-Pomset pour la modélisation et la vérification de systèmes parallèles

Mouhamadou Tafsir Sakho

École Doctorale
MATHEMATIQUES, INFORMATIQUE,
PHYSIQUE THEORIQUE ET INGENIERIE
DES SYSTEMES
Laboratoire d'Informatique Fondamentale d'Orléans

THÈSE

présentée par:

Mouhamadou Tafsir SAKHO

pour obtenir le grade de : **Docteur de l'Université d'Orléans et de l'université UPOA**

Discipline: **Mathématiques-Informatique**

Γ-pomset pour la modélisation et la vérification de systèmes parallèles

Soutenue le 17 décembre 2014

THÈSE codirigée par :

M Jean Michel **Couvreur** Professeur, Université d'Orléans
et Hamet **Seydi** Professeur, Université Polytechnique de l'Ouest Africain

RAPPORTEURS :

M. Serge **Haddad** Professeur, ENS de Cachan
M. Gane Samb **Lo** Professeur, Université G.B. de Saint-Louis

---

JURY :

M. Jean Michel **Couvreur**	Professeur, Université d'Orléans
M. Alain **Griffault**	Maître de Conférences, Université de Bordeaux
M. Serge **Haddad**	Professeur, ENS de Cachan
M. Gane Samb **Lo**	Professeur, Université G.B. de Saint-Louis
M. Fréderic **Loulergue**	Professeur, Université d'Orléans
M. Denis **Poitrenaud**	Maître de Conférences, Université Paris Descartes
M. Hamet **Seydi**	Professeur, UPOA Dakar

Vérité en deçà des Pyrénées, erreur au-delà.

(Blaise Pascal)

À mes grands pères **El Hadj Ibrahima Sakho** (20 dec. 1911 - 13 aout 1994) et **El Hadj Gallo Diagne** (02 oct. 1918 - 21 juin 2008)
Et à tous ceux qui sont assoiffés de connaissances.

Remerciements

Je ne trouve pas de mots assez forts pour exprimer mon sentiment de reconnaissance et de profonde gratitude à l'endroit de Jean Michel Couvreur, mon encadrant et directeur de thèse. Les efforts qu'il n'a cessé de faire pour mon encadrement actif, ses encouragements dans mes moments de doute, ses conseils et son accompagnement dans la bonne humeur resteront à jamais gravés dans ma mémoire.

M. Serge Haddad de l'ENS Cachan et M. Gane Samb Lo de l'Université Gaston Berger de Saint-Louis m'ont fait l'immense honneur d'être les rapporteurs de ma thèse. Je les remercie d'avoir relu et rapporté ce manuscrit avec patience et beaucoup d'attentions. Je suis également très honoré de la présence dans mon jury de M. Alain Griffault de l'université de Bordeaux, de M. Fréderic Loulergue de l'université d'Orléans et Denis Poitrenaud de l'université Paris Descartes comme examinateurs. Je leur en suis très reconnaissant.

M. Hamet Seydi, à titre officiel, a assuré la co-direction de ma thèse pour le compte de l'Université Polytechnique de l'Ouest Africain. Nos discussions ont été enrichissantes pour moi, et ses conseils et aides, d'une manière générale, m'ont été d'un grand intérêt. Qu'il trouve ici l'expression de mes remerciements les plus sincères.

Je tiens à remercier très chaleureusement Yohan Boichut, dont la collaboration et l'aide ont été précieuses et décisives pour l'aboutissement de mon travail. Je lui en suis très reconnaissant.

Je voudrais exprimer mes remerciements à toute l'équipe du Laboratoire d'Informatique Fondamentale d'Orléans et du département informatique de l'Université d'Orléans. Je citerai à cet effet Jérôme Durand-Lose, directeur du LIFO, Catherine Julié-Bonnet directrice du département, Ioan Todinca ancien directeur de l'école doctorale MIPTIS pour leurs conseils et leurs assistances.

Mes sincères remerciements vont également à tout le personnel de l'Université Polytechnique de l'Ouest Africain à commencer par le président El Hadj Ibrahima Sall pour son soutien sans faille, ainsi que Ndeyse Awa Sall, Moustapha Ndiaye, Marianne Dioh sans oublier Boubacar Cissé Fall. Qu'ils trouvent ici l'expression de ma très profonde gratitude.

Je garderai de ces années de doctorat un bon souvenir des moments passés avec mes collègues *masochistes* (thésards) du laboratoire. Je pense particulièrement à ceux avec qui j'ai eu à partager mon bureau: Vincent Levorrato, Thomas Pinsard, Joshua Amavi.

Je suis extrêmement reconnaissant à mes amis et frères d'Orléans qui m'ont toujours témoigné d'un attachement et d'une affection sincère et inébranlable. J'adresse une mention

spéciale à Pathé Diagne, mon *monsieur latex*. Je pense aussi à la famille Dia ainsi qu'à toute ma famille de Saint-Pierre-du-Perray.

Last but not least, je ne remercierai jamais suffisamment mes parents Mouhamadou Habib Sakho et Bineta Diagne, sans qui je ne serais rien, et qui n'ont ménagé aucun effort pour nous donner, Alioune et moi, la meilleure éducation. Je leur souhaite tout le bonheur du monde. Pour son amour et son soutien indéfectible, un grand merci à ma femme qui m'apporte beaucoup de bonheur depuis plusieurs mois déjà. J'associe enfin à ces remerciements mes frères, soeurs, cousins, cousines, tantes et oncles et tous ceux qui de près ou de loin ont contribué à ma réussite. Nul besoin de rappeler ici tout le bien que je pense de ce beau monde.

Table des matières

Liste des figures

Chapitre 1

Introduction

Sommaire

1.1 Contexte de la thèse

Un système parallèle (ou système concurrent) est un système composé d'entités fonctionnant en même temps et susceptibles de communiquer entre elles durant leur fonctionnement. En informatique, on assimile cela souvent à un programme composé de plusieurs processus (ou tâches) s'exécutant en même temps. Ces systèmes ont connu une évolution rapide ces dernières décennies grâce essentiellement aux progrès technologiques du matériel, et aussi à l'évolution qualitative des langages de programmation.

Cette multiplicité de tâches ou activités parallèles pour un système apporte des complications qui n'existent pas dans les systèmes où une seule activité s'exécute (systèmes séquentiels). Ces systèmes peuvent en particulier être composés d'un très grand nombre de tâches devant s'exécuter dans des conditions et sous des contraintes différentes. Ainsi une caractéristique importante de ces systèmes est leur propension à avoir un grand nombre de comportements possibles. On doit résoudre aussi le problème de l'accès concurrent aux ressources que doivent se partager les tâches.

Toutes ces sources de complexité sont l'objet de nombreux travaux théoriques et pratiques utilisant des outils de l'informatique théorique telle que les "méthodes formelles" dont le but est principalement de permettre la représentation rigoureuse des systèmes dans un formalisme satisfaisant permettant d'établir de manière automatique des preuves de leur propriétés.

L'avènement du parallélisme soulève dans ce cadre plusieurs questions notamment:

- Comment modéliser ou représenter l'exécution, la communication et la synchronisation entre systèmes concurrents ou à l'intérieur d'un système ?

- Comment vérifier le bon fonctionnement de ces systèmes après leur modélisation ?

Quand il s'agit d'un système séquentiel, les comportements sont généralement modélisés par des mots finis ou infinis partant d'un alphabet quelconque. Chaque lettre représente une action ou les valeurs des propriétés élémentaires de l'état d'un système.

De nombreuses méthodes de modélisation de la concurrence ont été proposées depuis les années 60, par exemple, les réseaux de Petri [Pet62], les automates communicants [BZ83], les algèbres de processus [BHR84] etc.

Les comportements d'un système parallèle peuvent aussi être décrits par des "modèles d'ordres partiels". Gisher [Gis85] et Pratt [Pra86] ont notamment introduit les ensembles partiellement ordonnés étiquetés (pomsets). Ces structures mathématiques constituent l'un des formalismes idéaux à employer dans le contexte des systèmes parallèles. Ce formalisme permet d'une part de modéliser graphiquement, et donc de manière intuitive, des interactions complexes entre différentes entités et constitue d'autre part, une représentation très compacte, comparée aux autres modèles qui décrivent des entrelacements d'exécutions séquentielles. De plus il constitue une fusion entre la théorie des langages formels où les mots, lettres et alphabets sont les maîtres-mots et la théorie mathématique des ordres partiels. Enfin, il permet de mieux considérer les questions liées à la "causalité" auxquelles certains modèles classiques peuvent difficilement répondre.

La vérification est une analyse effectuée à partir d'un modèle du système. Formellement cette analyse utilise généralement des logiques telles que MSO, LTL ou CTL, et vérifie si des propriétés du systèmes sont respectées. L'intérêt de ce type d'analyse est de pouvoir détecter, avant même son exécution, des problèmes éventuels dans la spécification d'un système. Une propriété est spécifiée par une formule logique et elle représente un ensemble de comportements qu'un système étudié doit avoir.

Malgré tous les avantages qu'offre la théorie des pomsets d'un point de vue de la modélisation, la vérification de systèmes décrits par ordres partiels est connue pour être difficile. Les modèles considérés dans cette thèse s'appuient sur les pomsets. Nous nous intéressons aux problèmes de faisabilité de la vérification de systèmes parallèles par ordre partiel. Nous utilisons la logique MSO.

1.2 Problématiques de la thèse

On peut considérer deux types de problèmes dans la vérification:

1. Le problème de vérification pour une formule et un modèle donné qui consiste à décider si la formule est vraie pour tout comportement du système représenté par le modèle,

2. Le problème de satisfaisabilité consiste à décider si un comportement satisfaisant la formule existe.

Les théories sur les langages réguliers de mots et d'arbres (finis et infinis)([BL69]) sont la pierre angulaire des techniques de vérification efficaces sur des systèmes d'état finis [CES83, Hol97]. En effet, considérant le cas des mots finis, Büchi a établi la relation suivante: *"un langage est régulier si et seulement si il est MSO définissable"*. Conséquemment, des systèmes peuvent être représentés par des automates d'un mot et des propriétés définies dans MSO peuvent aussi être représentées par des automates d'un mot. Plus précisément, un système

S, de domaine fini, peut être encodé par un automate $\mathcal{A}_S$ et la négation d'une propriété ϕ peut aussi être encodée par un automate $\mathcal{A}_{\neg\phi}$. Calculer l'intersection des deux langages générés par $\mathcal{A}_S$ et $\mathcal{A}_{\neg\phi}$ permet la décision de la satisfaction de ϕ par S.

Concernant les systèmes concurrents, des théories diverses et variées abordent le problème de satisfaction mais sont limitées par des questions d'indécidabilité. Ainsi, il est improbable que des théories pourtant attractives comme celle sur les langages réguliers géreront la vérification des systèmes concurrents dans sa totalité.

Étant donné une formule MSO avec des ordres partiels (MSO_{Pomset}) ϕ, le problème de satisfaction consiste à trouver un pomset satisfaisant la formule ϕ. Le problème de satisfaction est indécidable [AP99]. Dans [GK05], les auteurs gèrent le problème de
satisfaction de formule MSO sur les traces de quelques uns des ordres partiels qui sont hors de portée de ceux qui peuvent être générés à partir des traces de Mazurkiewicz. Conséquemment, même si pour une formule MSO_{Pomset} donnée, il existe une solution, les traces de Mazurkiewicz n'assurent pas sa détection. Des connexions ont aussi été établies entre des modèles de concurrence (MSC, séries-parallèles, automates cellulaires asynchrones) et des logiques [CG95, DGK00, LW00, Mor02, HMK$^+$05]. Malheureusement, pour le moment, le problème de satisfaction des formules MSO avec des ordres partiels ne peut pas être géré (ou partiellement géré pour une classe donnée de pomsets) sur de tels modèles. De plus les études qui ont été effectuées sur ces modèles en terme de vérification ont très peu été suivi d'implémentation dans des outils de vérification.

Un autre moyen d'aborder le problème de satisfaction est de restreindre l'expressivité de MSO_{Pomset} de telle sorte que le problème devienne décidable comme dans [AMP98].

