant les mots w_i, $w_1, \dots, w_n$ et w_o, nous savons que $w_i \in \mathbb{L}_{M_i}^{M_0}(e_i)$, $\forall j \in [1, n], w_j \in \mathbb{L}_{M_{j-1}}^{M_j}(e_l)$ et $w_o \in \mathbb{L}_{M_n}^{M_o}(e_o)$. Par la sémantique de l'itération (Table 3.3), nous déduisons que $w_1 w_2 \dots w_n \in \mathbb{L}_{M_0}^{M_n}(e_l^l/L/)$, car l'opérateur d'itération est paramétré par L, ainsi l'effet de bord a lieu d'une itération à l'autre pour toutes

les variables. De même, par la sémantique de la concaténation, nous déduisons que $w_i w_1 w_2 \dots w_n w_o \in \mathbb{L}_{M_i}^{M_0}(e_i) \cdot \mathbb{L}_{M_0}^{M_n}(e_l^l/L/) \cdot \mathbb{L}_{M_n}^{M_o}(e_o) \subseteq \mathbb{L}_{M_i}^{M_o}(e_i \cdot e_l^*/L/ \cdot e_o)$. Ainsi, la transition δ_{q_i,q_o} vérifie bien $(q_i, M_i) \xrightarrow[\delta_{q_i,q_o}]{w_i w_1 \dots w_n w_o} (q_o, M_o)$.

Prouvons l'autre sens $\Leftarrow$, de l'équivalence 4.2. Notons w le mot tel que $(q_i, M_i) \xrightarrow[\delta_{q_i,q_o}]{w} (q_o, M_o)$. Par définition de δ_{q_i,q_o}, $w \in \mathbb{L}_{M_i}^{M_o}(e_i \cdot e_l^*/L/ \cdot e_o)$, ainsi par la sémantique de la concaténation, le mot w peut être décomposé en trois sous mots $w = w_i w_l w_o$ tel qu'il existe les contextes mémoire M_0 et M_n tels que $w_i \in \mathbb{L}M_i^{M_l}(e_i)$, $w_l \in \mathbb{L}_{M_0}^{M_n}(e_l^*/L/)$ et $e_o \in \mathbb{L}_{M_l'}^{M_o}(e_o)$. Par la sémantique de $*$, w_l est décomposable en $n \in \mathbb{N}$ sous mots, $w_l = w_{l_1} w_{l_2} \dots w_{l_n}$, où si $n = 0$ alors $w_l = \varepsilon$, tel qu'il existe des contextes mémoire $M_1, M_2, \dots, M_{n-1}$ où $\forall j \in [1, n], w_{l_j} \in \mathbb{L}_{M_{j-1}}^{M_j}(e_j)$. On peut ainsi en déduire que la lecture du mot w permet de franchir le chemin : $(q_i, M_i) \xrightarrow[(q_i,e_i,q)]{w_i} (q, M_0) \xrightarrow[(q,e_l,q)]{w_{l_1}} \cdots \xrightarrow[(q,e_l,q)]{w_{l_1}} (q, M_n) \xrightarrow[(q,e_o,q_o)]{w_o} (q_o, M_o)$. Ce qui prouve le sens $\Leftarrow$ de l'équivalence 4.2.

Ainsi l'automate A_E reconnaît le même langage au début et à la fin de l'étape de suppression de place. Nous pouvons ainsi conclure qu'à la fin de chaque étape de l'algorithme, le langage représenté par A_E représente toujours le même langage tel que $\mathbb{L}_M(A_E)_{\setminus\epsilon} = \mathbb{L}_M(A)$ pour tout contexte mémoire initial M. Donc à la fin de l'algorithme, l'unique transition restante sera annotée avec une ECM e telle que $\mathbb{L}_M(A) = \mathbb{L}_M(e)_{\setminus\epsilon}$, pour tout M.

4.4 Les automates temporisés à couches mémoire

Afin de pouvoir exprimer des contraintes temporisées dans les ACM, nous définissons une extension des ACM disposant des mécaniques propres aux automates temporisées. Cette version est appelée les Automates temporisées à couches mémoire (**ATCM**) et dispose d'un ensemble d'horloges permettant de mesurer le temps et ses transitions sont annotées par des contraintes sur les valeurs des horloges. Dans cette section est présentée la preuve d'équivalence entre les ETCM et les ATCM.

4.4.1 La définition formelle

Définition 4.4.1 *Un automate temporisé à couches mémoire est un n-uplet de la forme :*

$$A_T = (Q, q_0, F, \Delta_T, V, L, M_0, C)$$

Comme dans les ACM Q est l'ensemble fini de places, $q_0 \in Q$ est la place initiale, $F \subseteq Q$ est l'ensemble de places finales, V et L sont les ensembles finis de

variables et de couches, et M_0 la valeur de contexte mémoire initial. Les ATCM contiennent en plus un ensemble fini C d'horloges utilisées pour mesurer le temps lors de la reconnaissance d'un mot temporisé. Enfin, Δ_T est l'ensemble fini de transitions, dont la définition étend celle des ACM avec l'ajout des contrainte de temps et des réinitialisations d'horloges. Les transitions de Δ_T sont des n-uplets de la forme :

$$(q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q')$$

où $q, q' \in Q$ sont sont respectivement les places d'origine et de destination de la transition. Comme pour les ACM, $\nu, \overline{\nu} \subseteq V \times L$ sont les ensembles de variables modifiables et de variables réinitialisées dans par la transition, et $\alpha : L \to V \cup \{\sharp\}$ est la fonction indiquant les variables consultées par la transition. Comme pour les ACM, les ε-transitions sont les transitions telle que $\alpha = \varepsilon$, c'est-à-dire les transitions ne consultant aucune variable. Les nouveaux éléments ajoutés sont la contrainte γ sur les valeurs des horloges et l'ensemble $\rho \subseteq C$ des horloges réinitialisées par la transition. Nous notons Γ l'ensemble des contraintes sur les horloges, et elles sont définies selon la grammaire suivante :

Définition 4.4.2 (Contraintes de temps)

$$\Gamma := \mathsf{True} \mid c \sim n \mid \Gamma \wedge \Gamma$$

où $c \in C$ est une horloge, $n \in \mathbb{Q}_+$ et $\sim \in \{<, >, \leq, \geq\}$.

On peut remarquer que cette grammaire ne contient pas de *contrainte diagonale*, c'est-à-dire des contraintes de la forme $c_1 - c_2 \sim c$. Dans les automates temporisés, les contraintes diagonales n'accroissent pas l'expressivité du modèle [2]. Nous faisons la conjecture que dans les ATCM les contraintes diagonales n'accroissent pas l'expressivité du modèle.

Pour alléger les différentes contraintes de temps qui seront exprimées dans la suite du chapitre, nous définissons quelques notations :

$$\begin{aligned} c = x &\stackrel{\mathsf{def}}{=} c \leq x \wedge c \geq x \\ c \in I &\stackrel{\mathsf{def}}{=} \mathsf{min}(I) \prec c \prec \mathsf{max}(I) \end{aligned}$$

où $I \subseteq [0, \infty[$ est un intervalle et les $\prec \in \{<, \leq\}$ sont choisies de manière cohérente en fonction des inclusions des bornes de l'intervalle.

Durant de la reconnaissance, les états d'un ATCM sont des triplets (q, M, v) contenant la place $q \in Q$ atteinte lors de la reconnaissance, le contexte mémoire $M : V \times L \to 2^{\mathcal{U}}$ indiquant quelles lettres sont associées aux différentes variables, et $v : C \times \mathbb{Q}_+$ est la valuation des horloges. La valuation des horloges représente

les valeurs des horloges à un instant donné et reprend la Définition 3.3.3. L'état d'origine de l'automate est $(q_0, M_0, \mathbf{0})$ où $\mathbf{0}$ est la valuation initiale des horloges telle que chaque horloge est initialisée à la valeur 0.

D'un état à l'autre les valuations d'horloges peuvent soit être incrémentées soit être en partie réinitialisées. Étant donné une valuation v et un délai x, on note $v + x$ l'incrémentation des horloges de v par x : $\forall c \in C, v + x(c) = v(c) + x$. De même, étant donné un ensembles d'horloges à réinitialiser ρ, on note par $v[\rho \to 0]$ la valuation telle que $\forall c \in \rho, v[\rho \to 0](c) = 0$ et $\forall c \notin \rho, v[\rho \to 0](c) = v(c)$

Pour rappel, les mots temporisées sont des séquences composées de lettres de l'alphabet infini $\mathcal{U}$ et de délais de $\mathbb{Q}_+$. Lors de la lecture d'un délai $x \in \mathbb{Q}_+$ les valeurs des horloges sont incrémentées et l'état est modifié sans qu'il ne soit nécessaire de franchir une transition, ce que l'on note $(q, M, v) \xrightarrow{x} (q, M, v + x)$. Lors de la lecture d'une lettre $e \in \mathcal{U}$ l'automate franchit une transition $\delta \in \Delta_T$, qui n'est pas une ε-transition. Il est possible d'aller de l'état (q, M, v) à l'état (q', M', v') lors de la lecture d'une lettre $e \in \mathcal{U}$ si il existe une transition $\delta = (q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q') \in \Delta_E$ franchissable, c'est-à-dire qu'elle vérifie :

- la lecture de e vérifie les contraintes liées au contexte mémoire, ce qui se traduit à l'aide de σ (Définition 4.1.3), par $M' = \sigma(\delta, M, e) \neq \perp$,
- la valuation v de l'état d'origine doit vérifier les contraintes γ de la transition, ce qu'on note par $v \in \gamma$,
- les horloges de ρ doivent être réinitialisées dans l'état d'arrivée : $v' = v[\rho \to 0]$.

4.4.2 L'équivalence avec les expressions temporisées à couches mémoire

L'ajout des caractéristiques des automates temporisés à celles des automates à couches mémoire se fait de manière relativement orthogonale. En effet, les horloges mesurent le temps et de nouvelles annotations permettent de contraindre les transitions par rapport à la valeur des horloges et de les réinitialiser. Aucun de ces nouveaux éléments ne contraint les lettres des mots reconnus, mais seulement l'instant où elles peuvent être reconnues. Cette section présente l'équivalence entre les ETCM et les ATCM en se basant sur la preuve de la Section 4.3.

Théorème 4.4.1 *Pour tout langage $\mathcal{L}$, il existe une ETCM e et un contexte mémoire M tel que $\mathbb{L}_M(e) = \mathcal{L}$ si et seulement si il existe un ATCM tel que $\mathbb{L}(A) = \mathcal{L}$.*

La preuve de ce théorème est décomposée en deux parties, le premier sens de la preuve (Section 4.4.2) indique que pour tout ETCM on peut construire un ATCM équivalent, et le second sens (Section 4.4.2) présente que pour tout ATCM on peut générer l'ETCM représentant le même langage.

ETCM vers ATCM

Pour construire l'ATCM reconnaissant le même langage qu'une ETCM e donnée, nous étendons la méthode de construction inductive de la Section 4.3.1. Les opérateurs permettant d'exprimer les contraintes de temps sont l'opérateur $set \overset{c}{\triangledown} e$ et l'opérateur $check\ e \overset{c}{\ddagger}_I$. Leur traduction en automate est assez simple et se fonde sur les constructions des opérateurs similaires de [25].

La construction de l'**opérateur *check*** $e \overset{c}{\ddagger}_I$ consiste à ajouter une contrainte temporelle $c \in I$ sur les transitions finales de l'automate généré pour e. Ainsi, étant donné l'automate $(Q, q_0, F, \Delta, V, L, M_0, C)$ reconnaissant le langage exprimé par e, on construit l'automate $A_{\ddagger}$ à partir de l'expression $e \overset{c}{\ddagger}_I$. tel que :

$$A_{\ddagger} = (Q \cup \{q_f\}, q_0, \{q_f\}, \Delta \cup \Delta_{\ddagger}, V, L, M_0, C \cup \{c\})$$

où $\Delta_{\ddagger} = \{(q, \nu, \alpha, \overline{\nu}, \gamma \wedge c \in I, \rho, q_f) \mid (q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q') \in \Delta, q' \in F\}$. Cette construction consiste à créer une nouvelle place q_f qui devient l'unique place finale de $A_{\ddagger}$ et de créer une copie de chaque transition finale dont la destination sera q_f et qui vérifie que la valeur de c fait partie de l'intervalle I.

La sémantique de cet opérateur précise que la vérification de la contrainte de temps a lieu à l'instant de la lecture de la dernière lettre des mots représentés par e. Or, la constructions de l'itération $*$ (Propriété 4.2.4) crée des ε-transitions terminales qui pourraient être franchies à n'importe quel instant. Pour s'assurer qu'une telle transition soit franchie au même instant que la lecture d'une lettre, nous ajoutons une contrainte de temps à cette ε-transition.

L'horloge spéciale c_r est introduite, elle sert uniquement à mesurer le temps depuis la lecture d'une lettre afin de vérifier si la contrainte sur l'ε-transition est satisfaite. Elle est réinitialisée lors de la lecture d'une lettre. Ce sont les expressions terminales qui représentent les lettres composant les mots du langage. Les constructions correspondant aux expressions terminales sont donc modifiées afin que l'horloge c_r y soit réinitialisée. Nous présentons cette modification sur les expressions terminales de la forme X^l, mais la même modification est effectuée sur les expressions des constantes et des lettres fraîches $\sharp X^l$. En reprenant les constructions présentées dans la Section 4.3.1, étant donné une expression de la forme X^l définie sur les ensembles de variables V, de couches L et sur l'alphabet fini Σ, on construit l'automate :

$$A_{X^l} = (\{q_0, q_f\}, q_0, \{q_f\}, \{(q_0, \emptyset, \alpha, \emptyset, \mathsf{True}, \{c_r\}, q_f)\}, V_\Sigma, L, M_e \cup M_\Sigma, \{c_r\})$$

où $\alpha(l) = X$ et $\forall k \in L \backslash \{l\}, \alpha(k) = \sharp$ et M_e est le contexte mémoire initial de l'expression.

La construction de l'itération définie dans la preuve de la propriété 4.2.4 est aussi modifiée afin d'ajouter la contrainte de temps. La seule modification de cette

construction par rapport à la Section 4.3.1 est l'ε-transition de Δ_ε de destination q_ε :

$$\Delta_\varepsilon = \{((q'_0, \emptyset), \emptyset, \varepsilon, \emptyset, \mathsf{True}, \emptyset, (q_0, \emptyset)), ((q'_0, \emptyset), \emptyset, \varepsilon, \emptyset, c_r = 0, \emptyset, (q_\varepsilon, \emptyset))\}$$

L'**opérateur *set*** $\overset{c}{\triangledown}\ e$ réinitialise l'horloge c au début de l'expression e. Ainsi, étant donné l'automate $(Q, q_0, F, \Delta, V, L, M_0, C)$ reconnaissant le langage exprimé par e, on construit l'automate $A_\triangledown$ à partir de l'expression $\overset{c}{\triangledown}\ e$ tel que :

$$A_\triangledown = (Q \cup \{q'_0\}, q'_0, F, \Delta \cup \{(q'_0, \emptyset, \varepsilon, \emptyset, c_r = 0, \rho \cup \{c\}, q_0)\}, V, L, M_0, C \cup \{c, c_r\})$$

La construction consiste simplement à ajouter une ε-transition au début de l'automate reconnaissant le langage de e. Pour s'assurer que la réinitialisation ait lieu au début du mot, avant la lecture de n'importe quel délai, on contraint le franchissement de la transition avec la même horloge c_r.

ATCM vers ETCM

Pour prouver que tout langage reconnu par un ATCM est représentable par une ETCM, nous adaptons l'algorithme de la Section 4.3.2 pour qu'il prenne en compte les annotations relatives aux horloges. Nous définissons donc les AETCM, une extension des AECM dont les transitions sont annotées par des ETCM. Leurs dynamique est très similaire à celle des AECM, avec la particularité de ne pouvoir reconnaître que des TES en forme canonique. Leur dynamique est formellement définie comme suit :

Définition 4.4.3 (La dynamique des AETCM) *Étant donné une AETCM définie sur l'ensemble de transitions Δ, il est possible d'aller d'un état (q, M, v) à un état (q', M', v') lors de la lecture d'une TES en forme canonique w s'il existe $(q, e, q') \in \Delta$ tel que $w \in \mathbb{L}^{M',v'}_{M,v}(e)$.*

La seule modification effectuée sur l'algorithme de la section 4.3.2 est la traduction des annotations des transitions de l'automate en ETCM correspondante. Les étapes de simplification et de suppression de place restent inchangées. Étant donné une transition $(q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q')$ d'un ATCM, elle sera traduite en une transition (q, e, q') dans l'AETCM où :

$$e = C'\overline{\nu}!\ \overset{c_1}{\ddagger}_{I_1} \ldots \overset{c_n}{\ddagger}_{I_n} \cdot \overset{\rho}{\triangledown}\ \epsilon$$

où le C' est la conjonction des expressions terminales des variables utilisées dans α construite telle que dans la Section 4.3.2.

Les $\overset{I_i}{\ddagger}_{c_i}$ représentent les contraintes de temps de la transition, où γ est de la forme $\bigwedge_{i=1}^{n} c_i \sim x_i$ avec $\sim \in \{<, \leq, >, \geq\}$ et I_i est l'intervalle des valeurs de c_i satisfaisant $c_i \sim x_i$. Par exemple, si pour un i donné $c_i \leq 7$ alors $I_i = [0, 7]$ ou encore si pour un j donné $c_j > 2$ alors $I_j = [2, \infty[$. L'ordre des $\ddagger$ n'a pas d'importance car la sémantique de l'opérateur (Section 3.2) montre bien que toutes les contraintes sont vérifiées lors de la lecture de la dernière lettre du mot. Bien entendu, si $\gamma = \mathsf{True}$ alors l'expression e ne contient aucun $\ddagger$.

Le $\overset{\rho}{\nabla}$ représente les réinitialisations des horloges effectuées par la transition. L'opérateur ∇ s'applique en prefixe d'une expression, alors que dans les ATCM les réinitialisations sont effectuées après la vérification des contraintes de temps. Ainsi, l'opérateur est appliqué sur l'expression ϵ dont la seule lettre est *masquée* à la fin de l'algorithme de traduction de l'ATCM en ETCM.

4.4.3 Les opérateurs dédiés

Dans la Section 3.4.4 sont présentés l'opérateur de mélange et l'expression terminale du lien. Seules leurs sémantiques y ont été décrites et ici sont présentées les constructions d'ATCM permettant de représenter les langages correspondant. Cela permettra entre autre de prouver la Propriété 3.4.1 de clôture des ATCM pour le mélange de langages, et donc par le Théorème 4.4.1 la clôture des langage représentés par des ETCM.