Ici, nous abordons le problème à partir d'un autre point de vue. Nous proposons un modèle qui rend possible l'exploration des espaces de pomsets. Les pomsets sont encodés par des mots, que nous appellerons Γ-Pomsets , et qui utilisent un alphabet spécifique Γ pour la description des dépendances causales. Pour tout pomset, on peut définir un alphabet Γ de telle sorte que ce pomset puisse être encodé par un Γ-Pomset . Cependant, il n'existe pas un alphabet Γ couvrant n'importe quel pomset. Conséquemment, nous pouvons semi-décider le problème de satisfaction en augmentant Γ autant que besoin. Notre approche est simple puisque nous explorons l'ensemble des pomsets étape par étape. Nous ne sommes pas restreints aux pomsets induits par les traces de Mazurkievicz comme dans [GK05] ou par des classes particulières de pomsets comme dans [DGK00].

Notre contribution s'articule donc principalement autour de deux aspects: un nouveau modèle de description du parallélisme (Γ-Pomset) et un algorithme de vérification de formule d'ordre partiel.

1.3 Plan de la thèse

Ce document est composé de 7 chapitres et la suite est structurée de la manière suivante:

Le chapitre 2 donne des préliminaires d'une part sur la théorie des langages et automates et d'autre part sur les logiques notamment la logique MSO. Nous présentons aussi dans ce chapitre la théorie des pomsets.

Le chapitre 3 est consacré à l'étude des modèles existants qui ont une sémantique ordre partiel et qui traite de la logique. Nous y présenterons notamment les traces de Mazurkiewicz, les pomsets Série-parallèle, les Messages Sequence Chart et les Automates Cellulaires Asynchrones de pomsets.

Dans le chapitre 4, nous présentons formellement notre nouveau modèle Γ-Pomset , décrivons ses relations avec la théorie des pomsets et donnons quelques exemples de modélisation.

Le chapitre 5 établit les relations entre notre modèle Γ-Pomset et les modèles classiques de la concurrence à sémantique ordre partiel afin de mieux s'apercevoir du pouvoir d'expression des Γ-Pomsets .

Le chapitre 6 présente la méthode de vérification avec les Γ-Pomset et donne les résultats sur la satisfaction de formule MSO. Il présente aussi quelques expérimentations sur l'outil Mona, partant de nos résultats.

Le chapitre 7 termine ce mémoire par une conclusion qui présente les apports de notre travail, ainsi que les perspectives de recherche envisagées.

Chapitre 2

Préliminaires

Sommaire

Dans ce chapitre, nous nous intéressons aux principales théories sur lesquelles se basent nos travaux. Nous donnons donc les définitions et notations essentielles qui nous serviront dans les chapitres ultérieurs. Ainsi, nous introduisons les notions de mots et langages, logique pour la vérification, avant de présenter la théorie des pomsets.

2.1 Mots, Langages et Automates

2.1.1 Alphabet et Mots

Un alphabet est un ensemble fini et non vide de caractères. Nous utilisons généralement la notation Σ pour désigner un alphabet. Les éléments de Σ sont appelés lettres, ou symboles. Un mot u sur un alphabet Σ est une suite de symboles, éventuellement vide, de Σ. Il peut s'écrire en $u = u_1.u_2 \dots u_n$ avec $u_i \in \Sigma$ pour tout $i \in [1..n]$. L'ensemble des mots finis de Σ est noté Σ^* et est aussi appelé algébriquement le *monoïde libre*. L'ensemble des mots infinis généré par Σ est noté Σ^ω. Nous posons $\Sigma^\infty = \Sigma^* \cup \Sigma^\omega$ pour désigner l'ensemble de tous les mots finis ou infinis de Σ.

Soit un mot $u \in \Sigma^*$, la longueur de u correspond au nombre de caractères qui le composent. On la note $|u|$. Ainsi pour le mot vide qui est noté ε, on a $|\varepsilon| = 0$. La notation $|u|_a$ renvoie au nombre d'occurrence du symbole a dans u. L'écriture u_i, quant à elle, désigne le symbole placé à la position i de u. Si une lettre u_i de u précède en terme de position une lettre u_j de u nous disons que u_i est plus petit que u_j et on note $u_i \leq u_j$. Il est donc facile de constater que la relation $\leq$ sur les symboles d'un mot est une relation d'ordre total, car étant réflexive, antisymétrique, transitive et deux symboles d'un mot sont toujours comparables par leur position.

Soient u et v deux mots finis d'un alphabet Σ. La *concaténation* $u.v$ dénote le mot obtenu en juxtaposant le mot v à la suite du mot u. On peut la noter plus simplement uv à la place de $u.v$. v est un *facteur* de u s'il existe des mots x et y de Σ^* tels que $u = xvy$. De plus, si $x = \varepsilon$ (resp. $y = \varepsilon$), on dit que v est un *préfixe* (resp. *suffixe*) de u.

2.1.2 Langages et régularité

En parlant de langage, nous passons de l'étude des mots individuels à celle des ensembles de chaînes finies et infinies. Les modèles de systèmes informatiques sont souvent caractérisés en termes de régularité dans la manière dont ils traitent différentes chaînes, donc il est important de comprendre d'abord les voies générales pour décrire et combiner les classes de chaînes.

Définition 2.1.1 (Langage) *Un langage L sur un alphabet Σ est une partie de Σ^*. Autrement dit, c'est un ensemble de mots finis sur Σ.*

Par exemple, $\{toto, bebe, abba, f\}$ est un langage de $\Sigma = \{a, b, ..., z\}$.

Les opérations sur les langages sont les opérations usuelles sur les ensembles: intersection, union, complémentaire, différence. On ajoute en plus la *concaténation des langages*. La concaténation de deux langages L et L' de Σ est le langage LL' défini comme suit:

$$LL' = \{uv \mid u \in L, v \in L\}$$

La concaténation d'un langage par lui même plusieurs fois nous pousse à utiliser des puissances pour leur définition. Ainsi on définit par récurrence:

$$L^0 = \{\varepsilon\} \; et \; \forall n \in \mathbb{N}, L^{n+1} = LL^n$$

On appelle fermeture (ou étoile) de Kleene du langage L, le langage :

$$L^* = \bigcup_{n \geq 0} L^n$$

.

Si F est une famille de langage de Σ, on dit que F est fermé (ou clos) pour une opération $\bullet$ si pour tout L_i et L_j de Σ^* : $L_i, L_j \in F \Rightarrow L_i \bullet L_j \in F$.

Le *complémentaire d'un langage* $L \subseteq \Sigma^*$, noté $\bar{L}$, désigne l'ensemble des mots de Σ^* qui n'appartiennent pas à L.

Définition 2.1.2 (Langage régulier) *REG(Σ^*) est le plus petit ensemble des parties de Σ^* contenant le langage vide et tous les langages réduits à un mot de longueur ≤ 1, et qui soit fermé pour la concaténation, l'union, et la fermeture de Kleene. Un langage régulier sur Σ est un élément de REG(Σ^*).*

Comme chaque partie finie de Σ^* est soit l'ensemble vide, soit une union finie de mots de Σ^*, et que ces mots sont soit le mot vide, soit la concaténation finie de lettres de Σ^*, on obtient une définition équivalente sous la forme d'expressions régulières.

Définition 2.1.3 (Expression régulière) *Une expression régulière (ou rationnelle) E sur Σ est un terme défini inductivement par:*

$$E := \emptyset \mid \varepsilon \mid a \mid E_1 E_2 \mid E_1 + E_2 \mid E^*$$

avec $a \in \Sigma$ et E_1 et E_2 des expressions rationnelles.

Une expression régulière sur Σ est interprétée comme un langage $L(E)$ défini inductivement par:

- $L(\emptyset) = \emptyset$;
- $L(\varepsilon) = \{\varepsilon\}$;
- $L(a) = a$ *pour tout* $a \in \Sigma$;
- $L(E_1 E_2) = L(E_1)L(E_2)$;
- $L(E_1 + E_2) = L(E_1) \cup L(E_2)$;
- $L(E^*) = L(E)^*$;

Un sous-ensemble $L \subseteq \Sigma^*$ est régulier s'il est le langage d'une expression régulière. De manière équivalente, L est régulier s'il peut être obtenu à partir d'une suite finie de concaténations, d'unions, et d'itérations de sous-ensembles de L.

Deux expressions régulières sont équivalentes si elles expriment le même langage.

Dans le domaine de la modélisation de systèmes que nous verrons plus tard, les langages tirent leurs intérêts du fait qu'ils expriment un ensemble d'exécutions possibles d'un système; chaque mot traduisant une exécution. Pour un informaticien, résoudre un problème peut consister à détecter si un mot appartient à un langage ou si tous les mots d'un langage ont une propriété donnée ou non. Pour cela, nous introduisons les notions d'automate et de reconnaissabilité.

2.1.3 Automate de mots finis

Le terme "théorie des automates" fut introduit en premier par Von Neuman dans [vN51] mais on doit les grands travaux fondateurs sur les automates finis à Kleene [Kle56] qui a étendu les travaux de McCulloch et Pitts sur les réseaux de neurones.

Plusieurs sortes d'automates ont été définis dans la littérature (se rapporter par exemple à [Sak03]), nous donnons ici une définition classique des automates finis.

Définition 2.1.4 (Automate fini) *Un automate fini sur un alphabet Σ est un tuple $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$ où Q est un ensemble fini d'états, $I \subseteq Q$ et $F \subseteq Q$ désignent respectivement l'ensemble des états initiaux et l'ensemble des états finaux, et $\Delta : Q \times \Sigma \times Q$ est une relation de transition.*

Alternativement, Δ peut être défini comme une fonction de $Q \times \Sigma$ dans l'ensemble des parties de Q qui est 2^Q. Dans le cas où Δ est une fonction de $Q \times \Sigma$ dans Q et I un singleton on dit que l'automate $\mathcal{A}$ est *déterministe*; dans le cas contraire $\mathcal{A}$ est dit *non déterministe*.

Un *chemin* ρ de $\mathcal{A}$ est une succession de transitions consécutives de Δ; c'est à dire ρ peut s'écrire $\rho = s_0 \xrightarrow{a_1} s_1 \cdots \xrightarrow{a_n} s_n$ avec $(s_i, a_{i+1}, s_{i+1}) \in \Delta$. ρ est un *chemin acceptant* si $s_0 \in I$ et $s_n \in F$. On définit pour chaque chemin ρ le mot $l(\rho) = a_1 \cdots a_n$ de Σ^*. Le langage de l'automate $\mathcal{A}$ est l'ensemble des éléments de Σ^* correspondant à des chemins acceptants de $\mathcal{A}$, autrement dit, $L(\mathcal{A}) = \{l(\rho) \mid \rho \text{ est un chemin acceptant de } \mathcal{A}\}$.

Définition 2.1.5 (Langage reconnaissable) *Un langage $L \subseteq \Sigma^*$ est reconnaissable s'il existe un automate $\mathcal{A}$ sur Σ tel que $L = L(\mathcal{A})$. On note REC(Σ^*) l'ensemble des langages reconnaissables de Σ^*.*

Le lien entre les langages reconnaissables et réguliers a été établi par Kleene. En particulier, il a démontré que ces ensembles sont égaux sur le monoïde libre. Ceci constitue l'un des résultats principaux dans la théorie des langages.

Théorème 2.1.1 ([Kle56]) *Un langage $L \subseteq \Sigma^*$ est régulier si et seulement si il est reconnaissable, et donc REG(Σ^*) = REC(Σ^*).*

2.1.4 Produit synchronisé d'Automates

Il peut être important de considérer les produits synchronisés d'automates dans le cadre d'une étude sur la modélisation de systèmes concurrents que nous verrons par la suite. Un produit synchronisé d'automates est paramétré par un ensemble de synchronisation permettant de définir quelles transitions des automates devront s'exécuter de manière synchronisée. Pour définir le produit synchronisé d'automates, on introduit la notion du vide, notée _. Un automate qui franchit une transition étiquetée par le vide (transition vide) ne fait rien et reste dans le même état, c'est une boucle sur l'état courant. Ainsi, chaque transition vide est de la forme $(q, \_, q)$.

Définition 2.1.6 (Produit synchronisé) *Le produit synchronisé d'un ensemble de n automates $\{\mathcal{A}_1 \ldots \mathcal{A}_n\}$ où pour tout $i \in [1..n]$, $\mathcal{A}_i = (\Sigma_i, Q_i, \Delta_i, I_i, F_i)$ avec l'ensemble de synchronisation $S \subset \prod_{1 \leq i \leq n}(\Sigma_i \cup \{\_\})$, est un automate $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$ où:*

- $Q = Q_1 \times \ldots \times Q_n$
- $I = I_1 \times \ldots \times I_n$
- $F = F_1 \times \ldots \times F_n$
- $\Sigma = S$
- $\Delta \subseteq Q \times \Sigma \times Q$ *et restreint aux seules transitions présentes dans* S.

Le nombre d'états de l'automate produit est le produit des nombres d'états des automates qui le composent

2.1.5 Automate et langage de mots infinis

Propulsé par les résultats sur le cas des mots finis, un corpus théorique stable et cohérent a été développé sur les mots infinis.

Définition 2.1.7 ($\omega - langage$) *Un* $\omega-$*langage est un ensemble de mots infinis sur* Σ. *En d'autres termes, c'est une partie de* Σ^ω.

En plus des opérations sur les langages, on dispose d'une nouvelle opération qui associe à un langage $L \subseteq \Sigma^*$ un langage de mots infinis: *l'itération infinie* définie comme suit:

$$L^\omega = \{u_0 \ldots u_n \ldots \mid \forall n \in \mathbb{N},\ u_n \in L \setminus \{\varepsilon\}\}$$

Nous ajoutons encore pour tout $L \subseteq \Sigma^*$, $L^\infty = L^* \cup L^\omega$.

Définition 2.1.8 (Langage ω-regulier) *La classe des langages* $\omega - reguliers$ *est le plus petit ensemble des parties de* Σ^∞ *contenant le langage vide et tous les langages réduits à un mot de longueur* ≤ 1, *et qui soit fermé pour la concaténation, l'union, la fermeture de Kleene et l'itération infinie.*

On note REG(Σ^ω) l'ensemble des langages $\omega - reguliers$ de Σ^∞ inclus dans Σ^ω.

Exemple 2.1.1 *L'ensemble des mots sur l'alphabet* $\Sigma = \{a, b\}$ *contenant une infinité de* b *et commençant par* a *est un langage* $\omega-$*regulier appartenant à REG*(Σ^ω). *Il peut être écrit avec l'expression régulière* $a(a^*b)^\omega$.

Il existe plusieurs automates pouvant reconnaitre des mots infinis (nous les appelons les $\omega - automates$. Pour plus d'informations se référer à [Eil74, Tho96, PP04]. Nous nous intéresserons à celui de Büchi.

Définition 2.1.9 (Automate de Büchi) *Un automate de Büchi est un automate* $\mathcal{B} = (\Sigma, Q, \Delta, I, F)$ *ayant une condition d'acceptation particulière (dit acceptation de Büchi): un chemin (infini)* ρ *est acceptant si et seulement si son état de départ est initial et* $Inf(\rho) \cap F \neq \emptyset$ *avec* $Inf(\rho)$ *désignant l'ensemble des états passés infiniment souvent par* ρ.

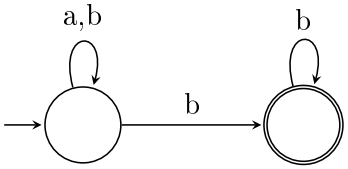


Figure 2.1: Automate de Büchi acceptant $(a+b)^*b^\omega$

L'ensemble des mots reconnus par un automate de Büchi $\mathcal{B}$ est l'ensemble des mots définissant un chemin partant d'un état initial et visitant F une infinité de fois. Cet ensemble est donc le langage de l'automate noté $L(\mathcal{B})$. La figure 2.1 montre une représentation d'un automate de Büchi $\mathcal{B}$ avec $L(\mathcal{B}) = (a+b)^*b^\omega$ (les états initiaux sont munis d'une flèche entrante sans état de départ, les états finaux sont doublement cerclés).

Un langage $L \subseteq \Sigma^\omega$ est *Büchi reconnaissable* s'il existe un automate de Büchi $\mathcal{B}$ tel que $L(\mathcal{B}) = L$.

Partant de ces définitions, Büchi [Büc62] a montré que le théorème de Kleene est naturellement extensible aux ω-langages. Les langages ω-réguliers correspondent strictement aux langages Büchi reconnaissables. Büchi poursuivit ses travaux en établissant le lien entre les automates et la logique. Les résultats qui en découlent sont encore utilisés dans le domaine de la vérification.

2.2 Logiques et Vérification sans concurrence

La vérification des systèmes assistée par ordinateur a connu, ces dernières décennies un grand succès particulièrement dans sa fondation théorique, mais aussi dans ses applications. Le but est de vérifier des propriétés de sûreté, de blocage, de validité etc. de systèmes de façon algorithmique, sans avoir besoin d'une compréhension approfondie du système à vérifier. La technique généralement utilisée est le *model-checking* qui se décline en trois aspects: un *modèle* S donnant une description formelle abstraite du système à étudier, *une spécification* ϕ renvoyant à la propriété à vérifier du système et un programme (*model-checker*) qui analyse automatiquement le modèle S et dit si oui ou non, la spécification ϕ est vérifiée par le système. L'un des fondements de cette méthode est l'équivalence entre la notion de reconnaissabilité par automates et la définissabilité logique. Le model-checking dans ce cas, revient à décider, pour un automate S et une propriété ϕ formulée en logique, si tous les mots acceptés par S satisfont ϕ.

Le lien entre logique et automate est bien établi et les pionniers Büchi et Elgot [Büc62, Elg61] ont basé leurs travaux sur la *logique monadique du second ordre*.

2.2.1 Logique Monadique du Second Ordre sur les mots

La logique du second ordre (SO) est une extension de la logique du premier ordre (FO) aux variables d'ensembles et de relations sur l'univers et autorise à quantifier sur celles-ci. La

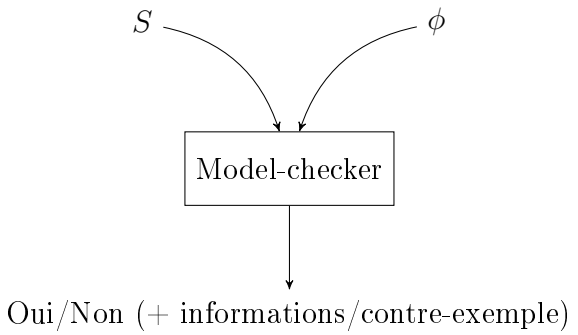


Figure 2.2: Fonctionnement Model-checker

logique monadique du second ordre (MSO) est la restriction de la logique SO aux prédicats du second ordre monadique (d'arité 1).

Dans cette section, nous présentons les syntaxes et sémantiques de la logique monadique du second ordre (MSO) et mettons en évidence sa relation avec les langages (ω)-réguliers. Nous verrons alors comment utiliser cette relation pour implémenter une procédure de vérification.

2.2.1.1 Syntaxes et sémantiques de MSO

La logique monadique du second ordre sur les mots (finis ou infinis) sur Σ est dénotée $MSO(\Sigma)$. Son vocabulaire consiste en une famille de prédicats unaires P_a pour chaque $a \in \Sigma$; un prédicat binaire $\leq$; un ensemble dénombrable de variables du premier ordre (désignées par des lettres en minuscules) $VFO = \{x, y, z, ...\}$; un ensemble dénombrable de variables du second ordre (variables ensemblistes conventionnellement désignées par des lettres majuscules) $VSO = \{X, Y, Z, ...\}$ et les quantificateurs et connecteurs logiques habituels du premier ordre.

Les formules de $MSO(\Sigma)$ sont alors données comme suit:

$$MSO ::= P_a(x) \mid x \in X \mid x \leq y \mid \neg\phi \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid (\exists x)\phi \mid (\exists X)\phi.$$

Une formule MSO est dite *close* si elle n'a pas de *variable libre* (variable sans quantificateur). Les formules closes sont aussi appelées *phrases*.

On définit maintenant par induction la satisfaction d'une formule MSO ϕ sur les mots.

Soit $u = u_1...u_n$ un mot de l'alphabet Σ et $N = \{1, ..., n\}$ l'ensemble des positions de u. Soient ρ et μ deux fonctions (partielles) telles que $\rho : VFO \mapsto N$ et $\mu : VSO \mapsto 2^N$. Nous définissons la relation de satisfaction $u, \rho, \mu \models \phi$ comme suit:

- $u, \rho, \mu \models P_a(x)$ si et seulement si $u_{\rho(x)} = a$
- $u, \rho, \mu \models x \in X$ si et seulement si $\rho(x) \in \mu(X)$
- $u, \rho, \mu \models x \leq y$ si et seulement si $\rho(x) \leq \rho(y)$
- $u, \rho, \mu \models \phi_1 \wedge \phi_2$ si et seulement si $u, \rho, \mu \models \phi_1$ et $u, \rho, \mu \models \phi_2$

- $u, \rho, \mu \models \phi_1 \vee \phi_2$ si et seulement si $u, \rho, \mu \models \phi_1$ ou $u, \rho, \mu \models \phi_2$
- $u, \rho, \mu \models \neg\phi$ si et seulement si $u, \rho, \mu \not\models \phi$
- $u, \rho, \mu \models \exists x\ \phi$ si et seulement si il existe $i \in N$ tel que $u, \rho', \mu \models \phi$ avec $\rho'(y) = i$ si $y = x$, $\rho'(y) = \rho(y)$ sinon
- $u, \rho, \mu \models \exists X\ \phi$ si et seulement si il existe un ensemble $P \subseteq N$ tel que $u, \rho, \mu' \models \phi$ avec $\mu'(Y) = P$ si $Y = P$, $\mu'(Y) = \mu(Y)$ sinon

Nous pouvons librement utiliser les abréviations suivantes:

- $x = y \stackrel{\text{def}}{=} x \leq y \wedge y \leq x$ et ainsi $x < y \stackrel{\text{def}}{=} x \leq y \wedge \neg(x = y)$
- $x \neq y \stackrel{\text{def}}{=} \neg(x = y)$
- $\forall x\ \phi \stackrel{\text{def}}{=} \neg(\exists x\ \neg\phi)$
- $x + 1 \in X \stackrel{\text{def}}{=} (\exists y)(x < y \wedge \neg(\exists z)(x < z \wedge z < y) \wedge y \in X)$. Cela peut être généralisé à $x + k \in X$ pour tout entier naturel k.
- $x \to y \stackrel{\text{def}}{=} \neg x \vee y$
- $x \leftrightarrow y \stackrel{\text{def}}{=} (x \to y) \wedge (y \to x)$
- $X \subseteq Y \stackrel{\text{def}}{=} (\forall x)(x \in X \to x \in Y)$ d'où aussi une (in)égalité sur des variables d'ensemble de la même manière. On peut construire aussi de manière similaire des formules pour les opérations d'union, d'intersection et de complémentation à partir de connecteurs propositionnels correspondants. Par exemple, $X \cap Y = Z$ peut être exprimé comme $(\forall x)(x \in X \wedge x \in Y \leftrightarrow x \in Z)$.

2.2.1.2 MSO et langages réguliers

L'ensemble des mots satisfaisant une formule close ϕ définit le langage ϕ noté $L(\phi)$. Nous disons que $L \subseteq \Sigma^\infty$ est définissable dans MSO(Σ) si et seulement si il existe une formule close $\phi \in MSO(\Sigma)$ telle que $L = L(\phi)$.

Le théorème fondamental dans ce domaine est le théorème de Büchi, qui était la motivation première pour les automates de Büchi.

Théorème 2.2.1 (Büchi [Büc62]) *Soit $L \subseteq \Sigma^\omega$. L est définissable dans $MSO(\Sigma)$ si et seulement si L est ω-régulier.*

Le théorème est aussi établi pour les langages réguliers de Σ^*. Nous donnons dans ce cadre le théorème de Kleene-Büchi qui est encore plus général.

Théorème 2.2.2 (Kleene-Büchi) *Les formules MSO, les reconnaissables, et les réguliers ont le même pouvoir expressif pour le monoïde libre Σ^**

2.2.2 Logique temporelle à temps linéaire

La logique temporelle a été introduite dans le domaine de la vérification formelle des systèmes dans une publication de Pnueli à la fin des années 70 [Pnu77]. Depuis, un large corpus de travaux est apparu et a permis à la logique temporelle à temps linéaire (LTL) de devenir un outil bien établi et bien compris pour spécifier le comportement dynamique des systèmes réactifs. Par rapport à d'autres logiques, LTL intègre de nouveaux opérateurs qui permettent d'exprimer la notion de temps. Elle permet ainsi d'exprimer des propriétés pour lesquels le temps se déroule linéairement.