Le mélange

Dans la Section 3.4.4 est présenté le mélange de langage temporisés. Le mélange classique, sans prendre en compte les liens, est une construction du produit non synchronisé d'automates, similaire à la composition parallèle d'automates temporisés [2]. Étant donnés les ATCM $A_1 = (Q_1, q_{0_1}, F_1, \Delta_1, V_1, L_1, M_1, C_1)$ et $A_2 = (Q_2, q_{0_2}, F_2, \Delta_2, V_2, L_2, M_2, C_2)$, on construit l'automate $A_{1\otimes 2}$ reconnaissant le mélange des langages reconnus par les automates A_1 et A_2. On suppose sans perte de généralité que les ensembles de couches et d'horloges des deux automates sont disjoints (un renommage pouvant être effectué).

$$A_{1\otimes 2} = (Q_1 \times Q_2, (q_{0_1}, q_{0_2}), F_1 \times F_2, \Delta_\otimes, V_1 \cup V_2, L_1 \cup L_2, M_1 \cup M_2, C_1 \cup C_2)$$

où

$$\begin{aligned}\Delta_\otimes = \ & \{((q_1, q_2), \nu, \alpha, \overline{\nu}, \gamma, \rho, (q_1', q_2)) \mid (q_1, \nu, \alpha, \overline{\nu}, \gamma, \rho, q_1') \in \Delta_1\} \\ & \cup \{((q_1, q_2), \nu, \alpha, \overline{\nu}, \gamma, \rho, (q_1, q_2')) \mid (q_2, \nu, \alpha, \overline{\nu}, \gamma, \rho, q_2') \in \Delta_2\}\end{aligned}$$

La construction du mélange est un produit d'automates, ainsi chaque place de $A_{1\otimes 2}$ est un couple (q_1, q_2) où $q_1 \in Q_1$ et $q_2 \in Q_2$. La place initiale est la

paire (q_{0_1}, q_{0_2}) composée des places finales des deux automates, et de même, les places finales sont toutes les paires dont les places sont finales dans leurs automates respectifs. L'automate est défini sur l'union des variables, couches et horloges de A_1 et A_2. Enfin, l'ensemble de transitions $\Delta_\otimes$,

Les liens

Pour rappel, le lien $\rightsquigarrow$ est un opérateur spécifique aux ETCM permettant de limiter le mélange d'expressions terminales successives, tout en spécifiant une contrainte de temps telle que les événements liés apparaissent au même instant. Il est présenté dans la Section 3.4.4 que deux expressions terminales liées $t_1 \rightsquigarrow t_2$ représentent le même langage qu'une expression $\langle t_1 \cdot t_2 \rangle_{[0,0]}$. C'est effectivement le cas dans la construction de l'automate correspondant à cette expression. Cependant, la particularité du lien est son interaction avec l'opérateur de mélange. En effet, c'est la construction du mélange qui est altérée afin de ne pas séparer les lettres qui sont liées ensembles.

Étant donnés les automates $A_1 = (Q_1, q_{0_1}, \{q_{f_1}\}, \Delta_1, V_\Sigma, L, M_e \cup M_\Sigma, \{c_r\})$ et $A_2 = (Q_2, q_{0_2}, \{q_{f_2}\}, \Delta_2, V_\Sigma, L, M_e \cup M_\Sigma, \{c_r\})$ construits à partir des expressions t_1 et t_2 respectivement, on construit l'automate :

$$A_{1\rightsquigarrow 2} = ((Q_1 \backslash \{q_{f_1}\}) \cup (Q_2 \backslash \{q_{0_2}\}) \cup \{q^\circ\}, q_{0_1}, \{q_{f_2}\}), \Delta_\rightsquigarrow, V_\Sigma, L, M_e \cup M_\Sigma, \{c_r\})$$

$$\begin{aligned} \Delta_\rightsquigarrow = \quad & (\Delta_1 \backslash \{(q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q_{f_1})\}) \cup (\Delta_2 \backslash \{(q_{0_2}, \nu', \alpha', \overline{\nu}', \gamma', \rho', q')\}) \\ & \cup \{(q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q^\circ) \mid (q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q_{f_1}) \in \Delta_1\} \\ & \cup \{(q^\circ, \nu', \alpha', \overline{\nu}', \gamma' \wedge c_r = 0, \rho', q) \mid (q_{0_2}, \nu', \alpha', \overline{\nu}', \gamma', \rho', q) \in \Delta_2\} \end{aligned}$$

Pour comprendre la construction, il faut savoir que les automates construits par chaque expression terminale contient une unique place initiale et une unique place finale et un unique chemin de transitions les reliant. De plus, aucune transition de ces automates n'est une ε-transition, donc l'horloge spéciale c_r y est réinitialisées dans chaque transition. L'idée de la construction du lien est tout d'abord de fusionner la place de destination de A_1 avec la place la place de départ de A_2 en q°. Ainsi la transition qui a pour destination q_{f_1} a sa destination changée pour q° et de même la transition sortante de q_{0_2} a son origine modifiée pour q°. De plus la transition sortante de q° est contrainte à être simultanée à sa transition entrante avec $c_r = 0$. Il n'est pas nécessaire d'ajouter cette contrainte sur toutes les transitions de Δ_1 et Δ_2 car si il y en a plus qu'une alors cela signifie que ces automates ont été construits avec des $\rightsquigarrow$ qui ont déjà ajouté la contrainte.

Le symbole $\circ$ indique que si cet automate est mélangé, alors les seules transitions sortantes de cette place doivent être celles de cet automate. On note par $Q^\circ \subset Q$ l'ensemble des places annotées par $\circ$. La modification de la construction

du mélange est une redéfinition de son ensemble de transitions :

$$\begin{aligned}\Delta_{\otimes} = \quad & \{((q_1, q_2), \nu, \alpha, \overline{\nu}, \gamma, \rho, (q_1', q_2)) \mid (q_1, \nu, \alpha, \overline{\nu}, \gamma, \rho, q_1') \in \Delta_1, q_2 \notin Q_2^{\circ}\} \\ & \cup \{((q_1, q_2), \nu, \alpha, \overline{\nu}, \gamma, \rho, (q_1, q_2')) \mid (q_2, \nu, \alpha, \overline{\nu}, \gamma, \rho, q_2') \in \Delta_2, q_1 \notin Q_1^{\circ}\}\end{aligned}$$

Cette construction assure que, lors du mélange d'un automate A_1 avec un automate A_2, les seules transitions sortant d'une place (q_1°, q_2) sont les transitions de Δ_1, et respectivement Δ_2 pour une place (q_1, q_2°). Pour que cette construction fonctionne, il faut s'assurer que le symbole $\circ$ soit préservé par les autres constructions modifiant les identifiants des places dans Q, c'est-à-dire, les constructions correspondant aux opérateurs & et $*$. La construction de & est le produit synchrone des automates correspondant à chaque sous-expression, et cette construction consiste à simuler le fonctionnement des deux automates en parallèle. Ainsi, les places et transitions de l'automate cumulent toutes les contraintes de chacune des places et transitions des deux automates synchronisés. Donc, étant donné un couple de places (q_1, q_2) de l'automate $A_{1\&2}$, si $q_1 \in Q_1^{\circ}$ ou $q_2 \in Q_2^{\circ}$ alors $(q_1, q_2) \in (Q_1 \times Q_2)^{\circ}$. De même, la construction correspondante $A^*/\bar{l}/$ ajoute des annotations S aux places pour simuler le fonctionnement de l'automate A. Ainsi, étant donné une place (q, S) de A_*, si q est annotée par $\circ$, $q \in Q^{\circ}$, alors toutes les paires contenant la place q le sont aussi, $(q, S) \in (Q \times 2^{V \times L})^{\circ}$.

Pour résumer, la contribution principale de ce chapitre est la définition du modèle des automates à couches mémoire et de son équivalent temporisé. Les ATCM offrent un formalisme de reconnaissance pour tous les langages exprimables sous forme d'ETCM, ce qui est démontré avec un théorème similaire au théorème de Kleene. À l'aide de cette propriété il devient alors possible de positionner la classe des langages sur alphabet infini exprimés par les ECM par rapport à ceux des différents modèles d'automates de la littérature. Ainsi, comme les ACM sont plus sont expressifs que les GRA et les FMA alors les ECM sont strictement plus expressifs que leurs langages de spécifications respectifs : les IARE et REM [56, 63]. Ce positionnement offre aussi des informations quand à la décidabilité de certains problèmes ainsi qu'une estimation de leurs classes de complexité. Le problème de la reconnaissance qui nous intéresse particulièrement est décidable dans les ACM, même si NP-difficile.

Pour exploiter un tel formalisme de reconnaissance pour détecter des motifs dans les systèmes complexes, il est nécessaire de concevoir un algorithme de reconnaissance. La description de la dynamique du modèle ne permet pas de déterminer un algorithme trivial résolvant ce problème. De plus, nous souhaitons l'implémenter dans un outil afin d'appliquer nos travaux sur des jeux de données réels, ainsi il est nécessaire de porter attention aux performances de l'algorithme.

Chapitre 5

L'algorithmique de reconnaissance

La problématique principale développée dans cette thèse est celle de la vérification dynamique de propriétés sur des systèmes complexes. Dans les chapitres précédents ont été présentés le langage de spécification permettant d'exprimer ces propriétés ainsi que le modèle d'automate sous-jacent servant de formalisme de reconnaissance. Bien que la dynamique de ce modèle définit l'ensemble des mots reconnaissables pour tout automate, il reste encore à présenter l'algorithme effectif permettant de vérifier si un mot donné appartient à cet ensemble. Ce chapitre est consacré à la présentation de l'algorithme de reconnaissance et des expérimentations effectuées avec son implémentation dans un prototype.

Dans le cadre de la vérification dynamique de propriété et l'analyse de flots de liens, nous nous intéressons aux algorithmes dit à la volée (*online* en anglais). L'objectif est de superviser un système au cours de son exécution, plutôt qu'une analyse a posteriori, afin de détecter à quelle étape la propriété recherchée est satisfaite. Un tel traitement peut être dans l'optique d'altérer ou d'interrompre son exécution une fois cette propriété détectée.

Le problème de la reconnaissance est simple lorsqu'il s'agit d'automate déterministe. Cependant, le modèle des ATCM est une combinaison des automates temporisés et des ACM, deux modèles qui ne sont pas déterminisables [2]. De plus, les classes d'automates temporisés déterministes et d'ACM déterministes sont strictement moins expressives que leurs équivalents non-déterministes. L'algorithme de reconnaissance que l'on a développé manipule donc des automates non-déterministes. C'est un problème qui est plus complexe que dans le cas déterministe car il est nécessaire de parcourir plusieurs chemins de transitions lors de la lecture d'un mot pour déterminer s'il appartient au langage de l'automate.

Ce chapitre débute avec la présentation globale de l'algorithme de reconnaissance. Dans la seconde section sont détaillés, à un niveau plus technique, les calculs permettant de déterminer les valeurs des horloges lors de la reconnaissance. La section suivante présente une analyse de la taille des automates générés afin

de proposer des solutions pour réduire la complexité en espace de l'algorithme. Enfin, le chapitre est conclu par une présentation des quelques expérimentations que nous avons effectuées à l'aide de notre prototype.

5.1 L'algorithme de reconnaissance

La dynamique des ATCM indique qu'un mot est reconnaissable s'il existe un chemin de transitions franchissable lors de la lecture du mot, allant de l'état initial de l'automate à un état final. La difficulté du problème de la reconnaissance dans un automate non-déterministe est qu'il existe, pour un même mot, plusieurs chemins franchissables. Nous avons donc développé un algorithme de reconnaissance à la volé calculant, à partir d'un automate A donné et d'un mot w, tous les états accessibles dans A lors de la lecture de w.

Les états sont représentés, dans la dynamique du modèle, par des triplets (q, M, v) composés d'une place q et des contexte de mémoire M et d'horloges v. Les états accessibles lors de la lecture d'une lettre sont déterminés par les transitions franchissables depuis l'état courant. Il peut exister plusieurs transitions sortantes franchissables lors de la lecture d'une même lettre, menant à des états différents. Toutefois, pour une transition (non-ε) donnée, si la lecture d'une lettre permet de la franchir, alors il existe un unique état acessible qui est défini par les annotations présentes sur la transition.

Dans le cas des ε-transitions, elles peuvent être franchies à tout moment tant que les contraintes de temps γ les annotant sont satisfaites par les valuations d'horloges v de l'état courant. Les ε-transitions sont franchissables sans qu'il soit nécessaire de lire une lettre, leur comportement est totalement non-déterministe car elles permettent l'accès à plusieurs états simultanément. Pour rappel, les mots sont représentés sous forme de séquences d'événements temporisés, ainsi lors de la lecture à la volée du mot, des délais vont être lus. Lors de la lecture d'un délai, il est possible de franchir une ε-transition durant l'incrémentation des horloges. Si la transition réinitialise une ou plusieurs horloges, alors plusieurs valuations d'horloges sont accessibles dans l'état d'arrivée à l'issue de l'incrémentation.

Exemple 5.1.1 *Soit une ε-transition $q \xrightarrow[2\leq c, c:=0]{\varepsilon} q'$ en supposant que l'état courant est de la forme $(q, M, c \to 0)$. Cette transition n'est pas franchissable depuis l'état courant car la valeur de l'horloge c est inférieure à 2. Lors de la lecture du délai 5, la valeur de c va alors être incrémentée de 5 et il sera possible de franchir la transition. Un scénario possible est d'incrémenter la valeur de l'horloge de 2, modifiant l'état en $(q, M, c \to 2)$ à partir duquel il est possible de franchir l'ε-transition, comme l'horloge est réinitialisée l'état atteint est $(q', M, c \to 0)$, ensuite il est nécessaire d'incrémenter l'horloge de 3 et l'état devient $(q', M, c \to 3)$ à la fin*

de l'incrémentation du délai. Cependant, il est possible de franchir la transition à tout moment durant l'incrémentation de l'horloge. Il est ainsi possible d'incrémenter c par une portion du délai supérieure à 2 avant de franchir la transition. La valeur de c pourra varier de 3 à 0 en fonction de la portion du délai incrémentée dans la place q.

Pour les automates temporisés, ce phénomène est bien connu dans la littérature et il est courant de regrouper les valuations d'horloges sous forme de *zone temporelle* [2]. Ce modèle permet de représenter l'ensemble des valuations d'horloges respectant les mêmes contraintes temporelles.

Définition 5.1.1 (Zone temporelle) *Une zone temporelle définie sur un ensemble fini d'horloges C est représenté par la conjonction d'un ensemble fini de contraintes temporelles. La zone représente l'ensemble, parfois infini, des valuations satisfaisant ces contraintes. Les contraintes temporelles peuvent être :*

— *des contraintes diagonales, de la forme* $c_i - c_j \sim x$

— *des contraintes d'horloge, de la forme* $c \sim x$,

où $c, c_i, c_j \in C$, $\sim \in \{\leq, <, \geq, >\}$ *et* $x \in \mathbb{Q}$

En utilisant les zones d'horloges, il est possible de représenter simultanément un ensembles d'états des ATCM dont les valuation d'horloges font toutes partie de la même zone. Ainsi, nous notons par $(q, M, \mathcal{Z})$ l'ensemble des états $\{(q, M, v) \mid v \in Z\}$ dont la valuation v fait partie de la zone $\mathcal{Z}$, c'est-à-dire qu'elle satisfait toutes les contraintes définissant la zone. De plus, les zones sont closes pour les opérations d'intersection et d'incrémentation. Le prédicat $\mathsf{empty}(\mathcal{Z})$ indique si la zone $\mathcal{Z}$ est vide ou non, c'est-à-dire si elle contient au moins une valuation d'horloges.

Le pseudo code

L'algorithme 2 prend en paramètre un ATCM A, effectue la reconnaissance à la volée d'une séquence d'événements temporisés w et détermine si un état final est atteint. Au fur et à mesure de la lecture du mot sont calculés l'ensemble S de tous les états accessibles. Initialement, S contient uniquement l'état initial, ainsi que l'ensemble des états qu'il est possible d'atteindre via des ε-transitions.

La fonction $\sigma_\varepsilon(S, x)$ calcule l'ε-fermeture temporisée, c'est-à-dire l'ensemble des états atteint via tout chemin d'ε-transitions depuis les états de S lors de la lecture d'un délai x. Les ε-transitions pouvant être franchies à tout moment, il est nécessaire de vérifier si de nouveaux états sont accessibles à travers elles avant et après chaque lecture. C'est pourquoi σ_ε est appelé initialement et suite à la lecture de chaque lettre et délai du mot.

Lors de la lecture d'un délai, seules des ε-transitions peuvent être franchies. Peu de calculs sont à effectuer sur la mémoire car les ε-transitions ne consultent aucune

Algorithme 2 Algorithme de reconnaissance

Entrée: Un ATCM $A = (Q, q_0, F, \Delta, V, L, M_0, C)$ et un TES $w = w_1 w_2 \dots w_n$
Sortie: Si $w \in \mathbb{L}(A)$ alors $\exists q \in F, (q, M, \mathcal{Z}) \in S$.

$S \leftarrow \{(q_0, M_0, \mathbf{0}\}$
$S \leftarrow \sigma_\varepsilon(S, 0)$
pour i allant de 1 à n **faire**
  si $w_i \in \mathbb{Q}$ **alors**
    $S \leftarrow \sigma_\varepsilon(S, w_i)$
  sinon
    $S' \leftarrow \emptyset$
    pour $\delta = (q, \nu, \alpha, \overline{\nu}, \gamma, \rho, q') \in \Delta$ où $\alpha \neq \varepsilon$ **faire**
      pour $(q, M, \mathcal{Z}) \in S$ **faire**
        si $\neg\mathsf{empty}(\mathcal{Z} \cap \gamma) \wedge \sigma(\delta, M, w_i) \neq \perp$ **alors**
          $S' \leftarrow S' \cup (q', \sigma(\delta, M, w_i), (\mathcal{Z} \cap \gamma)[\rho \leftarrow 0])$
        fin si
      fin pour
    fin pour
    $S \leftarrow \sigma_\varepsilon(S', 0)$
  fin si
fin pour

variable et peuvent seulement les réinitialiser. Cependant, comme les horloges sont incrémentées, une arithmétique complexe a lieu sur les zones afin de déterminer les zones temporelles atteintes dans chaque état.

Lors de la lecture d'une lettre, il faut calculer l'ensemble des états accessibles par le franchissement d'une transition acceptant la lettre lue. La fonction σ, de la définition 4.1.3, indique si une lettre permet de franchir une transition et la valeur de son contexte mémoire dans l'état atteint. Il est tout de même nécessaire de satisfaire les contraintes temporelles afin de franchir la transition. Pour ce faire on considère les contraintes γ de la transition comme une zone temporelle et en calculer l'intersection avec celle de l'état d'origine de la transition. Si le résultat de l'intersection est vide alors aucune valuation de l'état n'est capable de satisfaire ces contraintes. Une fois ces transitions franchies, il faut de nouveau calculer les états accessibles via des ε-transitions avec la fonction σ_ε.

La principale difficulté de cet algorithme est le calcul de l'ε-fermeture effectué par la fonction σ_ε. En effet, bien que le fonctionnement de σ est relativement complexe, chaque état atteint après la lecture d'une lettre est obtenu par le franchissement d'une seule transition. Cependant, le calcul de la fermeture nécessite de calculer l'ensemble des états accessibles par tout chemin d'ε-transitions.