2.2.2.1 Syntaxes et sémantiques de LTL

Les formules LTL sont construites à partir des propriétés élémentaires AP, des opérateurs et constantes booléennes et de deux opérateurs temporels : l'opérateur unaire $\mathbf{X}$ qui se lit "Next"(après) et l'opérateur binaire $\mathbf{U}$ qui se lit "Until"(jusqu'à). L'ensemble des formules de $LTL(\Sigma)$ est donné par la syntaxe:

$$LTL(\Sigma) ::= a \mid \neg\phi \mid \phi_1 \vee \phi_2 \mid \mathbf{X}_\phi \mid \phi_1 \mathbf{U} \phi_2 \text{ avec } a \in \Sigma$$

Soit ϕ une formule LTL, pour un mot $u = u_0 u_1 \ldots$ fini ou infini de l'alphabet $\Sigma = 2^{AP}$, on définit la relation de satisfaction $u, i \models \phi$, lue "u satisfait ϕ à la position (ou au temps) i" (où $0 \leq i \leq |u| - 1$), de la façon suivante.

- $u, i \models a$ si $a \in u_i$ pour tout $a \in AP$;
- $u, i \models \neg\phi$ si l'on n'a pas $u, i \models \phi$;
- $u, i \models \phi_1 \wedge \phi_2$ si $u, i \models \phi_1$ et $u, i \models \phi_2$;
- $u, i \models \mathbf{X}\phi$ si $i + 1 \leq |u| - 1$ (pour les mots finis) et $u, i + 1 \models \phi$;
- $u, i \models \phi_1 \mathbf{U} \phi_2$ s'il existe un entier j qui satisfait les conditions suivantes:
  - $i \leq j \leq |u| - 1$
  - $u, j \models \phi_2$,
  - pour tout k tel que $i \leq k \leq j - 1$, on a: $u, k \models \phi_1$.

On dit qu'un mot u satisfait une formule ϕ s'il la satisfait à l'instant 0, c'est-à-dire si $u, 0 \models \phi$.

De la même manière que pour MSO, on peut définir plusieurs abréviations en plus des bien connues $\vee$ (ou), $\rightarrow$ (implique), $\leftrightarrow$ (équivalent), $\top$ (vrai), $\bot$ (faux) comme: $F\phi \stackrel{def}{\equiv} \top \mathbf{U} \phi$, $G\phi \stackrel{def}{\equiv} \neg F \neg \phi$. $F\phi$ dit que ϕ est vraie dans un instant futur, le dual $G\phi$ dit que ϕ est vraie dans tous les instants futurs.

2.2.2.2 LTL et langage

L'ensemble des mots satisfaisant une formule ϕ de $LTL(\Sigma)$ représente le langage $L(\phi)$ de ϕ. On dit qu'un langage L est définissable dans $LTL(\Sigma)$ s'il existe une formule ϕ de $LTL(\Sigma)$ telle que $L = L(\phi)$. On dit aussi que ϕ définit L.

Exemple 2.2.1 *Le langage* $(ab)^*$ *est engendré par la formule LTL suivante* $a \wedge G((a \wedge \mathbf{X}b) \vee (b \wedge \mathbf{X}a)) \wedge F(b \wedge \neg\mathbf{X}\top)$.

Tout langage d'une formule LTL est aussi le langage d'une formule MSO. Ceci vient du fait que les formules LTL ont le même pouvoir expressif que la logique du *premier ordre FO* qui est une sous-classe de MSO.

2.2.3 Satisfiabilité et model-checking

Un système peut être vu comme un générateur de mots infinis. Il peut être représenté par un système de transition étiqueté que l'on peut facilement transformer en automate. Le problème du model-checking est de vérifier si tous les comportements d'un système de transitions dont les états vérifient des propriétés AP vérifient une formule ϕ. Ainsi, si S est un système et ϕ une formule, tester $S \models f$ se ramène à un problème sur les automates. L'idée est de transformer S en un automate $\mathcal{A}_S$ qui accepte les observations de S, puis de transformer la formule ϕ en automate de mots infinis $\mathcal{A}_\phi$, tels que $L(\phi) = L(\mathcal{A}_\phi)$. Ensuite on peut tester:

- si $L(\mathcal{A}_\phi) = \emptyset$, autrement dit, si la formule ϕ est *satisfiable*;
- si $L(\mathcal{A}_\mathcal{S}) \cap L(\mathcal{A}_{\neg\phi}) = \emptyset$, autrement dit, si tous les comportements du système S sont corrects vis-à-vis de ϕ (*model-checking*).

Cette approche pose clairement les problèmes algorithmiques de la vérification : (1) calculer un ω-automate pour une formule logique; (2) calculer l'intersection de deux automates ; (3) tester le vide d'un automate.

Pour LTL le problème de satisfiabilité ainsi que celui du model-checking sont PSPACE [1] [SC85, DS02]. De nombreux travaux traitent de la construction d'un automate pour une formule LTL. On pourra trouver dans [Cou99, Wol00, Tau03] plusieurs exemples. Le problème du modèle checking est *non élémentaire*[2] pour MSO [Sto74]. Ces complexités proviennent du fait que, la complémentation et la déterminisation d'un automate conduisent en général à une explosion combinatoire du nombre d'états et de transitions de l'automate résultant. Ce qui rend impossible l'implémentation d'un processus de décision en temps raisonnable par une machine.

Cependant, il est possible de limiter l'explosion combinatoire du nombre d'états et de transition en utilisant des représentations symboliques d'automates telles que les BDD (Binary Decision Diagrams) [Ake78, Bry86]. Des outils comme *Mona* [KM01] utilisent cet aspect pour implémenter une procédure de vérification basée sur MSO.

[1] PSPACE est la classe des problèmes résolubles par des machines de Turing déterministes, dont la mémoire est bornée par une fonction polynomiale de la taille des entrées

[2] "non élémentaire" veut dire en temps $f(n) = 2^{2^n}$, où la tour d'exponentielles a pour hauteur n

Une autre solution proposée est la vérification à la volée [GPVW95] qui consiste à ne construire les éléments de l'automate qu'à la demande. Le model-checker LTL *SPIN* [Hol03] est l'un des outils les plus connus implémentant cette solution en plus des techniques de réduction à base *d'ordre partiel*.

2.3 Pomsets et logique avec concurrence

Cette section présente les notions principales des ensembles partiellement ordonnés (pomsets) et leurs langages. Parlons d'abord des relations d'ordre au sens mathématique, fondement de toute théorie sur les ordres partiels.

2.3.1 Relation d'ordre et ordre partiel

Une *relation d'ordre* est une relation à la fois réflexive, antisymétrique et transitive. Un ensemble ordonné (E, R) est un ensemble E muni d'une relation d'ordre R. Une relation d'ordre R sur E est *totale* si tous les éléments sont comparables c'est à dire $\forall x, y \in E, xRy$ *ou* yRx. Un *ordre partiel* est un ordre qui n'est pas nécessairement total. Un *pomset* ("partial order set") est un ensemble partiellement ordonné que l'on peut noter par $p = (E, \preceq)$, $\preceq$ désignant la relation d'ordre partiel.

Diagramme de Hasse (DAG) : sur un ensemble fini, on peut représenter une relation R par son graphe. Si R est une relation d'ordre, son graphe ne contient pas de circuit de longueur supérieur ou égal à deux. Le diagramme de Hasse est alors une représentation plus simple. On le construit en appliquant la règle suivante:

- Les éléments sont représentés par des sommets
- Deux sommets a et b sont joints par une arête si et seulement si : aRb et il n'existe pas de $c \in E$ tel que aRc et cRb.

2.3.2 Théorie des pomsets

La théorie des pomsets ("Partial order multisets") s'est développée de manière analogue à la théorie des langages de mots. Cette section donne une définition des pomsets et des langages pomsets en se basant notamment sur les travaux de Gischer [Gis85] et Pratt [Pra86] pères fondateurs de la théorie.

Définition 2.3.1 (EPOE) *Soit Σ un alphabet fini. Un Ensemble Partiellement Ordonné Étiqueté (EPOE) est un triplé $(E, \preceq, \lambda)$ où E est un ensemble fini de sommets (qui représentent des évènements), $\preceq \subset E \times E$ est une relation binaire, réflexive, anti-symétrique et transitive représentant l'ordre partiel sur E et $\lambda : E \longrightarrow \Sigma$ est une fonction injective d'étiquetage d'élément de E par une action de l'alphabet Σ.*

Définition 2.3.2 (Pomset) *Un isomorphisme entre deux EPOE $(E, \preceq, \lambda)$ et $(E', \preceq', \lambda')$ est une bijection $f : E \longrightarrow E'$ telle que pour tout x et y de E, $\lambda' of(x) = \lambda(x)$ et $x \preceq y \Rightarrow f(x) \preceq f(y)$. Un pomset est la classe d'un ensemble d'EPOE isomorphe.*

Remarque: Pour un EPOE $(E, \preceq, \lambda)$, le pomset associé se note généralement $[E, \preceq, \lambda]$. Cependant nous utiliserons un représentant de la classe d'isomorphisme pour désigner un pomset. Nous utilisons donc les parenthèses pour la notation.

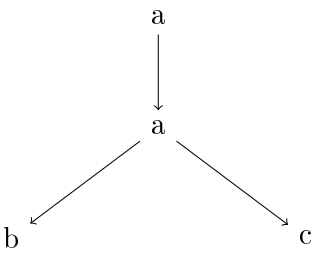


Figure 2.3: Exemple de pomset

Exemple 2.3.1 *Un pomset peut être représenté graphiquement par son diagramme de Hasse orienté comme à la figure 2.3. Le pomset représenté est celui défini par* $p = (E, \preceq, \lambda)$ *avec:*

- $E = \{1, 2, 3, 4\}$
- $1 \preceq 2, 1 \preceq 3, 1 \preceq 4, 2 \preceq 3, 2 \preceq 4$
- $\lambda(1) = a,\ \lambda(2) = a,\ \lambda(3) = b,\ \lambda(4) = c$

Nous notons $\mathbb{P}(\Sigma)$ l'ensemble de tous les pomsets finis sur Σ.

L'ensemble $(\emptyset, \emptyset, \emptyset)$ représente le *pomset vide* que nous désignerons plus simplement par ϵ ou $\emptyset$.

Soit $p = (E, \preceq, \lambda)$ un pomset. Les sommets désignent des événements de E.

La fermeture vers le bas (resp.fermeture stricte vers le bas) d'un sommet y est notée et définie par $\downarrow y = \{x \in E \mid x \preceq y\}$ (resp. $\Downarrow y = \downarrow y - \{y\}$). Autrement dit, c'est l'ensemble des prédécesseurs d'un événement. Dans l'exemple 2.3.1 on a $\Downarrow 4 = \{1, 2\}$.

On appelle *configuration (ou préfixe)* de p tout sous-ensemble fini $C \subseteq E$ fermé par le passé, i.e.: $\forall x \in C, \forall y \in E, (y \preceq x \Rightarrow y \in C)$.

Nous disons qu'un sommet y *couvre* un sommet x noté $x \lessdot y$ si $x \preceq y$, $x \neq y$ et $\nexists z \in E \mid x \preceq z \preceq y$. $x \lessdot y$ signifie donc que x est un prédécesseur immédiat de y et la relation de couverture $\lessdot$ correspond à la plus petite relation dont la fermeture transitive est égale $\preceq$.

Si on n'a pas $(x \preceq y)$ ni $(y \preceq x)$ alors on dit que x et y sont *concurrents* ou *parallèles* et on le note $x \parallel y$.

Un ensemble $A \subset E$ est un *antichaine* si tous ses éléments sont mutuellement concurrents.

Un pomset p est *"auto-concurrent"* s'il existe deux sommets x et y concurrents avec $\lambda(x) = \lambda(y)$.