Définition 5.1.2 (L'ε-fermeture)

$$\sigma_\varepsilon(S,x) = \left\{ (q, M', \mathcal{Z}') \;\middle|\; \begin{array}{l} \exists q_1 \underbrace{\xrightarrow[\gamma_1,\rho_1]{\nu_1,\varepsilon,\overline{\nu_1}}}_{\delta_1} q_2 \underbrace{\xrightarrow[\gamma_2,\rho_2]{\nu_2,\varepsilon,\overline{\nu_2}}}_{\delta_2} \cdots q_n \underbrace{\xrightarrow[\gamma_n,\rho_n]{\nu_n,\varepsilon,\overline{\nu_n}}}_{\delta_n} q \in \Delta^*, \\ \exists (q_1, M, \mathcal{Z}) \in S,\ \mathsf{walk}(\mathcal{Z}, [\delta_1, \delta_2, \ldots, \delta_n], x) = \mathcal{Z}' \\ \wedge \neg \mathsf{empty}(\mathcal{Z}') \wedge M' = M[\bigcup_{i=1}^{n} \overline{\nu}_i \leftarrow \emptyset] \end{array} \right\}$$

À partir d'un ensemble d'états S et d'un délai x, l'ε-fermeture est l'ensemble des états accessibles depuis un état de S via un chemin d'ε-transitions en incrémentant les horloges de x à travers le chemin. Étant donné un chemin d'ε-transitions (potentiellement vide) et un état de S, la fonction walk calcule la zone temporelle $\mathbf{D}'$ des horloges dans l'état atteint par le chemin. Si cette zone n'est pas vide, alors le chemin est franchissable. Le contexte mémoire peut évoluer à travers le chemin car les variables de $\overline{\nu}$ sont réinitialisées. Cependant, la mémoire ne contraint pas le franchissement des ε-transitions. La fonction walk est basée sur un calcul de zones temporelles plutôt complexe présenté dans la Section 5.2.

Propriété 5.1.1 *Étant donné l'ensemble d'états S et le délai x, le résultat de la fonction $\sigma_\varepsilon(S,x)$ est un ensemble fini d'états.*

Preuve : Dans un premier temps, il est facile de prouver qu'il existe un nombre fini de contextes mémoire accessibles dans une ε-fermeture. La seule opération qu'une ε-transition peut effectuer sur la mémoire est de réinitialiser des variables. Ré-initialiser une variable qui l'a déjà été ne modifie pas sa valeur de nouveau. Ainsi, étant donné un contexte mémoire composé de n variables de la forme X^l, il existe au maximum 2^n contextes mémoire accessibles dans une ε-fermeture à partir d'un état utilisant ce contexte mémoire.

De la même manière, il existe un nombre fini de zones temporelles accessibles dans une ε-fermeture quand la valeur des horloges est bornée [16]. Dans le cas de la reconnaissance, la borne maximale de la valeur des horloges est la durée du mot lu. Ainsi, comme il existe un nombre fini de zones et de contextes mémoire accessibles, alors il en existe un nombre fini de combinaisons correspondant aux états accessibles. □

5.2 Les calculs de zones temporisées

Dans cette section sont présentés les aspects de l'algorithme de reconnaissance liés au temps. Les contraintes de temps et les valeurs des horloges sont représentées sous forme de zones temporelles (Définition 5.1.1). L'approche que nous avons choisie reprend le modèle classique présenté dans [35], où les zones temporelles

sont représentées sous forme de matrices (*difference bound matrices*, **DBM**). À la différence de l'utilisation classique de ce formalisme pour le *model checking*, nous présentons les opérations permettant de l'adapter au problème de la reconnaissance.

5.2.1 *Difference Bound Matrix*

Les zones d'horloges permettent de représenter un ensemble de combinaisons de valeurs possibles pour les horloges, respectant un ensembles de contraintes de temps. D'un point de vue géométrique, les zones sont des polytopes convexes à n dimensions, où n est le nombre d'horloges, et chaque face du polytope représente une contrainte de temps. Les DBM permettent de représenter les zones sous la forme de matrices décrivant les contraintes de temps délimitant la zone. Comme indiqué par la Définition 4.4.2 de la grammaire des contraintes de temps, il existe deux types de contraintes délimitant une zone : des bornes supérieure et inférieure sur les valeurs d'une horloge ou sur la différence de valeur entre deux horloges.

Définition 5.2.1 (Difference Bound Matrix) *Étant donné l'ensemble d'horloges* $\{c_1, c_2, \ldots, c_n\}$ *une DBM est une matrice* $\mathbf{D} = \{d_{ij}\}_{0 \leq i,j \leq n}$ *où chaque* d_{ij} *est une borne temporelle, c'est-à-dire un élément de*

$$(\mathbb{Q} \times \{<, \leq\}) \cup \{(\infty, <), (-\infty, <)\}.$$

Chaque borne temporelle $d_{ij} = (x, \sim)$ *représente une contrainte de la forme* $c_i - c_j \sim x$ *où* $\sim \in \{<, \leq\}$.

La DBM représente la zone temporelle contenant l'ensemble des valuations d'horloges satisfaisant les contraintes exprimées par ses bornes temporelles. Les bornes diagonales de toutes DBM (d_{ii}) ont normalement pour valeur $(0, \leq)$, mais ces bornes étant superflues, il est classique que leurs valeurs soient modifiées pour les besoins de certains calculs. Les DBM disposent d'une horloge spéciale c_0 dont la valeur est fixée à 0 et est utilisée pour représenter les intervalles de définition de chaque horloge. En effet, la valeur $d_{i0} = (x_{i0}, \sim_{i0})$ représente la borne supérieure de l'horloge c_i car $c_i - c_0 \leq x_{i0} \Leftrightarrow c_i \leq x_{i0}$; et de la même manière la valeur $d_{0i} = (x_{0i}, <)$ représente la borne inférieure car $c_0 - c_i \sim_{0i} x_{0i} \Leftrightarrow -x_{0i} > c_i$.

Dans les différents calculs présentés, nous considérons qu'il existe une relation d'ordre totale sur les bornes temporelles telle que : $(x_1, \sim_1) < (x_2, \sim_2)$ si et seulement si $x_1 < x_2 \vee (x_1 = x_2 \wedge \sim_1 = < \wedge \sim_2 = \leq)$. De même, nous dénotons par $\mathsf{min}(d, d')$ le minimum entre les bornes d et d'. La relation d'ordre des bornes s'étend naturellement à une relation d'inclusion sur les DBM. Ainsi, étant donné les DBMs $\mathbf{D} = \{d_{ij}\}_{0 \leq i,j \leq n}$ et $\mathbf{D'} = \{d'_{ij}\}_{0 \leq i,j \leq n}$ définis sur les mêmes ensembles d'horloges, si $\forall 0 \leq i, j \leq n, d_{ij} \leq d'_{ij}$ alors $\mathbf{D}$ est incluse dans $\mathbf{D'}$.

Nous représentons par $v \in \mathbf{D}$ le fait que la valuation d'horloges $v : \{c_1, c_2, \ldots, c_n\} \to \mathbb{Q}$ appartient à la zone temporelle représentée par la matrice $\mathbf{D}$. Formellement, cela signifie qu'étant donné $\mathbf{D} = \{(x_{ij}, \sim_{ij})\}_{0 \leq i,j \leq n}$ on note $v \in \mathbf{D}$ si pour tous $0 \leq i, j \leq n$ la valuation vérifie $v(c_i) - v(c_j) \sim_{ij} x_{ij}$.

Nous reprenons trois opérations classiques des DBM de l'article [35]. Le prédicat $\mathsf{empty} : \mathbf{D} \to \mathbb{B}$ prenant en paramètre une DBM et renvoyant True si il n'existe aucune valuation dans la zone représenté par la DBM. L'intersection $\mathbf{D}_1 \cap \mathbf{D}_2$ de deux DBM, prenant en paramètre deux DBMs et dont le résultat est une DBM représentant la zone qui est à leur intersection. Et la forme canonique notée $[\![\mathbf{D}]\!]$ qui représente la DBM composée des bornes les plus fortes permettant de représenter la même zone que $\mathbf{D}$. Les définitions précises de ces opérateurs sont données dans [35].

Quelques opérateurs plus spécifiques sont nécessaires dans l'algorithme de reconnaissance. L'opérateur de prolongation, noté ext, prend en paramètre deux DBM, $\mathbf{D}^1 = \{d^1_{ij}\}_{0 \leq i,j \leq n}$ et $\mathbf{D}^2 = \{d^2_{ij}\}_{0 \leq i,j \leq n}$, et renvoie la zone contenant toutes les valuations intermédiaires de ces DBM. Étant donné une valuation v, elle est intermédiaire à $\mathbf{D}^1$ et $\mathbf{D}^2$, s'il existe $x_1 \in \mathbb{Q}_+$ tel que $v - x_1 \in \mathbf{D}^1$ et qu'il existe $x_2 \in \mathbb{Q}_+$ tel que $v + x_2 \in \mathbf{D}^2$. Grâce à cet opérateur on obtient la zone contenant toutes les valuations. Une manière d'interpréter cet opérateur est qu'il construit les chemins qu'empruntent les valuations de la zone de $\mathbf{D}^1$ lorsqu'elles sont incrémentées pour devenir des valuations de $\mathbf{D}^2$. Ainsi on peut calculer la zone intermédiaire comme :

$$\mathsf{ext}(\mathbf{D}^1, \mathbf{D}^2) = \left\{ d_{ij} = \left[\begin{array}{l} d^2_{i0} \text{ if } j = 0 \\ d^1_{0j} \text{ if } i = 0 \\ \text{sinon } \mathsf{min}(d^1_{ij}, d^2_{ij}) \end{array} \right\}_{0 \leq i,j \leq n} \right.$$

L'opérateur de réinitialisation assigne la valeur 0 à une horloge et met à jour la DBM afin que les contraintes restent cohérentes par rapport aux autres valeurs. Ainsi, étant donné une DBM $\mathbf{D} = \{d_{ij}\}_{0 \leq i,j \leq n}$ et une horloge c_k, la réinitialisation de c_0 dans $\mathbf{D}$ est notée et calculée ainsi :

$$\mathbf{D}[c_k \leftarrow 0] = \left\{ d'_{ij} = \left[\begin{array}{l} (0, \leq) \text{ if } (i = k \wedge j = 0) \vee (i = 0 \wedge j = k) \\ d_{0j} \text{ if } i = k \wedge j \neq 0 \\ d_{i0} \text{ if } j = k \wedge i \neq 0 \\ \text{otherwise } d_{ij} \end{array} \right\}_{0 \leq i,j \leq n} \right.$$

On dénote par $\mathbf{D}^1[c_1, c_2, \ldots \leftarrow 0]$ la réinitialisation de l'ensemble d'horloges $\{c_1, c_2, \ldots\}$ dans la DBM, ce qui est équivalent à $\mathbf{D}^1[c_1 \leftarrow 0][c_2 \leftarrow 0] \ldots$.

Il est possible d'*incrémenter* une zone temporelle par une durée $x \in \mathbb{Q}$. Étant donné une zone représentée par $\mathbf{D}$, on note par $\mathbf{D} + x$ la zone telle que pour toute

valuation $v \in \mathbf{D}$ il est vrai que $v + x \in \mathbf{D} + x$. Dans les DBM, cela se calcule en incrémentant les bornes des intervalles de valeurs des horloges :

$$\mathbf{D} + x = \left\{ d'_{ij} = \left[\begin{array}{l} d_{i0} + x \text{ if } i > 0, j = 0 \\ d_{0j} - x \text{ if } i = 0, j > 0 \\ \text{otherwise } d_{ij} \end{array} \right. \right\}$$

5.2.2 L'epsilon fermeture

La fonction σ_ε permet de calculer les états accessibles via un chemin d'ε-transitions lors de la lecture d'un délai x (où $x \geq 0$). Le franchissement de ces transitions est limité uniquement aux valuations d'horloges satisfaisant γ. Nous présentons ici les calculs effectués par la fonction walk, utilisée dans σ_ε pour déterminer les zones temporelles accessibles à l'issue du franchissement d'un chemin d'ε-transitions.

La complexité présente dans ce calcul est la gestion du délai x. Dans l'état atteint à l'issue du franchissement du chemin, les horloges doivent avoir été incrémentées par la totalité de x. Cependant, il est possible d'incrémenter les horloges de seulement une partie de la valeur de x dans un état intermédiaire afin de satisfaire une contrainte d'horloge d'une des transitions. Ainsi, au cours du calcul de walk il est nécessaire de calculer les différentes zones temporelles intermédiaires accessibles en fonction des différentes manières d'incrémenter les horloges.

Définition 5.2.2 (Franchir un ε-chemin)

$$\mathsf{walk}(\mathbf{D}, [\delta_1, \delta_2, \ldots, \delta_n], x) = \mathbf{F}^n$$

où $\mathbf{F^n}$, *est obtenue par la procédure itérative suivante :*

$$\left[\begin{array}{ll} (\mathbf{P^0}, \mathbf{W^0}, \mathbf{F^0}) & = (\mathbf{D}, \mathbf{D}, \mathbf{D} + x) \\ (\mathbf{P^i}, \mathbf{W^i}, \mathbf{F^i}) & = \mathsf{delay}(\delta_i, \mathbf{P^{i-1}}, \mathbf{W^{i-1}}, \mathbf{F^{i-1}}), i \in [1, n] \end{array} \right.$$

La fonction walk est basée sur une procédure itérative qui consiste à mettre à jour un ensemble de trois DBMs : $\mathbf{P}$ (passé), $\mathbf{W}$ (présent) and $\mathbf{F}$ (futur). Lors de la lecture d'un délai x, $\mathbf{P}$ représente la zone initiale précédant la lecture et $\mathbf{F}$ représente la zone incrémentée par le délai. La troisième zone, $\mathbf{W}$, représente les valuations intermédiaires qui sont atteintes pour vérifier les contraintes des ε-transitions du chemin.

Le cœur de la fonction walk est la fonction delay qui à partir de ces trois DBM, du contexte mémoire et d'une ε-transition permet de calculer les valeurs de ces DBM dans l'état après le franchissement de la transition. Elle est décrite dans la section suivante.

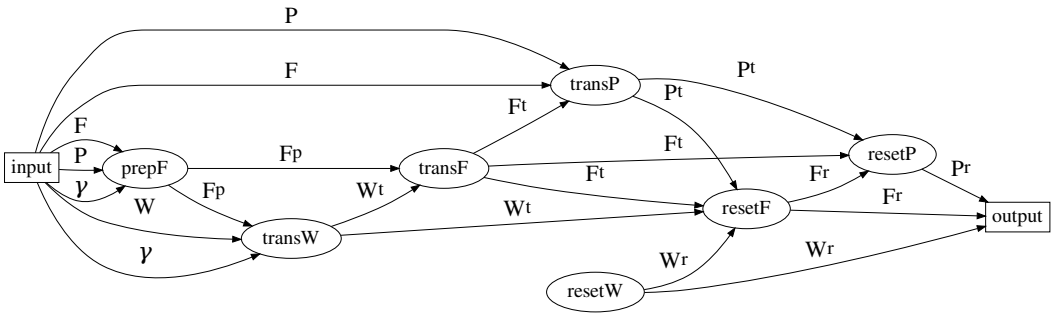


FIGURE 5.1 – Le dataflow pour le calcul de delay.

5.2.3 Le dataflow du calcul du délai

Définition 5.2.3 (Mise à jour des DBM) *Étant donné* $\delta = (q, \nu, \varepsilon, \overline{\nu}, \gamma, \rho, q')$ *et les DBMs* $\mathbf{P}$, $\mathbf{W}$, $\mathbf{F}$, *la fonction* $\mathsf{delay}(\delta, \mathbf{P}, \mathbf{W}, \mathbf{F})$ *est calculable via la procédure schématisée dans la Figure 5.1. Cette fonction est définie seulement si la totalité des calculs de ce dataflow sont possibles.*

La fonction partielle delay permet à partir des trois DBMs et de la transition de calculer la zone contenant l'ensemble des valuations d'horloges satisfaisant la contrainte de la transition ainsi que l'état de ces valuations après les potentiels réinitialisations. Les DBMs finales et initiales sont aussi mis à jour, le tout suivant le dataflow de la Figure 5.1. Depuis la source du dataflow *input*, sur la gauche, jusqu'au résultat, *output* sur la droite, les DBMs $\mathbf{P}, \mathbf{W}, \mathbf{F}$ traversent les nœuds qui vont mettre à jour les valeurs des matrices et donc les zones qu'elles représentent. Certaines fonction du dataflow prennent aussi en argument les contraintes de temps γ afin de vérifier si elles sont satisfaites. Chaque nœud du dataflow représente une fonction mettant à jour l'une des matrices. Lorsque le résultat d'une de ces fonctions est une zone vide, cela signifie que la transition n'est pas franchissable et que delay n'est pas défini pour ces paramètres.

On peut décomposer le calcul en trois étapes principales :

— L'étape de préparation (les nœuds de préfixe `prep`) qui applique les contraintes de γ grossièrement sur les DBM pour réduire les zones.

— L'étape de franchissement de la transition (préfixe `trans`) qui calcule la zone temporelle satisfaisant précisément γ.

— Enfin, l'étape de réinitialisation (préfixe `reset`) qui assigne 0 aux horloges de ρ dans les zones calculées à l'étape précédente. Ce qui correspond aux valuations d'horloges qui résultent du franchissement de la transition.

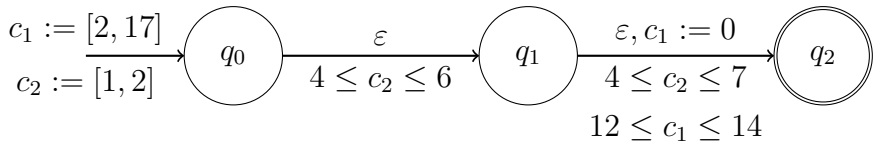


FIGURE 5.2 – Un ATCM composé de deux ε-transitions et deux horloges.

Comme présenté par les flèches dans la Figure 5.1, les dépendances entre les différentes étapes et les nœuds qui les composent ne sont pas triviales. Dans la suite de cette section sont présentées les fonctions correspondant à chacun de ces nœuds. Mais nous présentons tout d'abord un exemple qui servira à illustrer ces différentes définitions.

Exemple 5.2.1 *Pour illustrer les calculs du dataflow, nous calculons le franchissement des deux ε-transitions de l'automate de la Figure 5.2 lors de la lecture d'un délai de 9 unités de temps (ou secondes). Nous considérons que les valeurs initiales des horloges sont non nulles, que l'horloge c_1 à une valeur comprise dans l'intervalle $[2, 17]$ et c_2 à une valeur comprise dans l'intervalle $[1, 2]$, ce qui correspond à la zone temporelle décrite par la DBM :*

$$\mathbf{D} = \begin{pmatrix} (0, \leq) & (-2, \leq) & (-1, \leq) \\ (17, \leq) & (0, \leq) & (16, \leq) \\ (2, \leq) & (0, \leq) & (0, \leq) \end{pmatrix}$$

Les deux contraintes diagonales d_{12} et d_{21} sont calculées via l'opérateur $[\![.]\!]$ donnant la forme canonique.