Définition 2.3.3 (Restriction et Projection) *Soit* $p = (E, \preceq, \lambda)$ *un pomset sur l'alphabet* Σ. *La restriction de* p *à un ensemble d'événements* $A \subseteq E$, *est le pomset* $p_{|A} = (A, \preceq_{|A}, \lambda_{|A})$ *où* $\lambda_{|A}$ *est la restriction de* λ *au domaine* A *et* $\preceq_{|A}$ *celle de* $\preceq$ *aux événements de* A. *La projection de* p *sur un alphabet observable* $\Sigma_o \subseteq \Sigma$, *est une fonction* f_{Σ_o} *qui restreint* p *à ses éléments observables; c-a-d* $f_{\Sigma_o}(p) = p_{|\lambda^{-1}(\Sigma_o)}$

Définition 2.3.4 (Extension linéaire) *Soit $p = (E_p, \preceq_p, \lambda_p)$ un pomset de $\mathbb{P}(\Sigma)$. Une extension linéaire (ou linéarisation) de p est un pomset $t = (E_t, \preceq_t, \lambda_t)$ dont la relation d'ordre est totale et étant celle de p, (c.a.d. $\preceq_p \subseteq \preceq_t$). Une extension linéaire $x_1 \lessdot \ldots \lessdot x_n$ de p peut être vue comme un mot $u = \lambda_p(x_1) \ldots \lambda_p(x_n)$ de Σ^* avec $E_p = \{x_1 \ldots x_n\}$.*

Nous notons par $LE(p)$ l'ensemble des (mots correspondant aux) linéarisations d'un pomset p. Pour le pomset de l'exemple 2.3.1 on a $LE(p) = \{aabc, aacd\}$.

Définition 2.3.5 (Langage de pomset) *Un langage de pomset sur Σ est un ensemble de pomsets sur Σ. Autrement dit c'est un sous-ensemble de $\mathbb{P}(\Sigma)$.*

Notons que, comme tout mot est aussi un pomset, chaque langage de mot est aussi un langage de pomset.

Pour construire de nouveaux pomsets à partir d'autres pomsets et par la suite, de nouveaux langages de pomsets à partir d'autres langages de pomsets, les deux opérateurs suivants sur les pomsets sont introduits. Le premier est une composition séquentielle. Deux pomsets sont joints de sorte que chaque sommet du premier précède chaque sommet du second. Le second opérateur est une composition concurrente. Deux pomsets sont joints sans ajouter de contraintes d'ordre.

Soient $p_1 = (E_1, \preceq_1, \lambda_1)$ et $p_2 = (E_2, \preceq_2, \lambda_2)$ deux pomsets avec $E_1 \cap E_2 = \emptyset$:

Produit séquentiel (concaténation): $p_1 \parallel p_2 = (E_1 \cup E_2, \preceq_1 \cup \preceq_2, \lambda_1 \cup \lambda_2)$ où $\lambda_1 \cup \lambda_2$ représente la fonction obtenue par l'union des graphes des fonctions d'étiquetage λ_1 et λ_2.

Produit parallèle: $p_1 \bullet p_2 = (E_1 \cup E_2, \preceq_1 \cup \preceq_2 \cup (E_1 \times E_2), \lambda_1 \cup \lambda_2)$.

Voir l'illustration avec les exemples de la figure 2.4.

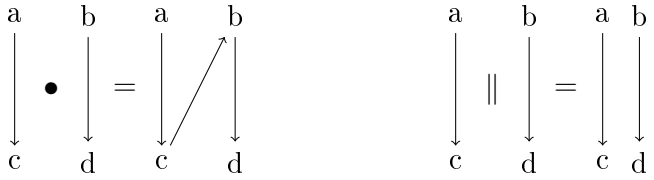


Figure 2.4: Exemple de concaténation (à gauche) et concurrence (à droite) de deux pomsets

L'étude des langages de pomsets en terme de reconnaissabilité, et de régularité n'est pas triviale. La difficulté réside sur l'absence d'ensembles finis d'opérateurs pouvant les générer tous contrairement aux mots [Gis88]. Des classes particulières de pomsets ont été définies et leur régularité et reconnaissabilité établies selon une approche algébrique. Ce sont principalement les monoïdes de traces [Die95] et les Messages Sequences Chart [Mor02]. Des automates asynchrones pour des classes restreintes de pomsets sans auto-concurrence ont aussi été définies [DGK00]. Pour ce qui est des pomsets avec auto-concurrence, les séries-parallèles font partie des rares modèles où les résultats de Kleene et de Büchi ont été appliqués [LW00, Kus00b]. Nous étudierons ces modèles dans le chapitre suivant.

2.3.3 Logique sur les pomsets

Comme sur les mots, il est possible d'utiliser la logique pour désigner un ensemble de pomsets. Nous allons nous intéresser dans cette partie à la logique monadique du second ordre pour les pomsets, qui est la logique qui nous intéresse dans le cadre de cette thèse.

2.3.3.1 Logique MSO sur les pomsets

Comme sur les mots, il est possible d'utiliser la logique pour décrire un ensemble de pomsets. Chaque pomset est traité comme une structure. Les variables, libres ou quantifiées, portent sur les évènements. Les variables du premier ordre $x, y, z...$ désignent des évènements discrets et les variables ensemblistes du second ordre $X, Y, Z...$ renvoient à des ensembles d'évènements. Les formules MSO sont construites à partir de prédicats ou formules atomiques $P_a(x)$ pour $a \in \Sigma$ (qui traduit: "l'évènement x est étiqueté par l'action a"), $x \preceq y$, et $x \in X$ et par usage des connecteurs booléens habituels $\neg, \vee, \wedge, \rightarrow, \leftrightarrow$ et des quantificateurs $\exists, \forall$ (aussi bien pour les variables du premier ordre que pour les ensembles).

Nous notons par $MSO(\Sigma, \preceq)$ l'ensemble des formules MSO sur pomsets. La principale différence avec $MSO(\Sigma)$ réside sur l'ajout de la relation d'ordre patiel $\preceq$.

Nous pouvons librement utiliser les abréviations suivantes:

- $x \prec y \stackrel{\text{def}}{=} x \neq y \wedge y \preceq x$

Les relations de couverture $\lessdot$ et de concurrence $\|$ ayant été logiquement définies précédemment, peuvent aussi être utilisées. Nous parlerons dans ce cas plus formellement de relation de :

- Causalité ($\lessdot$): $x \lessdot y \leftrightarrow x \prec y \wedge (\nexists z \in E | x \prec z \prec y)$
- Concurrence ($\|$): $x || y \leftrightarrow \neg(x \preceq y) \wedge \neg(y \preceq x)$

Il est important pour la suite de remarquer que $\preceq$ peut à tout moment être remplacé par $\lessdot$ car $x \preceq y$ est équivalent à:

$$\forall X[(y \in X \wedge \forall y_1, y_2(y_1 \lessdot y_2 \wedge y_2 \in X \rightarrow y_1 \in X)) \rightarrow x \in X]$$

De ce fait, utiliser $MSO(\Sigma, \preceq)$ ou $MSO(\Sigma, \lessdot)$ ne change pas le pouvoir d'expression de la logique.

La relation de satisfaction $\models$ entre un pomset $p = (E, \preceq, \lambda)$ et une phrase ϕ d'une logique monadique de second ordre est définie de manière équivalente à celle dans $MSO(\Sigma)$ pour les mots avec en plus la compréhension que pour une interprétation I, $p \models_I x \preceq y$ ssi $I(x) \preceq I(y)$.

La classe des pomsets qui satisfait une formule MSO ϕ est dénotée par $L(\phi)$. Nous disons qu'un langage de pomset $L \subseteq \mathbb{P}(\Sigma)$ est MSO-définissable s'il existe une phrase monadique du second ordre ϕ tel que $L = L(\phi)$.

Exemple 2.3.2 *Nous considérons ici un système producteur-consommateur. Son alphabet est $\Sigma = \{p, c\}$ où p représente une production d'un article et c une consommation. La figure*

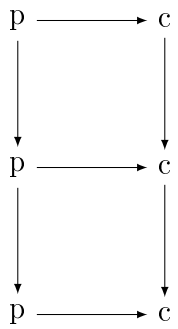


Figure 2.5: Pomset représentant un système producteur-consommateur

2.5 est une représentation en pomset de ce système. Le langage de ce système décrit toutes les séquences possibles pour lesquelles à chaque stade il y a au moins autant de productions que de consommations. Les pomsets propres à ce langage sont MSO-définissables avec la conjonction des propriétés suivantes:

(1) $\forall x : P_p(x) \vee P_c(x)$,

(2) $\forall x, y : (P_p(x) \wedge P_p(y)) \rightarrow (x \preceq y \vee y \preceq x)$,

(3) $\forall x, y : (P_c(x) \wedge P_c(y)) \rightarrow (x \preceq y \vee y \preceq x)$,

(4) $\forall x, y : (P_p(y) \wedge x \preceq y) \rightarrow P_p(x)$,

(5) $\forall y : (P_c(y) \rightarrow \exists x(P_p(x) \wedge x \prec y))$,

En langage naturel, ces propriétés décrivent successivement:

(1): Chaque événement est soit une production ou une consommation,

(2): Les productions sont ordonnées entre elles,

(3): Les consomations sont ordonnées entre elles,

(4): Les productions sont précédées par des productions,

(5): Toute consommation admet une production qui le précède immédiatement,

2.3.3.2 Problème de la satisfaction

Dans cette partie, nous montrons que le satisfaction d'une formule de $MSO(\Sigma, \preceq)$ est indécidable. Cette indécidabilité constitue l'une des motivations de nos travaux. Nous allons donc en donner la preuve complète.

Remarque: Si on réduit un problème A indécidable à un problème B, alors B est lui aussi indécidable.

Nous allons traduire le problème de correspondance de Post en un problème de satisfaction d'une formule $MSO(\Sigma, \preceq)$. Dans un premier temps, présentons l'énoncé du problème de correspondance de Post.

Définition 2.3.6 *[Problème de correspondance de Post [Pos46]] Soit une liste finie* $L = (\alpha_1, \beta_1), \ldots, (\alpha_N, \beta_N)$ *de couple de mots d'un alphabet* Σ *ayant au moins deux symboles. Une solution du problème est une suite d'indices* $(i_k)_{1 \leq k \leq K}$ *avec* $K > 0$ *et* $1 \leq i_k \leq N$ *pour tous les* k*, telle que les concaténations* $\alpha_{i_1}, \ldots, \alpha_{i_K}$ *et* $\beta_{i_1}, \ldots, \beta_{i_K}$ *soient égales. Le problème de correspondance de Post (PCP) consiste à déterminer si une solution existe ou non.*

Exemple 2.3.3 *Posons* $\Sigma = \{a, b\}$ *et* $L = (a, ab), (ba, a)$*. Une solution du problème PLC est* $(1, 2)$

Théorème 2.3.1 *[Pos46] Le problème de correspondance de Post est indécidable pour tout alphabet contenant au moins deux lettres*

Construisons un pomset traduisant une solution d'un problème de Post.