Les calculs des DBM relatifs au franchissement de la première transition ne sont pas détaillés. Elle permet d'atteindre un état intermédiaire plus représentatif pour expliquer les fonctions du dataflow. Comme le délai lu est 9, les DBMs initiales de la fonction walk *sont :*

$$\mathbf{P^{(0)}} = \mathbf{D};\ \mathbf{W^{(0)}} = \mathbf{D};\ \mathbf{F^{(0)}} = \mathbf{D} + 9.$$

Lors du franchissement de la première transition $\delta_1 = (q_0, \emptyset, \varepsilon, \emptyset, \gamma, \rho, q_1)$, on calcule les DBMs résultant du franchissement avec la fonction delay *:*

$$(\mathbf{P^{(1)}}, \mathbf{W^{(1)}}, \mathbf{F^{(1)}}) = \mathsf{delay}(\delta_1, \mathbf{P^{(0)}}, \mathbf{W^{(0)}}, \mathbf{F^{(0)}}).$$

Sans rentrer dans les détails, les résultats de delay *sont :* $\mathbf{P^{(1)}} = \mathbf{P^{(0)}}$, $\mathbf{F^{(1)}} = \mathbf{F^{(0)}}$ *et :*

$$\mathbf{W^{(1)}} = \begin{pmatrix} (0, \leq) & (-4, \leq) & (-4, \leq) \\ (22, \leq) & (0, \leq) & (16, \leq) \\ (6, \leq) & (0, \leq) & (0, \leq) \end{pmatrix}$$

où $\mathbf{W}^{(1)}$ est la zone temporelle contenant toutes les valuations intermédiaires à $\mathbf{P}^{(1)}$ et $\mathbf{F}^{(1)}$ satisfaisant la contrainte γ.

L'étape de préparation

Cette étape consiste à appliquer partiellement les contraintes de γ une première fois sur les zones $\mathbf{P}$ et $\mathbf{F}$. Sont supprimées de $\mathbf{P}$ toutes les valuations dont au moins une horloge a une valeur supérieure à sa borne maximum dans γ. De même, sont supprimées de $\mathbf{F}$ toutes les valuations dont au moins une horloge à une valeur inférieure à sa borne minimum dans γ. Cela consiste ainsi à supprimer de ces zones les valeurs qui n'ont jamais et ne pourront jamais vérifier les contraintes de temps. Ainsi formellement on a :

$$\mathsf{prep}_F(\mathbf{F},\gamma,\mathbf{P}) = \left[\!\!\left[\left\{ f^p_{ij} = \left[\begin{array}{l} \mathsf{min}(f_{ij}, g_{ij}) \text{ if } j \neq 0 \\ f_{i0} - p_{i0} + g_{i0} \text{ if } j = 0, g_{i0} < p_{i0} \\ \text{otherwise } f_{ij} \end{array} \right. \right\}_{0\leq i,j\leq n} \right]\!\!\right]$$

Un élément important de cette représentation à trois DBM est que $\mathbf{F}$ est une translation de $\mathbf{P}$, ainsi toutes les modifications appliquées sur l'une des deux zones est répercutée sur l'autre afin qu'elles soient cohérentes. Si la borne supérieure d'une horloge est diminuée dans $\mathbf{P}$, alors on diminue d'autant la borne supérieure de cette horloge dans $\mathbf{F}$. Le calcul $f_{i0} - p_{i0} + g_{i0}$ indique la nouvelle valeur maximale de cette horloge quand c'est le cas, où $f_{i0} - p_{i0}$ permet de recalculer le délai qui a été lu, et g_{i0} est la nouvelle valeur maximale pour l'horloge dans $\mathbf{P}$. L'opérateur de forme canonique est appliqué sur la DBM résultant du calcul afin d'éviter d'éventuelles pertes de précision.

Il n'est pas nécessaire de mettre à jour $\mathbf{P}$ ici car elle n'est pas utilisée dans les calculs de l'étape suivante, ainsi toutes les modifications y seront appliquées simultanément dans le nœud trans_P. Cependant, la modification est illustrée dans l'exemple suivant afin de clarifier les modifications appliquées sur $\mathbf{F}$.

Exemple 5.2.2 *La Figure 5.3 illustre la phase de préparation de* $\mathsf{delay}(\delta_2, \mathbf{P^{(1)}}, \mathbf{W^{(1)}}, \mathbf{F^{(1)}})$ *où* $\delta_2 = (q_1, \emptyset, \varepsilon, \emptyset, c_1 \in [12, 14] \wedge c_2 \in [5, 8], \{c_1\}, q_2)$. *Cette figure représente les zones correspondant aux DBM sous la forme de polyèdres à 2 dimensions, car elles sont définies sur deux horloges. La contrainte est représentée par le rectangle* $\mathbf{G}$ *au centre de la figure. Les trois DBM initiales* $\mathbf{P^{(1)}}, \mathbf{W^{(1)}}, \mathbf{F^{(1)}}$ *y sont représentées, où leurs contours indiquent leurs valeurs au début de l'étape, et les zones remplies leurs valeurs après les calculs.*

La zone remplie $\mathbf{F}^p$ *est celle résultante du calcul de* $\mathsf{prep}_F(\mathbf{F^{(1)}}, \gamma, \mathbf{P^{(1)}})$. *Les valuations de* $\mathbf{F^{(1)}}$ *telles que* $c_1 < 12$ *ont été supprimées de la zone car elles n'ont jamais pu satisfaire la contrainte de temps lors de la translation de* $\mathbf{P}^{(1)}$ *à* $\mathbf{F}^{(1)}$. *Les valuations qui ont ainsi été supprimées correspondent à la zone indiquée par une croix sur la gauche de* $\mathbf{F}^p$. *De même, les valuations de* $\mathbf{P}^{(1)}$ *dont* $c_1 > 14$ *sont supprimées car, même si la valuation est incrémentée, elle ne peut pas satisfaire* γ. *Les valuations ainsi supprimées de* $\mathbf{P}^{(1)}$ *sont représentées par la zone indiqué*

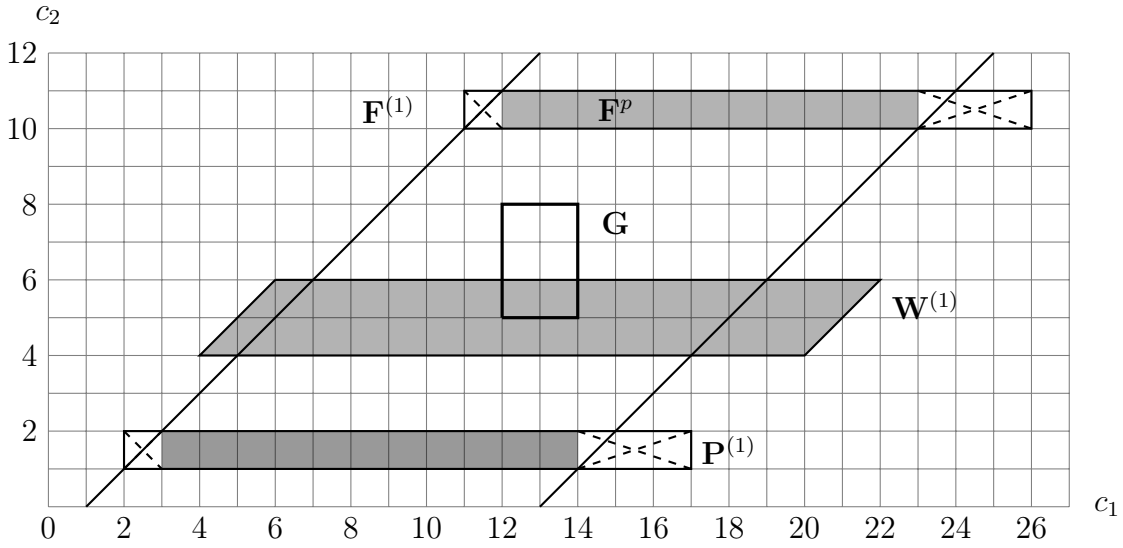


FIGURE 5.3 – Illustration de la phase de préparation de delay lors du franchissement de la seconde transition de la Figure 5.2.

par une croix sur la droite de $\mathbf{P}^{(1)}$*. Comme la zone* $\mathbf{F}$ *est une translation de* $\mathbf{P}$ *après l'incrément du délai lu, lorsqu'une valuation est supprimée d'une des zones, la valuation correspondante est supprimée de l'autre.*

Pour obtenir la valeur finale de $\mathbf{F}^p$ *l'opérateur de forme canonique permet de déterminer les contraintes diagonales de la matrice, et donc* $-1 \leq c_1 - c_2 \leq 13$*. Ainsi, à la fin de la première étape on obtient :*

$$\mathbf{F}^p = \begin{pmatrix} (0,\leq) & (-12,\leq) & (-10,\leq) \\ (23,\leq) & (0,\leq) & (13,\leq) \\ (11,\leq) & (-1,\leq) & (0,\leq) \end{pmatrix}$$

L'étape de franchissement

Dans cette seconde étape est mise à jour la zone $\mathbf{W}$ afin qu'elle contienne l'ensemble des valuations accessibles satisfaisant les contraintes temporelles et permettant de franchir la transition. Les valuations de $\mathbf{F}$ qui ne sont plus accessibles depuis $\mathbf{W}$ seront alors supprimées. La nouvelle valeur de $\mathbf{W}$ est calculée par :

$$\mathsf{trans}_W(\mathbf{W}, \mathbf{F}^p, \gamma) = [\![\mathsf{ext}(\mathbf{W}, \mathbf{F}^p) \cap \gamma]\!]$$

où $\mathbf{F}^p$ est le résultat de la fonction prep_F calculée à l'étape précédente.

Cette fonction calcule l'ensemble des valuations satisfaisant γ accessibles depuis $\mathbf{W}$. Pour ce faire, à l'aide de la fonction ext est créée la zone contenant l'ensemble

des valuations intermédiaires à $\mathbf{W}$ et $\mathbf{F}$, ce qui correspond donc à l'ensemble des valuations d'horloges accessibles en attendant avant de franchir la transition. Puis, en effectuant l'intersection de cette zone avec la zone représenté par γ on obtient la zone contenant toutes les valuations accessibles satisfaisant les contraintes de temps. Si le résultat de trans_W est une zone vide, vérifiable via le prédicat empty, alors la transition n'est pas franchissable car la contrainte de temps n'est pas satisfaisable. Si c'est le cas, la fonction delay n'a pas de résultat et les calculs suivants ne sont pas effectués.

Si la transition est franchissable, il faut mettre à jour la zone $\mathbf{F}$ et en supprimer les valuations qui ne sont plus accessibles depuis $\mathbf{W}$:

$$\mathsf{trans}_F(\mathbf{F}^p, \mathbf{W}^t) = \left[\!\!\left[f^t_{ij} = \left\{ \begin{array}{l} \mathsf{min}(w^t_{ij}, f^p_{ij}) \text{ if } i,j \neq 0 \\ \text{otherwise } f^p_{ij} \end{array} \right\}_{0 \leq i,j \leq n} \right]\!\!\right]$$

où $\mathbf{F}^p$ est le résultat de la fonction prep_F et $\mathbf{W}^t$ est le résultat de la fonction trans_W. La mise à jour de $\mathbf{F}$ consiste simplement à appliquer les contraintes diagonales de $\mathbf{W}^t$, les autres contraintes ayant déjà été traitées à l'étape de préparation. Il est aussi nécessaire de calculer la forme canonique de l'automate afin d'être sûr que les bornes supérieure et inférieure des horloges soient exactes dans $\mathbf{F}$ au début l'étape de réinitialisation.

La fonction trans_P consiste à synchroniser $\mathbf{P}$ avec la valeur courante de $\mathbf{F}$:

$$\mathsf{trans}_P(\mathbf{P}, \mathbf{F}, \mathbf{F}^t) = \left\{ p^t_{ij} = \left[\begin{array}{l} p_{ij} - f_{ij} + f^t_{ij} \\ \qquad \text{si } (i = 0 \vee j = 0) \wedge p_{i0} \neq (0, \leq) \\ \text{sinon } p_{ij} \end{array} \right\}_{0 \leq i,j \leq n}$$

Comme présenté sur le dataflow, la fonction trans_P prend en paramètre à la fois la valeur initiale de $\mathbf{F}$ et celle obtenue en résultat de trans_F, nommée $\mathbf{F}^t$. Cette fonction consiste à supprimer de $\mathbf{P}$ toutes les valuations d'horloges correspondantes à celles qui ont été supprimées de $\mathbf{F}$ jusqu'à l'étape courante. On peut remarquer que les contraintes diagonales de $\mathbf{P}$ ne sont jamais mises à jour car elles ne sont utilisées dans aucun calcul.

Exemple 5.2.3 *La Figure 5.4 illustre les différents calculs effectués dans l'étape de franchissement. Les valeurs des zones au début de l'étape sont représentées par leurs contours. Au début, les zones* $\mathbf{F}^p$ *et* $\mathbf{P}^p$ *ont été réduites dans l'étape de préparation, et* $\mathbf{W}^{(1)}$ *est le paramètre de* delay*. La zone* $\mathbf{P}^t$*, remplie dans l'illustration, est le résultat de la fonction* $funtrans_W$*, ce qui correspond à l'intersection de* γ *avec l'ensemble des valuations intermédiaires de* $\mathbf{W}^{(1)}$ *à* $\mathbf{F}^p$*. Ces valuations intermédiaires sont représentées par la zone hachurée et sont calculées à l'aide de la*

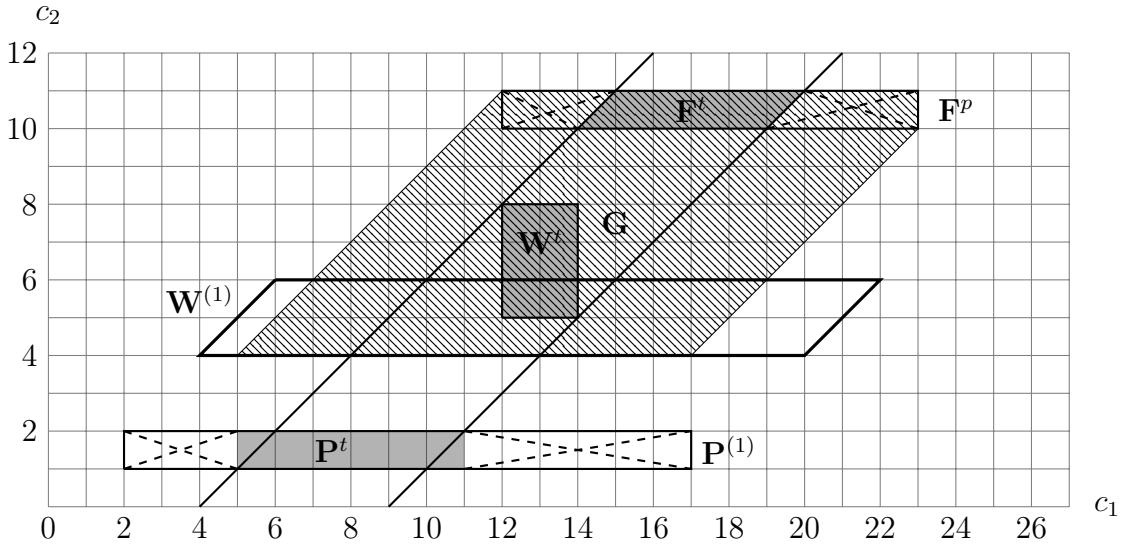


FIGURE 5.4 – Illustration de l'étape de franchissement de delay.

fonction ext *:*

$$\begin{array}{ccccc} \mathsf{ext}(\mathbf{W}^{(1)}, \mathbf{F}^p) & \cap & \gamma & = & \mathbf{W}^t \\ \begin{pmatrix} (0,\leq),(-4,\leq),(-4,\leq) \\ (23,\leq),(0,\leq),(13,\leq) \\ (11,\leq),(-1,\leq),(0,\leq) \end{pmatrix} & \cap & \gamma & = & \begin{pmatrix} (0,\leq),(-12,\leq),(-5,\leq) \\ (14,\leq),(0,\leq),(9,\leq) \\ (8,\leq),(-4,\leq),(0,\leq) \end{pmatrix} \end{array}$$

Les contraintes diagonales de $\mathbf{W}^t$ *sont appliquées sur la zone* $\mathbf{F}^p$ *afin d'en supprimer les valuations qui ne sont pas accessibles depuis la zone* $\mathbf{W}^t$*. La mise à jour de* $\mathbf{F}^t$*, illustrée par la zone remplie, est calculée avec la fonction* trans_F*, qui supprime de la zone les parties indiquées par des croix. La fonction* trans_P *est utilisée pour synchroniser les zones* $\mathbf{P}$ *et* $\mathbf{F}$ *en supprimant toutes les valuations de* $\mathbf{P}$ *qui ont été supprimées de* $\mathbf{F}$*.*

$$\mathbf{F}^t = \begin{pmatrix} (0,\leq),(14,\leq),(-10,\leq) \\ (20,\leq),(0,\leq),(9,\leq) \\ (11,\leq),(-4,\leq),(0,\leq) \end{pmatrix}; \mathbf{P}^t = \begin{pmatrix} (0,\leq),(-5,\leq),(-1,\leq) \\ (11,\leq),(0,\leq),(16,\leq) \\ (2,\leq),(0,\leq),(0,\leq) \end{pmatrix}$$

L'étape de réinitialisation

L'étape finale de delay réinitialise les horloges de ρ. Dans le dataflow, les fonctions correspondantes à cette étape sont préfixées par reset. Les zones resultantes des fonctions composant cette étape sont le résultat de delay.

La réinitialisation des horloges de $\mathbf{W}$ consiste simplement à calculer la projection de la zone sur les axes des origines correspondant aux horloges qui ont été réinitialisées.

$$\mathsf{reset}_W(\mathbf{W}^t, \rho, q') = [\![\mathbf{W}^t[\rho \leftarrow 0]]\!]$$

Le résultat de reset_W est la zone $\mathbf{W}^r$ contenant l'ensemble des valuations d'horloges représentant les valeurs des horloges une fois la transition franchie, arrivant dans la place q'.

Le calcul effectué par la fonction reset_F est le plus complexe de l'étape. La fonction est définie de la manière suivante :

$$\mathsf{reset}_F(\mathbf{F}^t, \rho, \mathbf{W}^t, \mathbf{W}^r, \mathbf{P}^t) =$$
$$[\![\{ f^r_{ij} = \begin{cases} \mathsf{min}(\{f^t_{k0} + w^t_{0k} \mid 1 \le k \le n\} \cup \{f^t_{k0} - p^t_{k0} \mid 1 \le k \le n\}) \\ \quad \text{si } c_i \in \rho, c_j = c_0 \\ \mathsf{min}(\{(0, \le)\} \cup \{f^t_{0k} + w^t_{k0} \mid 1 \le k \le n\}) \\ \quad \text{si } c_i = c_0, c_j \in \rho \\ w^r_{ij} \text{ si } i, j \neq 0 \\ \text{sinon } f^t_{ij} \end{cases} \}_{0 \le i,j \le n}]\!]$$

où $\mathbf{F}^t$ est le résultat de la fonction trans_F, $\mathbf{W}^t$ est le résultat de tran_W, $\mathbf{W}^r$ est le résultat de reset_W et $\mathbf{P}^t$ est le résultat de trans_P. Le résultat de cette fonction est la zone temporelle contenant toutes les valuations d'horloges accessibles dans l'état de destination de la transition. La difficulté de cette fonction est de retrouver l'intervalle de définition des horloges qui ont été réinitialisées à partir de $\mathbf{F}^t$ et $\mathbf{W}^t$. Au cours du chemin d'ε-transitions, les horloges ont pu être incrémentées d'une partie des valeurs du délai lu afin de satisfaire les contraintes de temps successives. Ainsi, l'intervalle de définition finale des horloges réinitialisées est l'intervalle de temps devant encore être incrémenté aux horloges après la réinitialisation.

La valeur finale maximale (resp. minimale) d'une horloge réinitialisée est la distance maximale (resp. minimale) séparant une valuation de $\mathbf{W}^t$ de la valuation correspondante dans $\mathbf{F}^t$. La distance entre une valuation de $\mathbf{P}^t$ et celle correspondante dans $\mathbf{F}^t$ et utilisée afin de gérer le cas particulier où la distance maximum entre une valuation de $\mathbf{W}^t$ et $\mathbf{F}^t$ est supérieure au délai lu. Les horloges qui n'ont pas été réinitialisées gardent les mêmes intervalles de définition. Enfin, les contraintes diagonales de $\mathbf{F}$ sont les mêmes que celles de $\mathbf{W}^r$, ainsi seules les valuations accessibles depuis celles de $\mathbf{W}^r$ sont retenues.

Enfin, il est possible de réinitialiser les horloges de ρ dans $\mathbf{P}$ avec la fonction

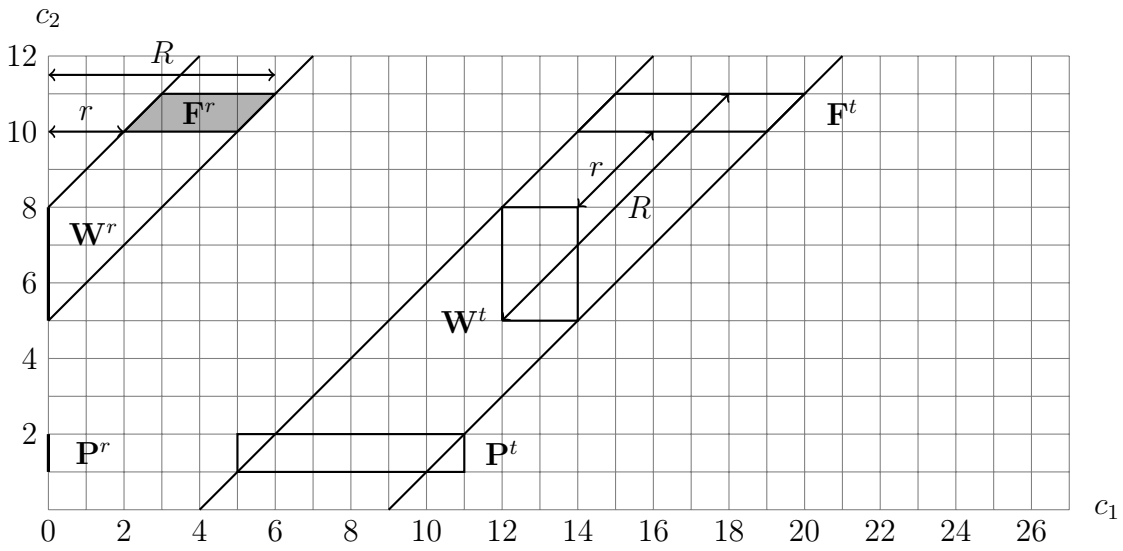


FIGURE 5.5 – Illustration de l'étape de réinitialisation de delay.

reset_p et de synchroniser les autres horloges avec les valeurs de $\mathbf{F}$:

$$\mathsf{reset}_P(\mathbf{P}^t, \rho, \mathbf{F}^t, \mathbf{F}^r) = \left\{ p^r_{ij} = \left[\begin{array}{l} (0, \leq) \text{ si } (i = 0, j \in \rho) \vee (i \in \rho, j = 0) \\ p^t_{ij} - f^t_{ij} + f^r_{ij} \text{ si } i = 0 \vee j = 0, p_{i0} \neq (0, \leq) \\ \text{sinon } p^t_{ij} \end{array} \right. \right\}_{0 \leq i,j \leq n}$$

où $\mathbf{P}^t$ est le résultat de trans_P, $\mathbf{F}^t$ est le résultat de trans_F et $\mathbf{F}^r$ est le résultat de reset_F. Toutes les horloges de ρ sont réinitialisées, c'est-à-dire que leurs intervalles de définition est $[0, 0]$. Les horloges de $\mathcal{P}$ qui ont été réinitialisées dans une transition précédente du chemin ne sont plus synchronisées avec celles de $\mathbf{F}$ et leurs valeurs sont fixées à 0. Les horloges qui ne sont pas réinitialisées sont celle synchronisées avec les nouvelles valeurs de $\mathbf{F}$, de la même manière que dans trans_P.

Exemple 5.2.4 *La Figure 5.5 représente l'étape de réinitialisation de la fonction* delay*, qui en est l'étape finale. Les zones résultantes de l'étape de franchissement* $(\mathbf{P}^t, \mathbf{W}^t, \mathbf{F}^t)$ *sont représentées dans la figure sous la formes de leurs contours. Les deux zones* $\mathbf{P}^r$ *et* $\mathbf{W}^r$*, résultantes des fonctions* reset_P *et* reset_W*, sont aplaties sur l'axe* c_2 *car l'horloge* c_1 *y a été réinitialisée et sa valeur est de* 0 *dans toutes leurs valuations.*

$$\mathbf{P}^r = \begin{pmatrix} (0, \leq), (0, \leq), (-1, \leq) \\ (0, \leq), (0, \leq), (16, \leq) \\ (2, \leq), (0, \leq), (0, \leq) \end{pmatrix}; \mathbf{W}^r = \begin{pmatrix} (0, \leq), (0, \leq), (-5, \leq) \\ (0, \leq), (0, \leq), (-5, \leq) \\ (8, \leq), (8, \leq), (0, \leq) \end{pmatrix}$$

La zone $\mathbf{F}^r$ *est représentée sur la forme de la zone remplie. Les horloges réinitialisées n'ont pas nécessairement pour valeur* 0 *dans* $\mathbf{F}$ *car cette zone représente l'ensemble des valuations d'horloges après que le délai lu soit totalement incrémenté sur les horloges. Pour calculer l'intervalle de valeurs dans lesquelles seront réinitialisées les horloges, il faut calculer l'intervalle du délai qui n'a pas encore été incrémenté sur les horloges. C'est l'intervalle* $[r, R]$ *de la distante minimale (représenté par la flèche* r*) et maximale (représenté par la flèche* R*) séparant une valuation de* $\mathbf{W}^t$ *de celle correspondante dans* $\mathbf{F}^t$*, avant les réinitialisations des horloges. Les contraintes diagonales de* $\mathbf{F}^r$ *sont celles de* $\mathbf{W}^r$ *afin d'uniquement garder les valuations accessibles depuis* $\mathbf{W}^r$*. Ainsi, après le calcul de la forme canonique de* $\mathbf{F}$*, on obtient :*

$$\mathbf{F}^r = \begin{pmatrix} (0,\leq), (-2,\leq), (-10,\leq) \\ (6,\leq), (0,\leq), (-5,\leq) \\ (11,\leq), (8,\leq), (0,\leq) \end{pmatrix}$$

Les DBM résultantes de l'étape de réinitialisation sont les résultats de delay*, elles peuvent être utilisées pour calculer le franchissement d'une transition suivante dans* walk*, et on note alors :* $\mathbf{P}^{(2)} = \mathbf{P}^r, \mathbf{W}^{(2)} = \mathbf{W}^r, \mathbf{F}^{(2)} = \mathbf{F}^r$*. De plus, comme c'est le cas dans l'exemple actuel, à la fin du chemin d'*ε*-transitions la zone* $\mathbf{F}^{(2)}$ *contient l'ensemble des valuations d'horloges accessibles dans l'état de place* q_2 *après la lecture du délai.*

Cet algorithme permet de calculer les zones temporelles accessibles par les ε-fermetures, et de représenter de manière fini l'ensemble d'états disposant de valuations d'horloges appartenant à ces zones. Cette abstraction dispose aussi d'une notion d'ordre partiel, où une zone A peut être incluse dans une zone B si toutes les valuations de A appartiennent aussi à B. Cela permet de mettre en place différentes optimisations lors de l'implémentation de l'algorithme en éliminant les états redondants.

Actuellement aucune abstraction similaire n'existe sur les contextes mémoire ou les places de l'automate. La quantité de variables et de places dépend de la structure de l'automate. Or, dans notre projet, on suppose que les ATCM manipulés par l'algorithme sont traduits à partir d'ETCM spécifiant les motifs recherchés. Ainsi, une manière de limiter le nombre d'états explorés par l'algorithme, et améliorer ses performances, est de générer des automates ne contenant pas de places ou de variables superflues.

5.3 L'étude des tailles des constructions d'automates à mémoire

Les ATCM sont utilisés comme un formalisme de reconnaissance pour les propriétés exprimées par des ETCM. La qualité de l'automate généré à partir d'une expression peut avoir un grand impact sur la durée de la reconnaissance, et plus généralement sur tous les types d'analyse du modèle. Bien entendu, il existe des cas particuliers où, avec des automates de très petite taille, la durée de la reconnaissance est très longue, et inversement la reconnaissance peut être très courte avec certains grands automates.

Cette étude est motivée par le constat que les constructions d'automates présentées sur les FMA [55] génèrent des automates avec un accroissement exponentiel du nombre de places en fonction de la taille de la mémoire. Cependant, les langages de spécifications basés sur les automates à mémoire (IARE, REM) [56, 63] se basent sur ces constructions pour prouver leurs propriétés de clôtures. Comme nous souhaitons exploiter les automates générés à partir des ETCM, il est important de contrôler la taille de l'automate afin de limiter le coût de la traduction, et ses conséquences sur les performances de l'algorithme de reconnaissance.

Dans la suite de la section, nous comparons pour différents modèles d'automates à mémoire les tailles des automates générés par les constructions correspondant aux opérateurs rationnels : l'union, la concaténation, l'étoile de Kleene et aussi l'intersection. Ces constructions sont étudiées car ce sont des opérations classiques sur les langages et qu'elles sont donc étudiées dans la littérature pour les différentes classes d'automate à mémoire.

5.3.1 Le coût combinatoire des constructions rationnelles sur les automates à mémoire

Dans cette section nous comparons les tailles des constructions pour trois modèles d'automates : les ACM, les FMA [55] et les HRA [46]. Le modèle des FMA est le modèle d'automate à mémoire de référence pour ce qui est de l'expression de langage sur alphabet infini et les HRA sont, d'une certaine manière, une généralisation des ACM. Les GRA et FRA ne sont pas pris en compte car leurs constructions se basent sur celles des FMA [55, 56]. De plus, l'ensemble des langages reconnus par les FRA n'est clos ni pour la concaténation ni pour l'étoile de Kleene. En comparant les tailles des constructions cela nous permet d'évaluer la qualité de la traduction des ECM en ACM par rapport aux autres langages de spécification de propriétés sur alphabet infini : les IARE [56] et REM [63] dont les formalismes de reconnaissance sont respectivement les GRA et les FMA.

Nous avons calculé les tailles des constructions des FMA présentées dans la

	Nombre de places	Nombre de transitions	Nombre de registres
FMA $\rightarrow$ M-FMA	$\lvert Q\rvert * (\lvert M\rvert !)$	$\lvert \Delta\rvert * (\lvert M\rvert !)$	$\lvert M\rvert + 1$
M-FMA $\rightarrow$ FMA	$\lvert Q\rvert * \lvert M\rvert^{\lvert M\rvert}$	$\lvert \Delta\rvert * \lvert M\rvert^{\lvert M\rvert}$	$\lvert M\rvert$

TABLE 5.1 – Traduction entre FMA et M-FMA

preuve du Théorème 3 de [55]. Pour les HRA, nous avons calculé les tailles des constructions présentées dans la Section 3 de [46]. Bien qu'aucune des constructions ne soit présentée comme optimale pour la taille des automates générés, même celles des ACM présentées dans la Section 4.3.1, ce sont les seules constructions disponibles dans la littérature.

Pour comparer la taille des automates, nous nous intéressons à trois critères :
— la quantité de places dans les automates, notée par $|Q|$,
— la quantité de transitions, notée par $|\Delta|$,
— et la quantité d'identifiants mémoire, notée par $|M|$.

Ce que nous appelons ici les identifiants mémoire sont les registres dans les FMA, les historiques dans les HRA et les variables $X^l \in V \times L$ dans les ACM. Les identifiants sont les seuls éléments dans la structure des automates permettant de quantifier la taille de la mémoire. Même si les fonctionnements des mémoires dans les modèles sont différents, principalement le fait que HRA et ACM peuvent stocker plusieurs lettres dans un même historique/variable tandis que les FMA en stockent au maximum une. Certaines constructions des HRA dépendent des lettres initialement stockées dans les historiques, aussi nous notons par $|\Sigma|$ le nombre de lettres stockées initialement dans la mémoire d'un automate. Les tailles présentées ici sont les estimations au pire cas des tailles des automates construits.

Avant de commencer à évaluer les tailles des constructions rationnelles, il faut savoir qu'aucune des constructions n'est présentée directement sur les FMA, mais en réalité sur un modèle d'automate équivalent appelé *M-Automate* (M-FMA). Ainsi, chaque FMA doit être converti en M-FMA pour pouvoir construire l'automate reconnaissant le langage souhaité. La Table 5.1 présente la taille des automates créés pour traduire des FMA et M-FMA et inversement. Les FMA et M-FMA résultant des traductions utilisent approximativement les mêmes quantités de registres. Cependant, comme l'utilisation de la mémoire y est très différente, le nombre de places contenu par l'automate est exponentiellement accru par l'ajout de métadonnées indiquant comment utiliser correctement les registres. La duplication des places s'ensuit d'une duplication des transitions.

Les constructions représentant l'intersection des langages $\mathcal{L}(A_1) \cap \mathcal{L}(A_2)$ sont des adaptations de celle du produit synchrone d'automates finis, et ceux dans les trois modèles. Ainsi, dans les trois modèles d'automates : M-FMA, HRA and

	Nombre de places	Nombre de transitions
M-FMA	$\lvert Q_1\rvert + \lvert Q_2\rvert$	$\lvert\Delta_1\rvert * 2^{\lvert M_2\rvert} + \lvert\Delta_2\rvert * 2^{\lvert M_1\rvert}$
HRA	$(\lvert Q_1\rvert + \lvert Q_2\rvert) * 2^{\lvert\Sigma_2\rvert * \lvert M_2\rvert}$	$(\lvert\Delta_1\rvert + \lvert\Delta_2\rvert) * (\lvert\Sigma_2\rvert + 1) * 2^{\lvert\Sigma_2\rvert * \lvert M_2\rvert}$
ACM	$\lvert Q_1\rvert + \lvert Q_2\rvert$	$2 * \lvert\Delta_1\rvert + \lvert\Delta_2\rvert$

	Nombre de registres
M-FMA	$\lvert M_1\rvert + \lvert M_2\rvert$
HRA	$\mathsf{max}(\lvert M_1\rvert, \lvert M_2\rvert) + \lvert\Sigma_2\rvert$
ACM	$\lvert M_1\rvert + \lvert M_2\rvert$

TABLE 5.2 – Taille des constructions de la concaténation $\mathcal{L}(A_1) \cdot \mathcal{L}(A_2)$

ACM l'automate généré contient $|Q_1| * |Q_2|$ places et $|\Delta_1| * |\Delta_2|$ transitions. Même si les mémoires fonctionnent de manière très différentes, le nombre d'identifiants mémoires dans les trois constructions croit de manière identique : $|M_1| + |M_2|$.

Pour l'union des langages $\mathcal{L}(A_1) \cup \mathcal{L}(A_2)$, les M-FMA et HRA utilisent de nouveau la construction du produit synchrone, ainsi les tailles des automates construits sont les mêmes que pour celles de l'intersection. Dans les ACM, la construction est celle des automates finis qui consiste à faire l'union des places et transitions des deux automates et à ajouter une nouvelle place initiale ainsi que des ε-transitions permettant d'accéder de manière non déterministe aux places initiales des deux automates. Cette simple construction est bien plus petite que celle du produit synchrone car le nombre de places y est $|Q_1|+|Q_2|+1$ et de transitions $|\Delta_1|+|\Delta_2|+2$. Cependant, dû à la nature des transitions dans les M-FMA et HRA, elle n'y est pas possible. Mais elle semble pouvoir être réalisable directement sur les FMA, si les allocations initiales des registres entre les deux automates ne contiennent pas de lettre en commun.

La concaténation de langage est la construction la moins naturelle dans les automates à mémoire. Dans les langages sur alphabet infini, on s'attend à une évolution de la mémoire durant la reconnaissance qui laisse place à ce qu'on appelle plus haut des *effets de bord* dans les langages concaténés. Cependant, on s'intéresse ici à la concaténation comme elle est définie dans les langages rationnels, c'est-à-dire que les langages concaténés sont indépendants. Les constructions dans les trois modèles sont relativement similaires à la construction de la concaténation dans les automates finis.

Dans les M-FMA, l'automate construit utilise l'ensemble des registres des deux automates concaténés, ainsi que leurs valuations initiales. Cependant, à cause du fonctionnement des transitions, similaire à celui des HRA, il devient nécessaire de dupliquer toutes les transitions pour chaque sous-ensemble de registres de l'autre automate. Ainsi, il y a une explosion combinatoire du nombre de transitions dans l'automate résultant de la construction.

	Nombre de places	Nombre de transitions	Nombre de registres
M-FMA	$\|Q\| * 2^{\|M\|}$	$\|\Delta\| * 2^{\|M\|}$	$2 * \|M\|$
HRA	$\|Q\| * 2^{\|\Sigma\|*\|M\|}$	$\|\Delta\| * 2^{\|\Sigma\|*\|M\|} * (\|\Sigma\| + 1)$	$\|M\| + \|\Sigma\|$
ACM	$\|Q\| * 2^{\|M\|}$	$\|\Delta\| * 2^{\|M\|} * 2^{\|L\|}$	$2 * (\|M\| + \|L\|)$

TABLE 5.3 – Taille des constructions pour l'étoile de Kleene $\mathbb{L}(A)^*$.