Définition 2.3.7 *Soit la liste* $L = (\alpha_1, \beta_1), \ldots, (\alpha_N, \beta_N)$ *de couple de mots d'un alphabet* Σ*, donnée d'un problème de correspondance de Post ayant une solution* $(i_k)_{1 \leq k \leq K}$*. Posons* $\Sigma' = \Sigma \cup_{i \in [1,N]} \{(_i,)_i\}$ *l'alphabet* Σ *enrichi des parenthèses. Notons*

$$\begin{aligned} \sigma_1 &= (_{i_1}\alpha_{i_1})_{i_1} \ldots (_{i_K}\alpha_{i_K})_{i_K} \\ \sigma_2 &= (_{i_1}\beta_{i_1})_{i_1} \ldots (_{i_K}\beta_{i_K})_{i_K} \\ m &= \alpha_{i_1} \ldots \alpha_{i_K} = \beta_{i_1} \ldots \beta_{i_K} \\ p &= (_{i_1})_{i_1} \ldots (_{i_K})_{i_K} \end{aligned}$$

Considérons le pomset $\pi = (E, \preceq, \lambda)$ *sur l'alphabet* $\{\sigma_1, \sigma_2\} \times \Sigma' \cup \{m\} \times \Sigma \cup \{p\} \times \cup_{i \in [1,N]} \{(_i,)_i\}$ *avec :*

- *L'ensemble des évènements est donné par l'ensemble des positions des mots* σ_1*,* σ_2*,* m *et* p*. Formellement :*

$$\begin{aligned} E &= \{e^{\sigma_1}_{k,j} \text{ avec } 0 < k \leq K \text{ et } 0 \leq j < |\alpha_{k_j}|\} \\ &\cup \{e^{\sigma_1}_{k,(} \text{ avec } 0 < k \leq K\} \cup \{e^{\sigma_1}_{k,)} \text{ avec } 0 < k \leq K\} \\ &\cup \{e^{\sigma_2}_{k,j} \text{ avec } 0 < k \leq K \text{ et } 0 \leq j < |\beta_{k_j}|\} \\ &\cup \{e^{\sigma_2}_{k,(} \text{ avec } 0 < k \leq K\} \cup \{e^{\sigma_2}_{k,)} \text{ avec } 0 < k \leq K\} \\ &\cup \{e^{m}_{k,j} \text{ avec } 0 < k \leq K \text{ et } 0 \leq j < |\alpha_{k_j}|\} \\ &\cup \{e^{p}_{k,(} \text{ avec } 0 < k \leq K\} \cup \{e^{p}_{k,)} \text{ avec } 0 < k \leq K\} \end{aligned}$$

- *La relation d'ordre* $\preceq$ *est construite de manière à ce que les événements associés aux mots* σ_1*,* σ_2*,* m *et* p *soient totalement ordonnés et que les évènements relatifs aux mots* σ_1*,* σ_2 *ont chacun un successeur immédiat dans les évènements liés à* m *ou* p *modélisant la projection des mots* σ_1*,* σ_2 *dans* m *et* p*. Plus précisément, nous avons :*
  - $\forall k, k' \in [1, K], \forall j \in [0, |\alpha_{i_k}|[\cup\{(,)\}, \forall j' \in [0, |\alpha_{i'_k}| - 1[\cup\{(,)\} : e^{\sigma_1}_{k,j} \preceq e^{\sigma_1}_{k',j'}$ *si* $k < k' \vee (k = k' \wedge (j =' ('\vee j' =')' \vee j \leq j'))$
  - $\forall k, k' \in [1, K], \forall j \in [0, |\beta_{i_k}|[\cup\{(,)\}, \forall j' \in [0, |\beta_{i'_k}| - 1[\cup\{(,)\} : e^{\sigma_2}_{k,j} \preceq e^{\sigma_2}_{k',j'}$ *si* $k < k' \vee (k = k' \wedge (j =' ('\vee j' =')' \vee j \leq j'))$

- $\forall k, k' \in [1, K], \forall j \in [0, |\alpha_{i_k}|[, \forall j' \in [0, |\alpha_{i'_k}| - 1[: e^m_{k,j} \preceq e^m_{k',j'}$ *si* $k < k' \vee (k = k' \wedge j \leq j'))$
- $\forall k, k' \in [1, K], \forall j, j' \in \{(,)\} : e^p_{k,j} \preceq e^p_{k',j'}$ *si* $k < k' \vee (k = k' \wedge (j =' ('\vee j' =')'))$
- $\forall k \in [1, K], \forall j \in [0, |\alpha_{i_k}|[: e^{\sigma_1}_{k,j} \lessdot e^m_{k,j}$
- $\forall k \in [1, K], \forall j \in \{(,)\} : e^{\sigma_1}_{k,j} \lessdot e^p_{k,j}$
- $\forall k \in [1, K], \forall j \in [0, |\beta_{i_k}|[: e^{\sigma_2}_{k,j} \lessdot e^m_{k',j'}$ *avec* $\sum_{u=1}^{k} |\beta_{i_u}| + j = \sum_{u=1}^{k'} |\alpha_{i_u}| + j'$ *et* $|\alpha_{i_{k'}}| \neq 0$
- $\forall k \in [1, K], \forall j \in \{(,)\} : e^{\sigma_2}_{k,j} \lessdot e^p_{k,j}$

- *La fonction d'étiquetage des évènements est donnée par :*
  - $\forall k \in [1, K], \forall j \in [0, |\alpha_{i_k}|[, \lambda(e^{\sigma_1}_{k,j}) = \{\sigma_1, \alpha_{i_k}[j]\}$ *où* $\alpha_{i_k}[j]\}$ *est la j-ième lettre de* α_{i_k},
  - $\forall k \in [1, K], \forall j \in \{(,)\}, \lambda(e^{\sigma_1}_{k,j}) = \{\sigma_1, j\}$,
  - $\forall k \in [1, K], \forall j \in [0, |\beta_{i_k}|[, \lambda(e^{\sigma_2}_{k,j}) = \{\sigma_2, \beta_{i_k}[j]\}$ *où* $\beta_{i_k}[j]\}$ *est la j-ième lettre de* β_{i_k},
  - $\forall k \in [1, K], \forall j \in \{(,)\}, \lambda(e^{\sigma_2}_{k,j}) = \{\sigma_2, j\}$,
  - $\forall k \in [1, K], \forall j \in [0, |\alpha_{i_k}|[, \lambda(e^m_{k,j}) = \{m, \alpha_{i_k}[j]\}$,
  - $\forall k \in [1, K], \forall j \in \{(,)\}, \lambda(e^p_{k,j}) = \{p, j\}$,

Exemple 2.3.4 *Le pomset de l'exemple 2.3.3 est donné par la figure 2.6.*

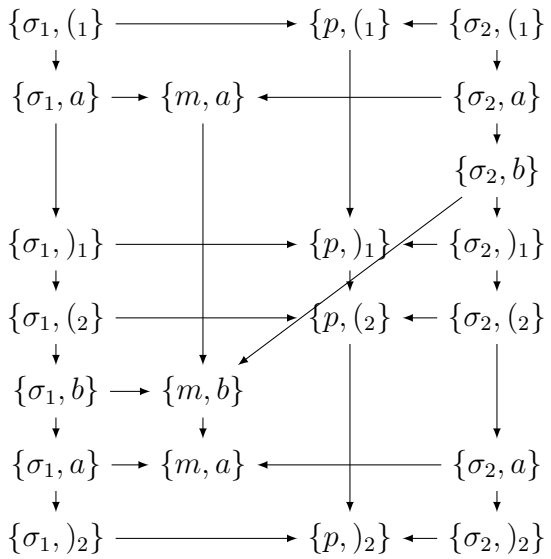


Figure 2.6: Pomset représentant une solution du problème de correspondance de Post

Caractérisons le pomset de la définition 2.3.6 à partir d'une formule $MSO(\Sigma, \preceq)$. Cette formulation repose sur quatre ingrédients :

1. les évènements représentants les mots σ_1, σ_2, m et p doivent être totalement ordonnés
2. les mots σ_1 et σ_2 doivent représenter une concaténation des mots des premières et deuxièmes composantes des couples de mots définissant le problème de Post.
3. les projections de σ_1, σ_2 sur les parenthèses donnent le mot p sous la forme d'une relation de successeur immédiat
4. les projections de σ_1, σ_2 sur les lettres (autres que parenthèse) donnent le mot m sous la forme d'une relation de successeur immédiat

Donnons pour chacune de ces contraintes, les formules $MSO(\Sigma, \preceq)$ correspondantes.

1. Soit θ une étiquette parmi σ_1, σ_2, m ou p, la priorité totalement ordonnée s'écrit
$$\phi_1(\theta) = \forall e, e' : \theta \in \lambda(e) \cap \lambda(e') \Rightarrow (e \preceq e' \vee e' \preceq e)$$
2. Soit $(\alpha_i)_i$ une famille finie de mots, spécifions qu'un sous ensemble d'événements totalement ordonné étiqueté par σ forme une concaténation des mots parenthésés de $(\alpha_i)_i$:
   (a) le premier événement est une parenthèse :
   $$\phi_{2,1}(\sigma, (\alpha_i)_i) = \forall e : (\neg\exists e' : e' < e) \wedge m \in \lambda(e) \Rightarrow \exists i : ({}_i \in \lambda(e))$$
   (b) un événement étiqueté par σ et $({}_i$ a son j-ième successeur étiqueté par la j-ième lettre de α_i
   $$\phi_{2,2}(\sigma, (\alpha_i)_i) = \forall i, \forall e : \sigma \in \lambda(e) \Rightarrow \forall j \in [0, |\alpha_i|[, \exists e' : \sigma, \alpha_i[j] \in \lambda(e') \wedge e \lessdot^{j+1} e'$$
   (c) un évènement étiqueté par σ et $({}_i$ a son $|\alpha_i| + 1$-ième successeur étiqueté par la lettre de $)_i$
   $$\phi_{2,3}(\sigma, (\alpha_i)_i) = \forall i, \forall e : \sigma \in \lambda(e) \Rightarrow \exists e' : \sigma,)_j \in \lambda(e') \wedge e \lessdot^{|\alpha_i|+1} e'$$
3. Soit σ et m des étiquettes donnant deux ensembles d'évènements disjoints totalement ordonnés. Soit L un ensemble d'étiquettes. Donnons une formule spécifiant que σ se projette en m sur les étiquettes par la relation de successeur immédiat:
   (a) tout événement de σ étiqueté par une lettre de L se projette dans m
   $$\phi_{3,1}(\sigma, m, L) = \forall e, \forall l \in L : \sigma, l \in \lambda(e) \Rightarrow \exists e' : m, l \in \lambda(e')$$
   (b) tout évènement de m est une projection d'un évènement de σ
   $$\phi_{3,2}(\sigma, m, L) = \forall e, \forall l \in L : m, l \in \lambda(e) \Rightarrow \exists e' : \sigma, l \in \lambda(e')$$

Nous somme maintenant en mesure d'énoncer l'équivalence entre le problème de Post et la satisfaction d'une formule MSO sur les pomsets.

Théorème 2.3.2 *Le problème de correspondance de Post* $(\alpha_1, \beta_1), \ldots, (\alpha_N, \beta_N)$ *a une solution si et seulement si la formule suivante est satisfiable :*

$$\begin{gathered}
\phi_1(\sigma_1) \wedge \phi_1(\sigma_2) \wedge \phi_1(m) \wedge \phi_1(p) \\
\wedge \phi_{2,1}(\sigma_1, (\alpha_i)_i) \wedge \phi_{2,2}(\sigma_1, (\alpha_i)_i) \wedge \phi_{2,3}(\sigma_1, (\alpha_i)_i) \\
\wedge \phi_{2,1}(\sigma_2, (\beta_i)_i) \wedge \phi_{2,2}(\sigma_2, (\beta_i)_i) \wedge \phi_{2,3}(\sigma_2, (\beta_i)_i) \\
\wedge \phi_{3,1}(\sigma_1, m, \sigma) \wedge \phi_{3,2}(\sigma_1, m, \sigma) \\
\wedge \phi_{3,1}(\sigma_2, m, \sigma) \wedge \phi_{3,2}(\sigma_2, m, \sigma) \\
\wedge \phi_{3,1}(\sigma_1, p, \cup_{i\in[1,N]}\{(_i,)_i\}) \wedge \phi_{3,2}(\sigma_1, p, \cup_{i\in[1,N]}\{(_i,)_i\}) \\
\wedge \phi_{3,1}(\sigma_2, p, \cup_{i\in[1,N]}\{(_i,)_i\}) \wedge \phi_{3,2}(\sigma_2, p, \cup_{i\in[1,N]}\{(_i,)_i\})
\end{gathered}$$

Preuve:
Dans un sens, si le problème de Post a une solution, nous pouvons construire le pomset associé et constater qu'il vérifie toutes les contraintes de la formule $MSO(\Sigma, \preceq)$.

Dans l'autre sens, une solution de la formule $MSO(\Sigma, \preceq)$ permet de reconstruire facilement une solution du problème de Post. □

Nous venons donc de montrer que la satisfaction d'une formule de la logique monadique du second ordre sur les pomsets est indécidable Le model-checking d'ensembles partiellement ordonnés est aussi un problème difficile. Il existe des résultats de décidabilité dans certaines classes de pomset. Nous les verrons dans le chapitre suivant.

Chapitre 3

Etude des modèles ayant une sémantique ordre partiel

Sommaire

Le modèle naturel pour décrire un comportement lorsqu'il s'agit d'un système séquentiel est un mot. Chaque lettre représente une action ou les valeurs des propriétés élémentaires de l'état d'un système. On parle de comportements linéaires dans le sens où la suite des

événements sont totalement ordonnés. Dans ce cadre le lien entre les langages de spécification et les modèles opérationnels (automates) est bien établi comme nous l'avons montré dans le chapitre précédent. Un concepteur dispose donc, d'un côté, d'outils simples et expressifs permettant de décrire le fonctionnement d'un système et de l'autre, de modèles d'exécution formellement définis et pour lesquels de nombreux problèmes de vérification sont décidables.