Dans les HRA, avant que la construction ne soit effectuée, toutes les valeurs initialement associées aux historiques de l'automate suffixe sont extraites des deux automates et associées à de nouveaux historiques. De cette manière, ces historiques peuvent être utilisés quand l'on souhaite lire ces valeurs spécifiquement, mais surtout on s'assure de ne pas perdre les valeurs initiales des historiques de l'automate suffixe lors de la reconnaissance d'un préfixe. Cependant, lorsque ces valeurs sont extraites, il est nécessaire d'ajouter des transitions afin de préserver le langage de l'automate. Ainsi dans l'automate construit par la concaténation, les transitions menant à la place initiale de l'automate suffixe réinitialisent tous les historiques sauf ceux contenant les lettres extraites.

Dans les ACM, la construction de la concaténation fonctionne directement à un renommage des couches près.

La construction de l'étoile de Kleene nécessite de s'assurer que le langage reconnu à chaque itération est exactement le même. Ainsi, cela consiste dans les automates à mémoire à s'assurer que la mémoire soit réinitialisée à sa valeur initiale à chaque itération.

La construction dans les HRA consiste dans un premier temps à extraire toutes les lettres initialement stockées dans les historiques et à les stocker dans de nouveaux historiques dédiés, comme dans la construction de la concaténation. Il est de nouveau nécessaire de dupliquer les transitions et les places afin que l'automate reconnaisse toujours le même langage. Ainsi à la fin de chaque itération il suffit de réinitialiser tous les autres historiques afin de réinitialiser la mémoire à sa valeur initiale.

La construction utilisée dans les ACM pour effectuer l'étoile de Kleene est inspirée sur celle des M-FMA. Elle consiste à doubler la mémoire, la première moitié est utilisée afin de préserver la valeur initiale de la mémoire et la seconde est utilisée pour stocker les nouvelles lettres qui seront lues au cours d'une itération. Ainsi, réinitialiser la mémoire consiste à supprimer toutes les lettres de la seconde moitié. Cependant, pour que l'automate reconnaisse le même langage, il est nécessaire d'ajouter des informations dans les places afin d'utiliser correctement la mémoire, et donc de dupliquer les transitions de manière exponentielle.

L'analyse des tailles des constructions de concaténation et d'étoile de Kleene montre que si la mémoire ne contient initialement aucune lettre, alors les construc-

tions des HRA sont les plus petites. Cependant, leurs tailles croissent de manière exponentielle par rapport au nombre de lettres dans la valeur initiale de la mémoire. Les constructions pour les ACM et M-FMA ne dépendent pas de la valeur initiale de la mémoire. Les constructions des M-FMA sont exponentielles pour la concaténation et l'étoile de Kleene et à cela peuvent s'ajouter les traductions depuis les FMA si les automates d'origines n'étaient pas des M-FMA. Dans le cas des ACM, à par pour l'étoile de Kleene, les constructions de chaque opérateur rationnel sont d'une taille similaire à celles dans les automates finis. La construction de l'étoile de Kleene dans les ACM fait croître de manière exponentielle les nombres de places et de transitions de l'automate en fonction de la taille de la mémoire, et cela de manières similaire à celle des M-FMA. La section suivante présente une variante des ACM dans laquelle la construction de l'étoile de Kleene est de taille raisonnable.

5.3.2 Les ACM$_\pi$, une réduction de la construction de l'étoile

Motivé par la taille insatisfaisante de la construction de l'étoile de Kleene dans les ACM, nous proposons une extension des ACM, appelée ACM$_\pi$. La caractéristique essentielle de ce modèle est de permettre de copier les lettres stockées dans une variable vers une variable d'une autre couche. À l'aide de cette mécanique, une transition peut simplement réinitialiser l'intégralité des variables d'une couche à celles d'une autre couche. Ainsi dans la construction de l'étoile de Kleene il n'est plus nécessaire de dupliquer les places et transitions car il devient simple de réinitialiser les variables à leurs valeurs initiales à chaque itération.

Les automates à couches mémoire avec transfert

La définition des ACM$_\pi$ est une extension des ACM dont la signature des transitions est modifiée. Un ensemble de paires de variables y est ajouté.

Définition 5.3.1 (Transition) *Les transitions des ACM$_\pi$ sont des n-uplets de la forme :*

$$\delta = (q, \nu, \alpha, \overline{\nu}, \pi, q') \in \Delta,$$

où

- $q, q' \in Q$ *sont les places d'origine et de destination,*
- $\nu \subseteq 2^{V \times L}$ *est l'ensemble des variables modifiables dans la transition,*
- $\alpha : L \to V \cup \{\sharp\}$ *est la fonction indiquant pour chaque couche la variable consultée.*
- $\overline{\nu} \subseteq 2^{V \times L}$ *est l'ensemble des variables réinitialisées.*
- $\pi \subseteq 2^{(V \times L)^2}$ *est un ensemble des couples de variables, tel que pour chaque* $(X^l, Y^k) \in \pi$ *les lettres associées à* X^l *deviennent aussi associées à* Y^k.

Définition 5.3.2 (Le franchissement d'une transition) *Soit une transition δ, un contexte mémoire M, et une lettre a et $M' = \sigma(\delta, M, e)$ le contexte mémoire calculé par la fonction σ de la Définition 4.1.3. Le contexte mémoire résultant du franchissement de la transition δ est donné par :*

$\sigma_\pi(\delta, M, e) =$

$$\begin{cases} \bot & \text{if } M' = \bot & (1) \\ & \vee\ \exists (X^l, Y^k), (Z^p, W^k) \in \pi^2, Y \neq W, M'(X^l) \cap M'(Z^p) \neq \emptyset & (2) \\ & \vee\ \exists (X^l, Y^k) \in \pi, \exists Z \neq Y, M'(X^l) \cap M'(Z^k) \neq \emptyset & (3) \\ M'' & \text{otherwise, where } M''(Y^k) = M'(Y^k) \underset{(X^l, Y^k) \in \pi}{\cup} M'(X^l) & (4) \end{cases}$$

Il existe trois situations où la transition ne peut pas être franchie, indiquées par les cas (1) à (3) :

- (1) la transition ne peut pas être franchie dans les ACM, c'est-à-dire que le résultat de σ est $\bot$.
- (2) deux variables étant associées à une même lettre ont leurs valeurs copiées vers différentes variables d'une même couche, brisant la contrainte d'injectivité.
- (3) une variable de la couche de destination, autre que celle devant être associée aux lettres copiées, est déjà associée à l'une de ces lettres.

La situation (4) est celle où la transition est franchissable, et les lettres copiées sont ajoutées à celle de la variable de destination, en plus de celles qui y sont déjà associées.

Le modèle des ACM_π est au moins aussi expressif que le modèle des ACM, car chaque ACM est simplement un ACM_π avec des π vides. Cependant, les ACM_π semblent plus expressifs que les ACM et nous n'avons pas de preuve d'équivalence avec les ECM.

Propriété 5.3.1 *Pour toute ECM e et son contexte mémoire initial M, il existe un ACM_π A tel que $\mathbb{L}_M(e) = \mathbb{L}(A)$.*

Preuve : Pour toute ECM, il est possible de construire l'ACM_π reconnaissant le langage qu'elle exprime avec les constructions de la Section 4.3.1, où π est vide pour toutes les transitions des constructions. Si $\pi = \emptyset$ alors le résultat de σ_π est égal au résultat de σ. En effet, les cas d'échecs (2) et (3) ne peuvent jamais arriver quand π est vide. Les seuls cas pouvant être vérifiés sont le (1) qui est le cas d'échec quand σ a échoué, et le (4) dont le résultat est le contexte mémoire dans l'état d'arrivée. Quand π est vide le contexte mémoire M'' est égal à celui résultant de σ (M'). □

La construction de l'itération

La construction de l'étoile de Kleene est beaucoup plus concise avec un ACM_π car il est trivial de réinitialiser les variables à leurs valeurs initiales au début de chaque itération.

Étant donné une ECM e et son contexte initial M, et soit l'ACM $A = (Q, q_0, F, \Delta, V, L, M_0)$ créé à partir des constructions de la Section 4.3.1 et celle ci, on peut construire l'automate suivant reconnaissant le langage $\mathbb{L}_M(e^*/\bar{l}/)$

$$A_* = (Q \cup \{q'_0, q_\varepsilon\}, q'_0, F \cup \{q_\varepsilon\}, \Delta' \cup \Delta_f \cup \Delta_\varepsilon, V, L \cup L_w, M_0 \cup \{X^l \to \emptyset \mid X^l \in V \times L_w\})$$

La structure de l'automate est alors très classique, l'automate A_* contient toutes les places de A plus une nouvelle place initiale q'_0 et une finale q_ε. Les transitions de Δ' sont une réécriture des transitons de Δ utilisant les nouvelles couches, et Δ_f sont les transitions permettant d'itérer sur le langage de A. Les ε-transitions de Δ_ε permettent d'accéder de q'_0 à q_ε pour reconnaître le mot vide ou à q_0 pour reconnaître un mot de A.

Comme dans la construction présentée dans la Section 4.3.1, un nouvel ensemble de couches L_w est ajouté afin de simuler l'évolution des couches n'appartenant pas à $\bar{l}$ et ainsi préserver les valeurs initiales des variables. Ainsi L_w contient une couche pour chaque couche de L absente de $\bar{l}$:

$$L_w = \{l_w \mid l \in L \backslash \bar{l}, l_w \notin L\}.$$

On définit la fonction $u(l) = \begin{cases} l & \text{si } l \in \bar{l} \\ l_w & \text{sinon} \end{cases}$ qui indique pour chaque couche de A celle qui la remplace dans A_*.

Les transitions de Δ sont réécrites dans Δ' afin qu'elles utilisent les variables des couches de L_w.

$$\Delta' = \{(q, \nu', \alpha', \overline{\nu}', \pi', q') \mid (q, \nu, \alpha, \overline{\nu}, \pi, q') \in \Delta\}$$

où $\nu' = \{X^{u(l)} \mid X^l \in \nu\}$, $\overline{\nu}' = \{X^{u(l)} \mid X^l \in \overline{\nu}\}$, $\pi' = \{(X^{u(l)}, Y^{u'k}) \mid (X^l, Y^k) \in \pi\}$ et $\alpha'(l) = \begin{cases} \alpha(l) & \text{si } l \in \bar{l} \\ \alpha(l') & \text{si } l = u(l') \in L_w \\ \sharp & \text{sinon} \end{cases}$.

Les variables de L_w doivent être initialisées au début de chaque itération aux valeurs qu'elles ont dans L, pour cela on utilise π. La première transition de chaque itération est une transition de Δ_ε ou de Δ_f qui sont définis ainsi :

$$\Delta_\varepsilon = \{(q'_0, \emptyset, \varepsilon, \emptyset, \{(X^l, X^{l_w}) \mid l \notin \bar{l}\}, q_0), (q'_0, \emptyset, \varepsilon, \emptyset, \emptyset, q_\varepsilon)\}$$

$$\Delta_f = \{(q, \nu, \alpha, \overline{\nu} \cup (V \times L_w), \{(X^l, X^{l_w}) \mid l \notin \bar{l}\}, q_0) \mid (q, \nu, \alpha, \overline{\nu}, \emptyset, q_f) \in \Delta', q_f \in F\}$$

Les transitions de Δ_ε sont les ε-transitions sortantes de q'_0. Celle de destination q_ε permet de représenter le mot vide et celle dont la destination est q_0 débute une itération. Cette dernière utilise π pour initialiser les variables de L_w aux valeurs des variables correspondantes dans L. Les transitions de Δ_f sont des copies des transitions finales de Δ' dont la destination est modifiée en q_0, l'état initial de A. Ce sont les transitions permettant d'itérer sur A, et elles vont réinitialiser les variables de L_w puis y assigner de nouveau leurs valeurs initiales avec π. Comme aucune des constructions présentées ne permet d'avoir une transition finale utilisant π, il est assuré que les transitions copiées ici vérifient $\pi = \emptyset$ ainsi aucun conflit ne peut arriver.

Il reste à montrer que A_* vérifie la Propriété 4.3.1 pour l'expression $e^*/\bar{l}/$ afin que cette construction soit correcte. C'est-à-dire que pour tous les contextes mémoire M_i et M_f pour chaque $w \in \mathbb{L}_{M_i}^{M_f}(e^*/\bar{l}/)$ il est possible de franchir un chemin de transitions allant de l'état (q'_0, M'_i) à un état (q_f, M'_f) où $q_f \in F$ et les variables $X^l \in V \times L$ de M_i et de M'_i ont les mêmes valeurs, et de même pour M_f et M'_f. La sémantique de l'itération est indiquée dans la Table 3.3 et est : $\mathbb{L}_{M_i}^{M_f}(e^*/\bar{l}/) = \{\varepsilon \mid M_i = M_f\} \cup \bigcup_{M'} \mathbb{L}_{M_i}^{M'}(e) \cdot \mathbb{L}_{\mathsf{comp}(M_i,\bar{l},M')}^{M_f}(e^*/\bar{l}/)$.

Il est évident que le mot vide appartient bien au langage de A_* avec l'ε-transition allant de q'_0 à q_ε. Par hypothèse d'induction, nous savons que A vérifie la Propriété 4.3.1 avec l'expression e. Dans toutes les transitions de Δ, les couches $l \in L \backslash \bar{l}$ sont remplacées par les couches correspondantes dans L_w. Donc aucune des variables des couches $L \backslash \bar{l}$ ne peut être modifiée lors d'une itération. Toutes les modifications qui devraient être effectuées sur ou depuis ces variables (avec π) sont effectuées sur les couches correspondantes de L_w. Comme au début de chaque itération, toutes ces variables sont réinitialisées aux valeurs M_i préservées dans les couches de L, alors la dynamique de l'expression e est exactement reproduite à chaque itération sur L_w. On peut en déduire que la Propriété 4.3.1 est vérifiée par cette construction.

Cette variante des ACM permet de réduire la taille de la construction de l'itération d'un facteur exponentiel. Le nombre de places de A_* est $|Q| + 2$ et la quantité de transitions double au pire cas : si toutes les transitions de Δ sont finales. La taille du contexte mémoire ne change pas par rapport à la construction des ACM, et la quantité de variables va au pire doubler. Ce modèle est utilisé comme une optimisation locale permettant d'améliorer les performances de la reconnaissance. Nous avons implémenté les ACM_π avec cette construction de l'itération et l'algorithme de reconnaissance pour résoudre le problème de la détection de motifs représentés sous forme d'ETCM.

5.4 Les expérimentations

L'un des objectifs principaux de cette thèse est d'être capable de reconnaître les propriétés définies sous forme d'ETCM voire d'ATCM. Nous avons développé l'outil PaMaTiNA implémentant les différents modèles d'automates que nous avons définis dans cette thèse ainsi que l'algorithme de reconnaissance présenté dans la Section 5.1. L'outil permet aussi de spécifier des motifs sous forme d'ETCM qui sont ensuite traduits en automates équivalents à l'aide de l'algorithme présenté dans la Section 4.4.2. Une implémentation de l'outil est disponible en ligne [1]. La suite de cette section présente les résultats des expérimentations faite avec notre outil sur des motifs décrits dans les flots de liens venant de données du monde réel, et non des données générées.

Comme présenté dans la Propriété 4.2.14, le problème de l'appartenance dans les ACM est NP-difficile. Ainsi les expérimentations ont un certain enjeu pour ce qui est des performances de notre outil et de ses potentielles applications. Les expérimentations présentées ici sont effectuées sur deux flots de liens créés depuis des deux jeux de données réels. Le premier est l'enregistrement du trafic réseaux sur un important routeur trans-pacifique [42], et l'autre est un historique d'un mois de tweets publics envoyés sur Twitter France.

La première analyse que nous faisons est effectuée dans le cas du trafic internet, nous souhaitons détecter des attaques coordonnées similaires à une attaque par déni de service (DDOS). Nous représentons ce scénario à l'aide du motif du diamant, qui est identifié par [83] comme significatif à cet égard. Nous le représentons par l'ETCM présentée dans la Section 3.5 :

$$(\sharp O^1 \rightsquigarrow \sharp S^1) \cdot (O^1 \rightsquigarrow \sharp S^1)^*/\emptyset/ \\ \cdot \langle ((S^1 \rightsquigarrow \sharp T^1)\&(\sharp H^2 \rightsquigarrow \sharp @)) \cdot ((S^1 \rightsquigarrow T^1)\&(\sharp H^2 \rightsquigarrow @))^*/\emptyset/\rangle_{[0,\delta]}$$

Le trafic réseaux enregistré depuis le routeur contient approximativement un lien toutes les $2\mu s$, et l'enregistrement dure une journée entière. Il est ainsi évident qu'il n'est pas possible de détecter l'intégralité des motifs non temporisés dans les flots de liens. Dans le contexte du DDOS, les attaques sont coordonnées et soudaines, aussi la fenêtre de temps dans laquelle elles ont lieu doit être restreinte.

La Figure 5.6 présente les résultats de l'expérimentation pour deux tailles de la fenêtre de temps δ. Les diagrammes présentent les temps d'exécution totaux au cours de la lecture du flot de liens (la courbe bleue) et le nombre d'instance du motif détecté (en rouge). Comme l'on peut s'y attendre, le nombre d'instances détectées est plus important sur une fenêtre de temps plus grande. Cependant, le nombre de liens traités sur une même durée (85 heures dans ces expérimentations) est bien moins important. Bien que notre implémentation ne soit pas optimale, on

1. Lien vers la page github de l'outil : `https://github.com/clementber/MaTiNA`

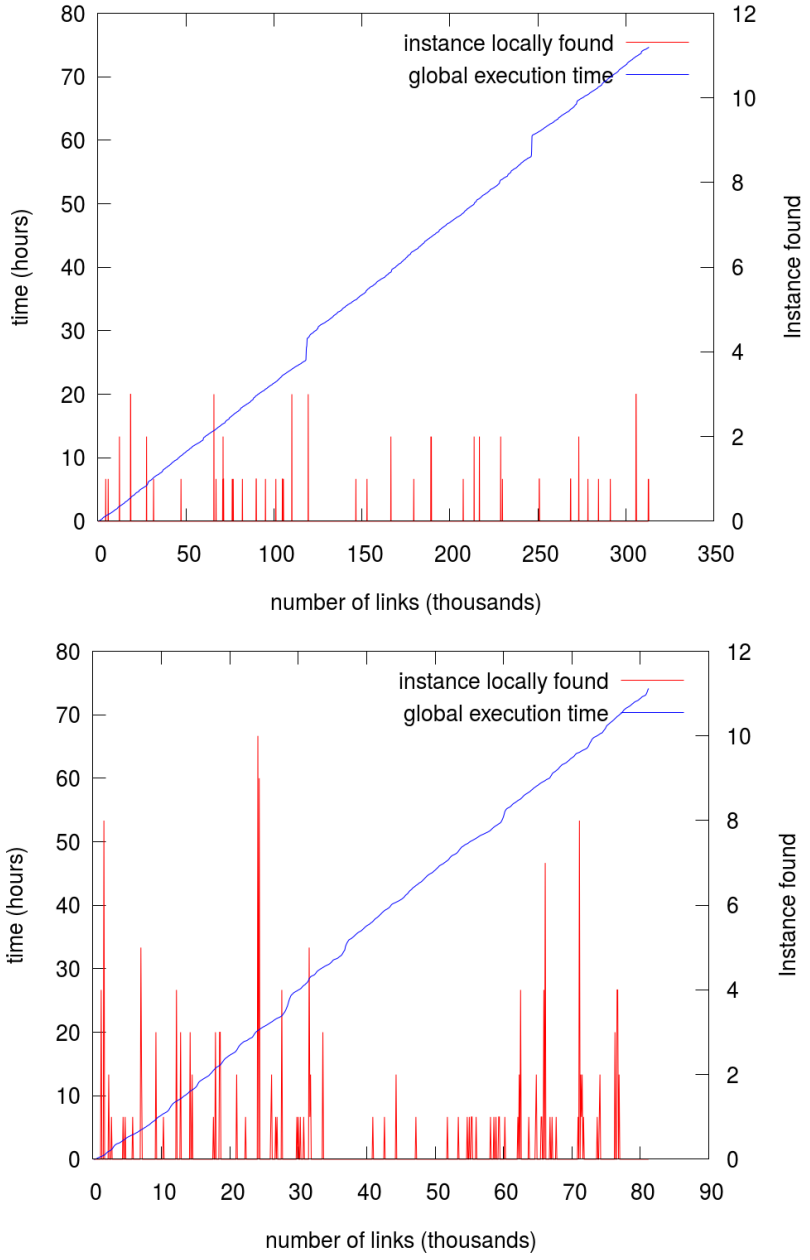


FIGURE 5.6 – Temps de reconnaissance du motif du DDOS sur une fenêtre de temps de $\delta = 0.01$ (haut) et $\delta = 0.02$ (bas).

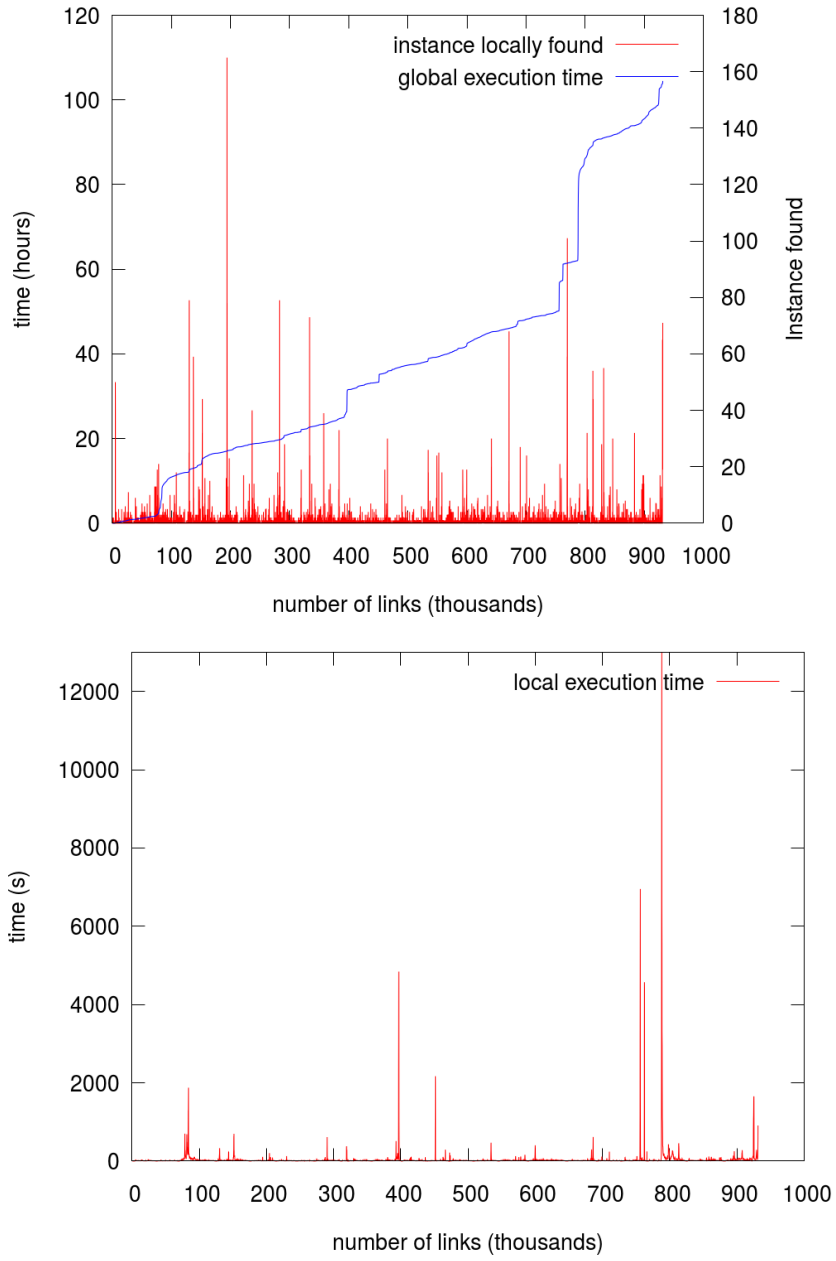


FIGURE 5.7 – Temps de reconnaissance du motif du triangle dans les échanges sur Twitter avec le temps de calcul et nombre d'instance détecter (haut) temps de calculs local (bas).

peut clairement observer un temps de calcul linéaire par rapport aux nombres de liens traités.

La seconde expérimentation effectuée cherche à détecter des communautés parmi les utilisateurs de Twitter. Le flot de liens traité ici est l'historique de communication d'un mois de tweets publics en France, comptabilisant 1,3 million de liens [2]. Le motif que nous recherchons est un graphe complet non dirigé entre k utilisateurs, pour un k donné, c'est-à-dire une clique de taille k apparaissant dans une fenêtre de temps de 10 minutes. La Figure 5.7 contient des diagrammes présentant les résultats de la recherche de cliques de taille $k = 3$, donc des triangles. Le premier diagramme est similaire à ceux de la précédente exécution, la courbe bleue représente le temps total d'exécution par rapport au nombre de liens lus, et l'histogramme rouge indique le nombre d'instances localement détectées. La courbe indiquant le temps total d'exécution ne semble ici plus linéaire, ou uniquement des segments séparés par des pics du temps de calcul. Le second diagramme présente les temps de calculs locaux aux sous-segments du flot de liens, et on peut y observer bien plus nettement ces pics du temps de calcul. Ils correspondent en réalité à des pics d'activité sur Twitter où la quantité de messages est bien plus importante sur une courte durée. Sur ces segments, l'outil doit traiter une quantité de données bien plus importante, ce qui engendre un accroissement du temps de traitement pour chaque lien lu.

5.5 Synthèse

Nous avons discuté dans ce chapitre de l'algorithme de reconnaissance conçu pour les automates temporisés à couches mémoire. La partie technique du calcul des zones temporelles a été particulièrement détaillée, car elles sont une abstraction permettant de représenter l'ensemble des valuations d'horloges accessibles lors de la lecture d'un délai. Nous avons adapté les *difference bound matrix* (DBM), couramment utilisées dans les techniques de *model checking*, à la problématique de la reconnaissance de langage temporisé. C'est cependant une contribution que l'on a publiée [19] simultanément à [10] qui propose une solution similaire à la nôtre.

Un autre point abordé dans ce chapitre est la qualité de la traduction des ETCM en ATCM avec une étude qui porte sur la taille des constructions d'automates. Nous avons calculé et comparé les tailles des automates à mémoire construits pour reconnaître les langages produits par les opérations rationnelles, et ce pour les différentes classes d'automates à mémoire abordées dans la thèse. Cela nous permet de mettre en avant un accroissement exponentiel de la taille des ACM générés pour les constructions de l'étoile de Kleene et de l'itération. Nous proposons

2. Les données viennent du projet Politoscope du CNRS Institut des Systèmes Complexes Paris Ile-de-France (`https://politoscope.org`).

ainsi une extension de modèle d'automates qui permet de réduire d'un facteur exponentiel la taille de l'automate généré.

Enfin, le chapitre se conclut par la présentation de quelques expérimentations effectuées avec notre prototype d'outil de vérification dynamique. Ces expérimentations permettent de mettre en avant les avantages de l'utilisation simultanée de contrainte de temps avec la mémoire mais aussi leurs limitations.

Conclusion

Dans cette thèse nous avons appliqué la théorie des langages et des automates à la formalisation et la détection de motifs dans les systèmes complexes, en particulier la représentation par des flots de liens. Pour ce faire, nous avons fait l'analogie entre ces derniers et les mots temporisés sur alphabets infinis.

Nous avons défini le formalisme des *expressions temporisées à couches mémoire* (ETCM), qui est, à notre connaissance, le premier langage de spécification générique pour les langages temporisés sur alphabet infini. Les ETCM sont une extension des expressions régulières contenant des opérateurs permettant d'exprimer des contraintes de temps, ainsi que des variables pour représenter des lettres de l'alphabet infini. Le sous-ensemble non-temporisé des ETCM, les *expressions à couches mémoire* (ECM), est strictement plus expressif que les langages similaires de la littératures : IARE [56] et REM [63]. L'un des avantages des ECM est la possibilité d'exprimer des langages contenant une quantité arbitraire de lettres différentes. Une contribution importante est la notion de couches mémoire qui permet de spécifier de manière modulaire des propriétés complexes sur les données. Avec les opérateurs de conjonction et de mélange, les couches permettent de définir des propriétés simultanées ou concurrentes assez simplement. Une autre contribution à la théorie des langages temporisés est la définition des opérateurs de spécification de contraintes de temps des ETCM. Ils reprennent une syntaxe similaire aux *balanced timed regular expressions* [8] pour abstraire en partie la notion d'horloge, mais avec les avantages d'avoir une grammaire inductive et une sémantique plus directe.

Nous avons introduit plusieurs opérateurs dérivés sur les ETCM, tels que le mélange $\otimes$ et l'opérateur de lien $\rightsquigarrow$ qui permettent de spécifier des motifs sur les flots de liens. Ils nous ont permis d'exprimer plusieurs motifs classiques pour l'analyse des systèmes complexes tels que les triangles et les chemins et permettent également de spécifier des propriétés plus complexes de manière générique et modulaire. Les ETCM sont une contribution dans le domaine des flots de liens, où elles sont le premier langage de spécification formelle de motifs générique.

Une autre contribution significative de cette thèse est le modèle des *automates temporisés à couches mémoire* (ATCM), qui est le formalisme de reconnaissance

pour les ETCM. Pour démontrer la complétude de ce formalisme pour notre problématique, nous avons prouvé, tel le théorème de Kleene, l'équivalence entre l'ensemble des langages reconnaissables par les ATCM et celui exprimable par les ETCM. Cette preuve met en jeu des algorithmes permettant de *traduire* les ETCM en ATCM et inversement. Les ATCM sont un hybride entre les automates temporisés et les *automates à couches mémoire* (ACM), que nous avons aussi défini. Afin de pouvoir caractériser précisément les propriétés exprimables sur les langages de données, nous avons positionné précisément notre modèle par rapport à ceux présents dans la littérature. Ce positionnement nous a aussi permis de déduire que le problème de la reconnaissance (*membership*) est décidable, et NP-difficile, pour les ACM, ainsi que le problème du langage vide (*emptiness*).

Une autre problématique abordée dans cette thèse a été la taille des constructions d'automates. Comme les motifs sont spécifiés par les ETCM et que la reconnaissance est sur les ATCM, la qualité des automates générés influe sur les performance de la reconnaissance. Ainsi, nous avons comparé les tailles des ATCM générés pour tous les opérateurs rationnels par rapport à celles que l'on trouve dans la littérature. Nous avons ainsi pu mettre en évidence les explosions combinatoires des constructions présentées pour les FMA [55] et les HRA [46]. Nous pouvons aussi constater que les couches mémoire permettent effectivement d'obtenir des automates de tailles comparables à celles des automates finis, sauf pour l'étoile de Kleene. Pour réduire la taille de cette dernière, nous présentons les ACM_π introduisant la notion de transfert de mémoire entre les couches.

Enfin, nous avons conçu un algorithme de reconnaissance à la volée (*online*) pour les ATCM. C'est une contribution au domaine des automates temporisés, partagée avec [10], où l'on adapte le modèle des DBM pour le calcul des zones d'horloges dans le cadre de la reconnaissance. Cet algorithme ainsi que le modèle d'automate et le langage de spécification ont servi à l'implémentation d'un prototype. Il a été utilisé dans le cadre de la détection de motifs dans des flots de liens issus de données réelles. Les premiers résultats d'expérimentation ont été présentés dans ce manuscrit et semblent prometteurs.

Perspectives

Les travaux présentés dans cette thèse sont à l'intersection de plusieurs domaines, ainsi plusieurs axes de continuation sont disponibles. D'un point de vue pratique, plusieurs de ces travaux n'ont pour le moment pas été publiés et des articles sont en cours de rédaction, en particulier nos travaux dans le domaine des automates à mémoire.

Dans le domaine de l'analyse des systèmes complexes, même si nous avons pu détecter certains motifs emblématiques sur les flots de liens, de plus amples expéri-

mentations sont encore nécessaires afin d'étayer nos résultats et démontrer le vrai potentiel de notre approche. Dans nos projets à (très) court terme, nous pensons modéliser des motifs classiques des systèmes complexes, autres que les chemins et triangles, par exemple l'étoile et les arbres (causalité en cascade). Nous souhaiterions également effectuer des *benchmarks* par rapport aux outils et algorithmes existants. Notre approche générique ne pourra probablement pas obtenir des performances supérieures aux outils spécialisés, mais nous avons de nombreuses pistes d'améliorations, autant pour le formalisme de reconnaissance que pour le prototype. Au niveau des automates, nous appliquons déjà une heuristique sur la structure de l'automate qui permet de déterminer à l'avance si des contraintes de temps sont violées, avec l'ajout d'invariants dans les états. Cette heuristique est actuellement uniquement applicable avec l'opérateur dérivé $\langle e \rangle_I$, aussi ne l'avons-nous pas détaillée dans cette thèse mais seulement présentée dans [19]. Nous pensons qu'elle peut s'étendre, au moins partiellement, aux opérateurs $\overset{c}{\triangledown}$ et $\overset{c}{\ddagger}_I$ lors de la construction des automates. L'implémentation actuelle du prototype peut être optimisée afin de grandement améliorer les calculs des états atteignables. D'un point de vue technique, il est possible de paralléliser les calculs des états atteignables, accélérer la comparaison des contextes mémoire avec des techniques de hachage ou l'utilisation de BDD...

Plusieurs problématiques ont été abordées dans la théorie des automates, autour du modèle des ACM. Grâce au positionnement des ACM par rapport aux autres modèles d'automates à mémoire, nous avons pu déduire la décidabilité de plusieurs problèmes. Entre autres, nous avons déduit des estimations des classes de complexité des problèmes du langage vide et de la reconnaissance. Cependant, il nous reste à déterminer la classe de complexité exacte à laquelle appartiennent ces problèmes, et tout particulièrement celui de la reconnaissance, central à notre sujet. Nous disposons de plusieurs pistes en s'appuyant sur les recherches similaires dans les HRA [46] et les FMA [55]. L'autre problématique qui a été abordée est la taille des automates et particulièrement des constructions d'automates. Elle a mené à l'élaboration des ACM_π, qui n'ont été que peu étudiés dans cette thèse mais qui servent de base à notre algorithme effectif. Aussi, il pourrait être intéressant d'approfondir les travaux sur cette variante afin de déterminer son expressivité et la positionner clairement par rapport aux ACM et aux HRA. Une problématique plus ambitieuse que l'on souhaite aborder est celle de la minimisation d'automate, afin de développer une méthode générale, à la place d'optimiser individuellement les constructions. Les ν-automates qui sont très similaires aux ACM proposent une méthode de minimisation de la mémoire [31] qui pourrait être particulièrement intéressante à adapter.

Dans le domaine de la théorie des langages, nous avons développé le formalisme des ETCM qui est un langage qui fait preuve d'une grande expressivité et sur

lequel nous avons réussi à dériver plusieurs opérateurs assez simplement, dont les opérateurs rationnels classiques. Cependant, c'est un langage qui nous semble manquer d'abstraction, particulièrement pour ce qui est des langages de données, où la manipulation de la mémoire et les couches apparaissent explicitement dans la syntaxe. C'est une problématique commune aux autres langages de spécification sur les langages de données. Une première idée serait de définir une notion de portée, qui permettrait d'abstraire les manipulations mémoires.

Parmi les opérateurs dérivés, nous avons présenté le mélange et sa sémantique a été formellement définie directement sur les ATCM sous-jacents. La définition de cet opérateur pourrait mener à des contributions supplémentaires dans la théorie des langages, avec potentiellement un principe de *parsing derivatives* sur les ETCM, mais aussi la mise en avant d'une notion de mélange sur les langages de données. En effet, la notion de mélange présentée dans cette thèse est celle issue des langages rationnels. Cependant, dans la thèse nous avons défini sur les automates un mélange avec *effets de bord* qui pourrait alors représenter une généralisation de cette notion. Cette nouvelle opération manque encore d'une définition formelle dans la théorie des langages.

Pour terminer, la discipline de la supervision de systèmes a été mentionnée dans cette thèse, principalement dans l'état de l'art. Nos travaux et notre prototype sont assez similaires à ceux existant sur les *quantified event automata* (QEA) [12] et l'outil MarQ [73]. Une possibilité d'application de nos travaux serait le développement d'un outil semblable, mais avec un rapport plus fort à la théorie des automates, autant pour les contraintes de temps que le fonctionnement de la mémoire. Pour être capable de modéliser des systèmes ou des propriétés plus complexes, il pourrait alors être intéressant de développer des principes de reconnaissance pour des mécaniques de contraintes de temps plus sophistiquées, comme celles des *stopwatch automata* ou des *Updatable automata*, qui pourraient s'appliquer sans perte de décidabilité au problème de la reconnaissance, mais se posera alors le problème de la complexité.

Bibliographie

[1] Y. Abdeddaïm and O. Maler. Preemptive job-shop scheduling using stopwatch automata. In J.-P. Katoen and P. Stevens, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 113–126, Berlin, Heidelberg, 2002. Springer Berlin Heidelberg.

[2] R. Alur and D. L. Dill. A theory of timed automata. *Theoretical Computer Science*, 126(2) :183 – 235, 1994.

[3] R. Alur, T. A. Henzinger, and M. Y. Vardi. Parametric real-time reasoning. In *Proceedings of the Twenty-Fifth Annual ACM Symposium on Theory of Computing*, STOC '93, page 592–601, New York, NY, USA, 1993. Association for Computing Machinery.

[4] É. André, I. Hasuo, and M. Waga. Offline timed pattern matching under uncertainty. In *2018 23rd International Conference on Engineering of Complex Computer Systems (ICECCS)*, pages 10–20, 2018.

[5] R. Angles and C. Gutierrez. Survey of graph database models. *ACM Comput. Surv.*, 40(1), Feb. 2008.

[6] E. Asarin. Challenges in Timed Languages : From Applied Theory to Basic Theory. *Bulletin- European Association for Theoretical Computer Science*, 83 :106–120, June 2004. The Concurrency Column, by Luca Aceto. Partially based on the invited talk at FORMATS'03 workshop.