Dans le domaine des systèmes paralléles et répartis, le tableau est plus complexe. La plupart des modèles qui ont eté proposés ne sont pas assez expressifs car ils ne permettent pas d'exprimer certains aspects du parallélisme, ou inutilisables en pratique car de nombreux problèmes (de vérification) sont indécidables ou très peu efficaces, pour ces modèles. L'étude des modèles des théories décrivant le fonctionnement de systèmes concurrents a été motivée par les travaux de Petri qui a exposé dans [Pet62] un modèle basé sur la communication de systèmes séquentiels interconnectés. Il a en même temps présenté de nouveaux problèmes apparaissant quand on considère le parallélisme intrinsèque de tels systèmes. Dans beaucoup de modèle, l'opération de composition parallèle se base sur l'entrelacement (interleaving). Ainsi, si a et b sont des actions paralléles, leur modélisation conduit à exécuter a puis b ou b puis a et non a et b en même temps. Pour distinguer le non déterminisme du vrai parallélisme, il est nécessaire de pouvoir parler d'actions globales d'un système provenant de l'activité de ses composants agissant ensemble.

Dans ce contexte, les ensembles partiellement ordonnés (pomsets) constituent un modèle idéal pour modéliser un système sans en décrire explicitement les entrelacements produits par son exécution séquentielle. Ce modèle est compact et intuitif mais présente un inconvénient qui découle du fait que la vérification de systèmes avec des formules de logiques *ordre partiel* usuelles sur un système concurrent est complexe. Il est à noter que le problème de la satisfaction d'une formule est en général indécidable dans ce cadre. Il existe cependant des résultats de décidabilité pour certaines classes de modèles étendant ou se basant sur les pomsets. Dans ce chapitre, nous présentons ces principaux modèles et finirons à la fin par une discussion sur les apports de ces modèles avant de situer notre contribution dans le domaine.

3.1 Théorie des traces

En 1977 Mazurkiewicz [Maz77] a proposé un modèle de parallélisme consistant à considérer des événements comme des symboles. Ce modèle des traces est depuis plus de 30 ans un cadre formel fondamental de la recherche sur la modélisation et la vérification de systèmes distribués. Deux événements pouvant s'effectuer concurremment définissent un couple de symboles qui peuvent commuter. En effet, l'indépendance de certains événements dans un système concurrent permet d'avoir différentes suites d'événements pour représenter une seule et même exécution. Ainsi, un modèle fondé sur la représentation intégrale de ces suites d'événements équivalents n'est pas efficace. Le modèle des traces représente donc ces suites par des classes d'équivalence. Chaque classe représentera à elle seule un ensemble de comportements équivalents du système.

La théorie des traces offre deux manières de représenter ces classes d'équivalence, et ces manières diffèrent par leurs commodités selon l'utilisation qu'on veut faire de ces classes.

Le point de départ de la théorie des traces de Mazurkiewicz est la notion d'alphabet de concurrence.

3.1.1 Alphabet de dépendance

Soit Σ un alphabet fini dont les éléments appelés *actions* peuvent être notés par des symboles (comme a, b, c, $\cdots$ par exemple). L'idée intuitive de Mazurkiewicz est de doter Σ d'informations complémentaires concernant la façon dont certaines actions peuvent, ou ne peuvent pas, être exécutées en parallèle.

Un *alphabet de concurrence* ou de *dépendance* est une paire ordonnée (Σ, D) où $D \subseteq \Sigma \times \Sigma$ est une relation binaire symétrique et réflexive sur Σ c.à.d: pour tout couple d'actions (a, b), si $(a, b) \in D$ alors $(b, a) \in D$, et pour toute action a, nous avons $(a, a) \in D$. La relation D est appelée *relation de dépendance* dans Σ. La relation complémentaire $I = (\Sigma \times \Sigma) \setminus D$ est une relation symétrique et non réflexive appelée la *relation d'indépendance*. Intuitivement, les actions a et b sont concurentes (donc peuvent être exécutées en même temps) si et seulement si $(a, b) \in I$. Comme D et I sont symétriques, on pourra parfois désigner notre alphabet de concurrence par (Σ, I) (qui est en fait ici un alphabet d'indépendance) et donner pour I simplement l'une des deux relations $(a, b) \in I$ au lieu de $(a, b) \in I$ et $(b, a) \in I$.

Proposition 3.1.1 *L'union et l'intersection de relations de dépendance sont aussi des relations de dépendance.*

Preuve: Soit D_1 et D_2 deux relations de dépendance sur les alphabets Σ_1 et Σ_2. Il s'agit de montrer ici que les relations binaires $D_1 \cup D_2$ et $D_1 \cap D_2$ sont aussi des relations de dépendance sur les alphabets $\Sigma_1 \cup \Sigma_2$ et $\Sigma_1 \cap \Sigma_2$.

- Cas $D_1 \cup D_2$:
  - $x \in \Sigma_1 \cup \Sigma_2 \Rightarrow x \in \Sigma_1 \vee x \in \Sigma_2 \Rightarrow (x, x) \in D_1 \vee (x, x) \in D_2 \Rightarrow (x, x) \in D_1 \cup D_2$. La relation $D_1 \cup D_2$ est donc réflexive.
  - $(x, y) \in D_1 \cup D_2 \Rightarrow (x, y) \in D_1 \vee (x, y) \in D_2 \Rightarrow (y, x) \in D_1 \vee (y, x) \in D_2 \Rightarrow (y, x) \in D_1 \cup D_2$. La relation $D_1 \cup D_2$ est donc symétrique.

  La relation $D_1 \cup D_2$ est une relation binaire, réflexive et symétrique sur l'alphabet $\Sigma_1 \cup \Sigma_2$; elle est donc une relation de dépendance.

- Cas $D_1 \cap D_2$:
  - $x \in \Sigma_1 \cap \Sigma_2 \Rightarrow x \in \Sigma_1 \wedge x \in \Sigma_2 \Rightarrow (x, x) \in D_1 \wedge (x, x) \in D_2 \Rightarrow (x, x) \in D_1 \cap D_2$. La relation $D_1 \cap D_2$ est donc réflexive.
  - $(x, y) \in D_1 \cap D_2 \Rightarrow (x, y) \in D_1 \wedge (x, y) \in D_2 \Rightarrow (y, x) \in D_1 \wedge (y, x) \in D_2 \Rightarrow (y, x) \in D_1 \cap D_2$. La relation $D_1 \cap D_2$ est donc symétrique.

  La relation $D_1 \cap D_2$ est une relation binaire, réflexive et symétrique sur l'alphabet $\Sigma_1 \cap \Sigma_2$; elle est donc une relation de dépendance.

L'union et l'intersection de relations de dépendance sont donc des relations de dépendance. □

Après le rappel des notions d'alphabet de dépendance, on s'intéresse maintenant aux notions de traces et de systèmes de traces.

3.1.2 Traces

Étant donné un alphabet de concurrence (Σ, I), la relation d'indépendance I induit une relation d'équivalence notée $\sim_I$ (ou simplement $\sim$ s'il n'y a pas d'ambiguité) sur le monoïde Σ^* de telle manière que nous avons: $(a,b) \in I \Rightarrow xaby \sim xbay$ avec x et $y \in \Sigma^*$. Deux mots u et v sont équivalents si et seulement si il existe une suite de mots $(w_0, w_1, \ldots, w_n)$, avec $n \geq 0$, telle que $w_0 = u, w_n = v$, et pour tout $0 < i \leq n$ il existe $(a_i, b_i) \in I$ et deux mots x_i, y_i telles que $w_{i-1} = x_i a_i b_i y_i$ et $w_i = x_i b_i a_i y_i$. En d'autres termes, u, v sont équivalents si il existe une suite de commutations de lettres adjacentes.

Définition 3.1.1 *Soit (Σ, I) un alphabet de concurrence. Pour tout mot u de Σ^* on peut définir la classe d'équivalence $[u]_I \subseteq \Sigma^*$ qui correspond à l'ensemble $\{v \in \Sigma^* \mid u \sim v\}$. $[u]_I$ est alors appelé trace sur l'alphabet de concurrence (Σ, I).*

A défaut d'ambiguïté, nous utiliserons la notation $[u]$ à la place de $[u]_I$

Exemple 3.1.1 *Soit (Σ, I) un alphabet de concurrence tel que $\Sigma = \{a, b, c\}$ et $I = \{(a,b), (b,a)\}$. Une trace sur (Σ, I) est $[abcbac] = \{abcbac, abcabc, bacbac, bacabc\}$.*

On note $\mathbb{T}(\Sigma, I)$ l'ensemble de toutes les traces sur (Σ, I). $\mathbb{T}(\Sigma, I)$ est en fait le quotient de Σ^* par $\sim_I$. On l'appelle *monoïde de trace*. Il a comme élément neutre la trace vide $[\varepsilon]$

Pour toute trace $[u]$ sur un alphabet de concurrence (Σ, I), on note D_{u_i} (resp. I_{u_i}), l'ensemble des symboles de u qui sont dépendants (resp. indépendants) de u_i (le symbole de u placé à la i^{me} position). Nous avons donc:

$$D_{u_i} = \{u_j \in \Sigma \mid (u_i, u_j) \in D\}$$

.

3.1.3 Langage et automate de traces

On peut évidemment considérer des ensembles réguliers de traces et définir aussi des automates sur les traces afin de décrire des machines concurrentes et étendre les techniques de model-checking vues dans le chapitre précédent pour les ensembles de traces.

Définition 3.1.2 (Langages de traces) *Soit (Σ, I) un alphabet de concurrence. Pour tout langage de mots $L \subseteq \Sigma^*$, le langage de traces $[L]_I$ (ou plus simplement $[L]$) est défini par:*

$$[L]_I = \{[u]_I \mid u \in L\}.$$

Un langage de trace est donc un sous-ensemble de $\mathbb{T}(\Sigma, I)$.

La concaténation de langages de traces se définit comme à l'accoutumée: pour les langages de traces T et S, leur concaténation est le langage:

$$T.S = \{t.s \mid t \in T, s \in S\}.$$

Remarque: Pour décider si pour deux langages de traces $[L]$ et $[L']$, on a $[L] \subseteq [L']$, on peut tester si $L \subseteq [L']$. Pour tester si $[L] \cap [L'] = \emptyset$, on peut juste tester si $L \cap [L'] = \emptyset$ (ou l'inverse).

Définition 3.1.3 (Langages réguliers de traces) *Soit $L \subseteq \mathbb{T}(\Sigma, I)$. L est régulier si $L = [K]$ où $K \subseteq \Sigma^*$ est régulier. Plus précisément L est régulier s'il est définissable par une expression régulière sur $\mathbb{T}(\Sigma, I)$.*

On note $\mathrm{REG}(\mathbb{T}(\Sigma, I))$, l'ensemble des langages réguliers sur $\mathbb{T}(\Sigma, I)$. Une trace $[v]$ appartient à un langage régulier de traces $[L]$ s'il existe un mot $u \in L$ avec $u \sim_I v$.

On peut considérer des automates sur les traces que l'on appelle plus souvent $I-diamant$ et dont la définition formelle est la suivante:

Définition 3.1.4 *Soit (Σ, I) un alphabet de concurrence. Un automate $I-diamant$ est un automate fini $\mathcal{A} = (\Sigma, Q, \Delta, I, F)$ ayant en plus la propriété suivante: pour tout $a, b \in (\Sigma, I)$ si $q_0 \xrightarrow{a} q_1 \xrightarrow{b} q_2$ alors il existe q_1' $(\in Q)$ tel que $q_0 \xrightarrow{b} q_1' \xrightarrow{a} q_2$.*

Exemple 3.1.2 *Prenant l'alphabet d'indépendance $(\Sigma, I) = (\{a, b, c\}, \{(a, c), (c, a)\})$. Sur la figure 3.1, l'automate $\mathcal{A}_1$ n'est pas I-diamant car de l'état p, on peut tirer la suite d'actions ac, avec $(a, c) \in I$, mais pas la suite équivalente ca. Il n'est pas non plus clos par commutation parce que $aca \in L(\mathcal{A}_1)$, mais $aac \notin L(\mathcal{A}_1)$. Les automates $\mathcal{A}_2$ et $\mathcal{A}_3$ sont I-diamant.*

Un langage de trace est reconnaissable s'il est reconnu par un automate I-diamant. On note $\mathrm{REC}(\mathbb{T}(\Sigma, I))$, l'ensemble des langages reconnaissable sur $\mathbb{T}(\Sigma, I)$.