[7] E. Asarin, P. Caspi, and O. Maler. Timed regular expressions. *Journal of the ACM*, 49(2) :172–206, 2002.

[8] E. Asarin and C. Dima. Balanced timed regular expressions. *Electronic Notes in Theoretical Computer Science*, 68(5) :16 – 33, 2003. MTCS '02, Models for Time-Critical Systems (CONCUR 2002 Satellite Workshop).

[9] C. Baier and J.-P. Katoen. *Principles of Model Checking (Representation and Mind Series)*. The MIT Press, 2008.

[10] A. Bakhirkin, T. Ferrère, D. Nickovic, O. Maler, and E. Asarin. Online Timed Pattern Matching using Automata. In *16th International Conference on Formal Modelling and Analysis of Timed Systems (FORMATS)*, Beijing, China, Sept. 2018.

[11] P. Barceló, L. Libkin, A. W. Lin, and P. T. Wood. Expressive languages for path queries over graph-structured data. *ACM Trans. Database Syst.*, 37(4), Dec. 2012.

[12] H. Barringer, Y. Falcone, K. Havelund, G. Reger, and D. Rydeheard. Quantified event automata : Towards expressive and efficient runtime monitors. In D. Giannakopoulou and D. Méry, editors, *FM 2012 : Formal Methods*, pages 68–84, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[13] E. Bartocci, J. Deshmukh, A. Donzé, G. Fainekos, O. Maler, D. Ničković, and S. Sankaranarayanan. *Specification-based monitoring of cyber-physical systems : A survey on theory, tools and applications*, pages 135–175. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics). Springer Verlag, 2018.

[14] D. Basin, F. Klaedtke, and E. Zălinescu. Algorithms for monitoring real-time properties. In S. Khurshid and K. Sen, editors, *Runtime Verification*, pages 260–275, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[15] G. Behrmann, A. David, and K. Larsen. A tutorial on Uppaal 4 . 0. 2006.

[16] J. Bengtsson and W. Yi. *Timed Automata : Semantics, Algorithms and Tools*, pages 87–124. Springer Berlin Heidelberg, Berlin, Heidelberg, 2004.

[17] B. Bérard, P. Gastin, and A. Petit. On the power of non-observable actions in timed automata. In C. Puech and R. Reischuk, editors, *STACS 96*, pages 255–268, Berlin, Heidelberg, 1996. Springer Berlin Heidelberg.

[18] C. Bertrand, H. Klaudel, M. Latapy, and F. Peschanski. Pattern matching in link streams : A token-based approach. In *Petri Nets*, volume 10877 of *Lecture Notes in Computer Science*, pages 227–247. Springer, 2018.

[19] C. Bertrand, F. Peschanski, H. Klaudel, and M. Latapy. Pattern matching in link streams : Timed-automata with finite memory. *Sci. Ann. Comp. Sci.*, 28(2) :161–198, 2018.

[20] H. Björklund and T. Schwentick. On notions of regularity for data languages. *Theor. Comput. Sci.*, 411(4-5) :702–715, 2010.

[21] H. Björklund and T. Schwentick. *Class-Memory Automata Revisited*, pages 201–215. Springer International Publishing, Cham, 2017.

[22] B. Blonder, T. W. Wey, A. Dornhaus, R. James, and A. Sih. Temporal dynamics and network analysis. *Methods in Ecology and Evolution*, 3(6) :958–972, 2012.

[23] M. Bojanczyk, C. David, A. Muscholl, T. Schwentick, and L. Segoufin. Two-variable logic on data words. *ACM Trans. Comput. Log.*, 12(4) :27 :1–27 :26, 2011.

[24] P. Bouyer, C. Dufourd, E. Fleury, and A. Petit. Updatable timed automata. *Theoretical Computer Science*, 321(2) :291 – 345, 2004.

[25] P. Bouyer and A. Petit. Decomposition and composition of timed automata. In J. Wiedermann, P. van Emde Boas, and M. Nielsen, editors, *Automata, Languages and Programming*, pages 210–219, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[26] R. S. Boyer and J. S. Moore. A fast string searching algorithm. *Commun. ACM*, 20(10) :762–772, Oct. 1977.

[27] J. A. Brzozowski. Derivatives of regular expressions. *J. ACM*, 11(4) :481–494, Oct. 1964.

[28] J. A. Brzozowski and E. J. McCluskey. Signal flow graph techniques for sequential circuit state diagrams. *IEEE Transactions on Electronic Computers*, EC-12(2) :67–76, 1963.

[29] A. Casteigts, P. Flocchini, W. Quattrociocchi, and N. Santoro. Time-varying graphs and dynamic networks. *CoRR*, abs/1012.0009, 2010.

[30] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[31] A. Deharbe. *Analyse de ressources pour les systèmes concurrents dynamiques*. Thèses, Université Pierre et Marie Curie - Paris VI, 2016.

[32] A. Deharbe and F. Peschanski. The omniscient garbage collector : A resource analysis framework. In *ACSD 2014*. IEEE Computer Society, 2014.

[33] S. Demri and R. Lazić. LTL with the freeze quantifier and register automata. *ACM Trans. Comput. Logic*, 10(3), Apr. 2009.

[34] J. V. Deshmukh, A. Donzé, S. Ghosh, X. Jin, G. Juniwal, and S. A. Seshia. Robust online monitoring of signal temporal logic, 2015.

[35] D. L. Dill. Timing assumptions and verification of finite-state concurrent systems. In J. Sifakis, editor, *Automatic Verification Methods for Finite State Systems*, pages 197–212, Berlin, Heidelberg, 1990. Springer Berlin Heidelberg.

[36] C. Dima. Timed shuffle expressions. In M. Abadi and L. de Alfaro, editors, *CONCUR 2005 – Concurrency Theory*, pages 95–109, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[37] J. Dong, S. Qin, W. Yi, J. Sun, and P. Hao. Timed automata patterns. *IEEE Transactions on Software Engineering*, 34(06) :844–859, nov 2008.

[38] A. Donzé and O. Maler. Robust satisfaction of temporal logic over real-valued signals. In K. Chatterjee and T. A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, pages 92–106, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[39] D. A. Easley and J. M. Kleinberg. *Networks, Crowds, and Markets - Reasoning About a Highly Connected World*. Cambridge University Press, 2010.

[40] D. FIGUEIRA, P. HOFMAN, and S. LASOTA. Relating timed and register automata. *Mathematical Structures in Computer Science*, 26(6) :993–1021, Dec 2014.

[41] O. Finkel. Undecidable problems about timed automata. In E. Asarin and P. Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, pages 187–199, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[42] R. Fontugne, P. Borgnat, P. Abry, and K. Fukuda. MAWILab : Combining Diverse Anomaly Detectors for Automated Anomaly Labeling and Performance Benchmarking. In *ACM CoNEXT '10*, 2010.

[43] F. Franek, C. G. Jennings, and W. F. Smyth. A simple fast hybrid pattern-matching algorithm. In A. Apostolico, M. Crochemore, and K. Park, editors, *Combinatorial Pattern Matching*, pages 288–297, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[44] X. Gao, Q. Zheng, D. A. Vega-Oliveros, L. Anghinoni, and L. Zhao. Temporal network pattern identification by community modelling. *Scientific Reports*, 10, 2020.

[45] R. Grigore, D. Distefano, R. L. Petersen, and N. Tzevelekos. Runtime verification based on register automata. In N. Piterman and S. A. Smolka, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 260–276, Berlin, Heidelberg, 2013. Springer Berlin Heidelberg.

[46] R. Grigore and N. Tzevelekos. History-register automata. *Logical Methods in Computer Science*, 12(1), 2016.

[47] O. Grumberg, O. Kupferman, and S. Sheinvald. Variable automata over infinite alphabets. In A.-H. Dediu, H. Fernau, and C. Martín-Vide, editors, *Language and Automata Theory and Applications*, pages 561–572, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[48] L. Gulyás, G. Kampis, and R. O. Legendi. Elementary models of dynamic networks. *The European Physical Journal Special Topics*, 222(6) :1311–1333, 2013.

[49] S. Gurukar, S. Ranu, and B. Ravindran. Commit : A scalable approach to mining communication motifs from dynamic networks. In *Proceedings of the 2015 ACM SIGMOD International Conference on Management of Data*, SIGMOD '15, page 475–489, New York, NY, USA, 2015. Association for Computing Machinery.

[50] T. A. Henzinger, P. W. Kopke, A. Puri, and P. Varaiya. What's decidable about hybrid automata ? *Journal of Computer and System Sciences*, 57(1) :94 – 124, 1998.

[51] P. Herrmann. Renaming is necessary in timed regular expressions. In C. P. Rangan, V. Raman, and R. Ramanujam, editors, *Foundations of Software Technology and Theoretical Computer Science*, pages 47–59, Berlin, Heidelberg, 1999. Springer Berlin Heidelberg.

[52] H.-M. Ho, J. Ouaknine, and J. Worrell. Online monitoring of metric temporal logic. In B. Bonakdarpour and S. A. Smolka, editors, *Runtime Verification*, pages 178–192, Cham, 2014. Springer International Publishing.

[53] J. E. Hopcroft, R. Motwani, and J. D. Ullman. *Introduction to automata theory, languages, and computation, 3rd Edition.* Pearson international edition. Addison-Wesley, 2007.

[54] M. Iliofotou, M. Faloutsos, and M. Mitzenmacher. Exploiting dynamicity in graph-based traffic analysis : Techniques and applications. In *Proceedings of the 5th International Conference on Emerging Networking Experiments and Technologies*, CoNEXT '09, page 241–252, New York, NY, USA, 2009. Association for Computing Machinery.

[55] M. Kaminski and N. Francez. Finite-memory automata. *Theoritical Computer Science*, 134 :329–363, 1994.

[56] M. Kaminski and D. Zeitlin. Finite-memory automata with non-deterministic reassignment. *Int. J. Found. Comput. Sci.*, 21(5) :741–760, 2010.

[57] A. Kara. *Logics on data words : Expressivity, satisfiability, model checking.* PhD thesis, Technical University of Dortmund, Germany, 2016.

[58] L. Kovanen, M. Karsai, K. Kaski, J. Kertész, and J. Saramäki. Temporal motifs in time-dependent networks Temporal motifs in time-dependent networks Temporal motifs in time-dependent networks. *J. Stat. Mech*, 2011.

[59] R. Koymans. Specifying real-time properties with metric temporal logic. *Real-Time Syst.*, 2(4) :255–299, Oct. 1990.

[60] G. Krings, M. Karsai, S. Bernhardsson, V. D. Blondel, and J. Saramäki. Effects of time window size and placement on the structure of an aggregated communication network. *EPJ Data Science*, 1(1), May 2012.

[61] M. Latapy. Main-memory triangle computations for very large (sparse (power-law)) graphs. *Theoretical Computer Science*, 407(1) :458 – 473, 2008.

[62] M. Latapy, T. Viard, and C. Magnien. Stream graphs and link streams for the modeling of interactions over time. *Soc. Netw. Anal. Min.*, 8(1) :61 :1–61 :29, 2018.

[63] L. Libkin, T. Tan, and D. Vrgoč. Regular expressions for data words. *Journal of Computer and System Sciences*, 81(7) :1278 – 1297, 2015.

[64] L. Libkin and D. Vrgoč. Regular path queries on graphs with data. In *Proceedings of the 15th International Conference on Database Theory*, ICDT '12,

page 74–85, New York, NY, USA, 2012. Association for Computing Machinery.

[65] P. Liu, A. R. Benson, and M. Charikar. Sampling methods for counting temporal motifs. In *Proceedings of the Twelfth ACM International Conference on Web Search and Data Mining*, WSDM '19, page 294–302, New York, NY, USA, 2019. Association for Computing Machinery.

[66] P. Mackey, K. Porterfield, E. Fitzhenry, S. Choudhury, and G. Chin. A chronological edge-driven approach to temporal subgraph isomorphism. In *2018 IEEE International Conference on Big Data (Big Data)*, pages 3972–3979, 2018.

[67] O. Maler, D. Nickovic, and A. Pnueli. From MITL to timed automata. In E. Asarin and P. Bouyer, editors, *Formal Modeling and Analysis of Timed Systems*, pages 274–289, Berlin, Heidelberg, 2006. Springer Berlin Heidelberg.

[68] R. Milo, S. Shen-Orr, S. Itzkovitz, N. Kashtan, D. Chklovskii, and U. Alon. Network motifs : Simple building blocks of complex networks. *Science*, 298(5594) :824–827, 2002.

[69] F. Neven, T. Schwentick, and V. Vianu. Towards regular languages over infinite alphabets. In J. Sgall, A. Pultr, and P. Kolman, editors, *Mathematical Foundations of Computer Science 2001*, pages 560–572, Berlin, Heidelberg, 2001. Springer Berlin Heidelberg.

[70] F. Neven, T. Schwentick, and V. Vianu. Finite state machines for strings over infinite alphabets. *ACM Trans. Comput. Logic*, 5(3) :403–435, July 2004.

[71] D. Ničković and N. Piterman. From Mtl to deterministic timed automata. In K. Chatterjee and T. A. Henzinger, editors, *Formal Modeling and Analysis of Timed Systems*, pages 152–167, Berlin, Heidelberg, 2010. Springer Berlin Heidelberg.

[72] A. Paranjape, A. R. Benson, and J. Leskovec. Motifs in temporal networks. In *Proceedings of the Tenth ACM International Conference on Web Search and Data Mining*, WSDM '17, pages 601–610. ACM, 2017.

[73] G. Reger, H. C. Cruz, and D. Rydeheard. MarQ : Monitoring at runtime with QEA. In C. Baier and C. Tinelli, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 596–610, Berlin, Heidelberg, 2015. Springer Berlin Heidelberg.

[74] P. Ribeiro, N. Perra, and A. Baronchelli. Quantifying the effect of temporal resolution on time-varying networks. *Scientific Reports*, 3(3006), 2013.

[75] H. Sakamoto and D. Ikeda. Intractability of decision problems for finite-memory automata. *Theoretical Computer Science*, 231(2) :297 – 308, 2000.

[76] M. Sulzmann and P. Thiemann. Derivatives for regular shuffle expressions. In A.-H. Dediu, E. Formenti, C. Martín-Vide, and B. Truthe, editors, *Language and Automata Theory and Applications*, pages 275–286, Cham, 2015. Springer International Publishing.

[77] T. Takaguchi, Y. Yano, and Y. Yoshida. Coverage centralities for temporal networks. *The European Physical Journal B*, 89(2) :1–11, 2016. Electronic supplementary material Supplementary Online Material 10.1140/epjb/e2016-60498-7.

[78] J. Tang, M. Musolesi, C. Mascolo, and V. Latora. Characterising temporal distance and reachability in mobile and online social networks. *SIGCOMM Comput. Commun. Rev.*, 40(1) :118–124, Jan. 2010.

[79] N. Tzevelekos. Fresh-register automata. In *POPL*, pages 295–306, 2011.

[80] D. Ulus. Montre : A tool for monitoring timed regular expressions. *Lecture Notes in Computer Science*, page 329–335, 2017.

[81] D. Ulus, T. Ferrère, E. Asarin, and O. Maler. Timed Pattern Matching. *Formal Modeling and Analysis of Timed Systems*, 8711, 2014.

[82] D. Ulus, O. Maler, E. Asarin, and T. Ferrère. Online Timed Pattern Matching Using Derivatives. *LNCS*, 9636 :7–8, 2016.

[83] T. Viard, R. Fournier-S'niehotta, C. Magnien, and M. Latapy. Discovering patterns of interest in IP traffic using cliques in bipartite link streams. *CoRR*, abs/1710.07107, 2017.

[84] T. Viard, M. Latapy, and C. Magnien. Computing maximal cliques in link streams, 2016.

[85] M. Waga, T. Akazaki, and I. Hasuo. A Boyer-Moore type algorithm for timed pattern matching. *Formal Modeling and Analysis of Timed Systems*, page 121–139, 2016.

[86] M. Waga and É. André. Online parametric timed pattern matching with automata-based skipping. *NASA Formal Methods*, page 371–389, 2019.

[87] M. Waga, É. André, and I. Hasuo. Symbolic monitoring against specifications parametric in time and data. In *CAV 2019,*, pages 520–539, 2019.

[88] S. Wasserman, K. Faust, et al. *Social network analysis : Methods and applications*, volume 8. Cambridge university press, 1994.

[89] B. W. Watson and R. E. Watson. A Boyer–Moore-style algorithm for regular expression pattern matching. *Sci. Comput. Program.*, 48(2–3) :99–117, Aug. 2003.

[90] K. Wehmuth, A. Ziviani, and E. Fleury. A unifying model for representing time-varying graphs. In *2015 IEEE International Conference on Data Science and Advanced Analytics (DSAA)*, pages 1–10, 2015.

[91] J. Whitbeck, M. D. de Amorim, V. Conan, and J.-L. Guillaume. Temporal reachability graphs, 2012.

université PARIS-SACLAY
ÉCOLE DOCTORALE
Sciences et technologies de l'information et de la communication (STIC)

Titre : Reconnaissance de motifs dynamiques par automates temporisés à mémoire

Mots clés : Automate à mémoire, Automate temporisé, Expression régulière,Vérification dynamique, Système complexe

Résumé : L'utilisation toujours plus importante de l'informatique et d'internet mène à une génération toujours plus importante de donnés et de communications. Ces données peuvent être par exemple des historiques de communication dans des réseaux sociaux ou encore des traces du trafic internet. De tels historiques de communications sont une forme de graphe dynamique formalisable par des flots de liens. De nombreux travaux s'articulent autour de la supervision et l'analyse de ces systèmes afin de détecter l'apparition de certains phénomènes ou scénarios spécifiques. Par exemple, dans le cadre de la sécurité, on souhaite détecter des tentatives d'intrusions concertées, tel des DDOS. Une première problématique est la création d'un langage général et normalisé de spécification de tel phénomène, car peu de langages existent et ils sont souvent spécifiques à certaines catégories de scénarios par soucis de performance. La seconde problématique principale est l'implémentation d'un prototype d'outil de reconnaissance pour détecter ces phénomènes dans des jeux de données issues de situation réelles.

Title : Matching of dynamic patterns with timed memory automata

Keywords : Memory Automata, Timed Automata, Regular Expression, Runtime verification, Complex network

Abstract : The globalized and increasing use of computers and the Internet have for consequences an always increasing quantity of data and online communications. Some examples of this data can be logs of social networks messages of Internet routers usages. Such communication logs are similar to dynamic graph formalized as link streams. The monitoring and analysis of this kind of systems are quite common, often looking for occurrences of specific communication patterns. For example, a common pattern in Security is the detection of concerted attacks toward a same target, such as a DDOS. One of the problems addressed in this thesis is the creation of a normalized specification language for such patterns in the networks. Not many similar languages exist because they are often specific to some kind of patterns due to performance issues. The other main issues applied during this thesis is the implementation of a prototype tool for the detection of this patterns in real life link streams.

Université Paris-Saclay
Espace Technologique / Immeuble Discovery
Route de l'Orme aux Merisiers RD 128 / 91190 Saint-Aubin, France