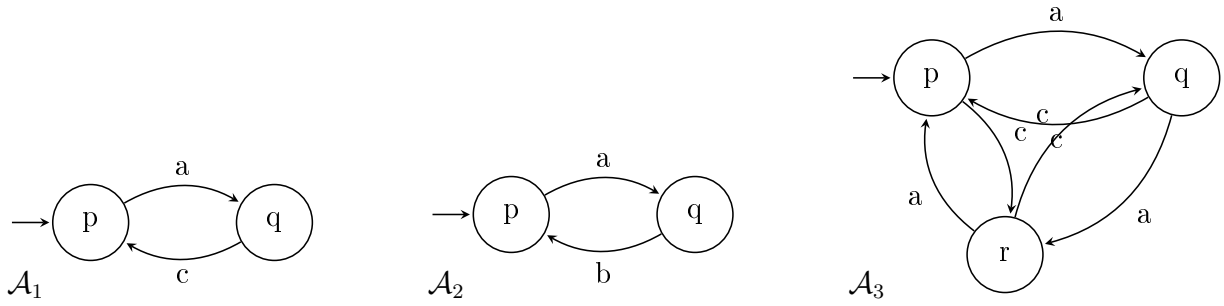


Figure 3.1: Trois automates I-diamant $\mathcal{A}, \mathcal{B}, \mathcal{C}$

Avec la propriété I-diamant on a si $u \sim_I v$ alors $q \xrightarrow{u} q'$ si et seulement si $q \xrightarrow{v} q'$. On peut donc simplifier la notation par $q \xrightarrow{[u]} q'$. En fait, pour un automate I-diamant $\mathcal{A}$, on a $L(\mathcal{A}) = [L(\mathcal{A})]$, c'est à dire que $L(\mathcal{A})$ est clos par commutation.

Les langages reconnaissables de trace et les réguliers de traces ne sont pas nécessairement égaux. Par exemple, si on prend le langage régulier $[(ab)^*]_I$ sur l'alphabet de concurrence (Σ, I) avec $\Sigma = \{a, b\}$ et $I = \{(a, b), (b, a)\}$, alors l'ensemble de traces reconnu par ce langage est $\{u \in \Sigma^* \mid |u|_a = |u|_b\}$, qui n'est pas un reconnaissable (de mots), donc pas non plus un reconnaissable de traces. Comme un automate I-diamant est un automate sur les mots, et qu'on a l'équivalence avec les réguliers sur les mots, on peut dire que les reconnaissables de traces sont inclus dans la classe des langages réguliers de traces c-a-d $REC(\mathbb{T}(\Sigma, I)) \subseteq REG(\mathbb{T}(\Sigma, I))$.

Ochmanski a cependant montré dans [Och85] qu'il existe une sous-classe syntaxique des expressions régulières de trace, appelée traces corationnelles, qui correspond aux reconnaissables. Le précédent exemple laisse voir que les boucles concernant un alphabet non connecté posent problème. Soit (Σ, D) un alphabet de dépendance. Un mot $u \in \Sigma$ est $D-$*connexe* si le graphe $(V, D|_{V\times V})$, dont les sommets V sont les lettres de u, est connexe [1].

Exemple 3.1.3 *Reprenant les exemples de la figure 3.1 sur l'alphabet d'indépendance* $(\Sigma, I) = (\{a, b, c\}, \{(a, c), (c, a)\})$, $\mathcal{A}_1$ *et* $\mathcal{A}_2$ *ne correspondent pas à des automates de mots D-connexes parce qu'ils autorisent des boucles de ac, avec* $(a, c) \in I$. *En revanche,* $\mathcal{A}_3$ *l'est puisque la seule boucle concerne les lettres* a, b *qui sont dépendants.*

Une expression régulière E est dite $(D-)$*corationnelle* si, pour chaque sous-expression e de E, tout $u \in L(e)$ est D-connexe. Le résultat d'Ochmanski concernant les corationnelles stipule que celles-ci forment un sous-ensemble strict des traces rationnelles et correspondent à l'ensemble des langages reconnaissables de traces.

3.1.4 Traces et Pomsets

Les traces ont plusieurs représentations équivalentes. Il existe notamment une équivalence bien connue entre trace et pomset [BK91].

Soit $[L]$ un langage de trace sur un alphabet de concurrence (Σ, D) et soit $u \in \Sigma^*$. Alors la trace $[u]$ est précisément l'ensemble des extensions linéaires $LE(t)$ d'un pomset $t = (E, \preceq, \lambda)$. Ce pomset t satisfait les propriétés suivantes:

P_1: pour tous les évènements $e_1, e_2 \in E$ avec $\lambda(e_1)D\lambda(e_2)$, nous avons $e_1 \preceq e_2$ ou $e_2 \preceq e_1$;

P_2: pour tous les évènements $e_1, e_2 \in E$ avec $e_1 \lessdot e_2$, nous avons $\lambda(e_1)D\lambda(e_2)$.

P_3: pour tous les événements e, $\downarrow e$ est fini.

Nous pouvons donner une manière de construire ce pomset de sorte qu'il respecte ces propriétés.

Définition 3.1.5 (Ψ_D) *Nous définissons par* $\Psi_D : \mathbb{T}(\Sigma, I) \longmapsto \mathbb{P}(\Sigma)$ *l'application qui à toute trace* $[u]$ *de l'alphabet de dépendance* (Σ, D) *associe son pomset équivalent* $\Psi_D([u])$ *tel que* $\Psi_D([u]) = (E_u, \preceq_u, \lambda_u)$ *où:*

- $E_u = \{0, \ldots, |u| - 1\}$;
- $\lambda_u(i) = u_i$;
- $\preceq_u$ *est la fermeture transitive de la relation* $\rightarrowtail_D = \{(i, j) \mid i \leq j \wedge (u_i, u_j) \in D\}$ *pour tout* i *et* $j \in [0, |n|[$, *c'est à dire* $i \rightarrowtail_D j$ *ssi* $i \leq j$ *et* $(u_i, uj) \in D$.

La figure 3.2 est un DAG de pomset de la trace de l'exemple 3.1.1.

[1] c-a-d quels que soient les sommets x et y de V, il existe une une suite d'arêtes permettant d'atteindre y à partir de x.

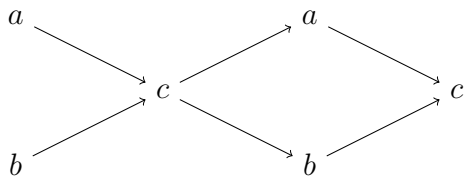


Figure 3.2: pomset représentant la trace de l'exemple 3.1.1

Bien que toute trace peut être représentée en un pomset équivalent, l'inverse n'est pas toujours vrai. Tout pomset n'est pas l'équivalent d'une trace de Mazurkiewicz. Il est simple de s'en apercevoir en prenant comme contre exemple un pomset avec auto-concurrence auquel cas on violerait la propriété $P1$.

3.1.5 Logique et vérification avec les traces

On peut définir des formules MSO sur les traces avec le même formalisme que la logique sur les pomsets donnée dans la section 2.3.3. Une trace satisfait une formule MSO si le pomset associé satisfait la formule MSO.

Proposition 3.1.2 *[EM93] Un langage de traces $L \subseteq \mathbb{T}(\Sigma, I)$ est MSO-définissable si et seulement si le langage de mots $[L]_I$ est définissable en MSO. Ainsi, la logique MSO sur les traces a le même pouvoir d'expression que les langages réguliers de traces.*

Il est aussi possible de définir sur les traces des logiques moins expressives comme la logique FO. Il a été énoncé là encore dans [EM93] que cette logique a le même pouvoir d'expression que les langages réguliers sans étoile de traces.

Des logiques temporelles peuvent aussi décrire des traces. Il existe deux approches pour ces logiques:

- Les logiques qui parlent d'états globaux du système, c'est-à-dire qui s'évaluent sur les préfixes d'une trace (ou configuration). Ces logiques sont appelées logiques temporelles globales. L'une des plus connue est $LTrL$ [TW02] qui permet l'utilisation de modalités dans le passé. Ces modalités ne rajoutent pas d'expressivité à LTL comme prouvé dans [DG02].

- Les logiques qui s'évaluent sur les événements des traces, dites logiques locales. Diekert et Gastin ont montré que ces logiques sont équivalentes à FO [DG06].

Ces deux approches sont équivalentes du point de vue de leur pouvoir expressif. Néanmoins les logiques globales ont une complexité plus grande et sont même souvent indécidable [Pen92, AP99]. Walukiewicz a montré dans [Wal98] que LTrL est non-élémentaire.

Avec les logiques locales, les problèmes de satisfiabilité et de model-checking associés sont en général décidables en PSPACE. On distingue en deux parties dans cette approche, en fonction de la sémantique choisie pour l'opérateur décrivant ce qui se passera après un événement. Dans le cas où l'opérateur choisi est existentiel (c'est à dire quand on peut

exprimer que des propriétés du type "il existe une chaîne de dépendance qui mène à un événement étiqueté par a") Alur et al. ont défini la logique TLC dont le model-checking et la satisfiabilité et sont dans PSPACE [APP95]. Dans le cas où l'opérateur est universel, Diekert et Gastin proposent la logique $LocTL(EX, U)$ [DG01] qui n'utilise certe que deux modalités EX et U ne permettant pas d'exprimer toutes les formules du premier ordre, mais dont le problème de satisfiabilité est PSPACE-complet. Gastin et Mukund ont par la suite proposé, dans [DG02], une logique qui modifie la modalité U de $LocTL(EX, U)$ afin d'être plus expressive tout en conservant la complexité.

Il a été montré enfin dans [GK03] (Gastin et Kuske) que ces problèmes de décision sur les traces pouvaient se réduire à la satisfiabilité et au model-checking de logiques temporelles définissables en MSO et restant dans PSPACE.

3.2 Les pomsets série-parallèles

Toujours dans l'optique de modéliser la concurence, Lodaya et Weil ont étendu la théorie classique des automates de mots aux langages de *pomsets série-parallèles* [LW98, LW00]. Ils se sont basés sur une classe d'automate qu'ils ont appelé *automates branchants* et ont prouvé des résultats analogues aux théorèmes de Kleene et de Büchi.

3.2.1 sp-pomset

Un pomset séries-parallèles ou *sp-pomset* pour abréger, peut être défini comme suit:

Définition 3.2.1 *Un sp-pomset est un pomset vide ou un pomset pouvant être obtenu à partir d'une suite finie de produits séquentiels et parallèles à partir de pomsets singleton.*

Nous avons défini ces opérations produits parallèles et séquentiels dans la section 2.3.2. Pour cette dernière, nous écrirons par moments $p_1.p_2$ ou p_1p_2 au lieu de $p_1 \bullet p_2$ pour le produit séquentiel de deux pomsets p_1 et p_2.

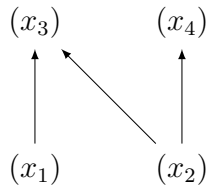


Figure 3.3: Pomset N

Alternativement, un pomset série-parallèle peut être défini en tant que *pomset N-free* [VTL79, Gra81]. Un pomset est N-free s'il ne contient pas le pomset $N = (E_N, \preceq_N)$ représenté dans la figure 3.3; plus formellement s'il ne contient pas 4 éléments distincts x_1, x_2, x_3, x_4 tels que les seules relations d'ordre sont: $x_1 \prec x_3, x_2 \prec x_3, x_2 \prec x_4$ (peu importe l'étiquette).

La figure 3.4 représente un exemple de pomset série-parallèle. Il peut s'écrire en $a.(b \parallel c).a$. Un pomset représentant un système de producteur-consommateur (cf 2.5) n'est par contre pas un pomset série-parallèle. On note $\mathbb{SP}(\Sigma)$ l'ensemble des pomsets série-parallèles

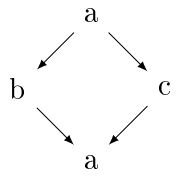


Figure 3.4: Exemple de sp-pomset

sur Σ.

$\mathbb{SP}(\Sigma)$ peut être généralisé en algèbre de série-parallèle [LW00]. Une algèbre de série-parallèle (ou *sp-algèbre*) est un ensemble S muni des opérations . et $\parallel$ où $(S, .)$ est un semi-groupe (c'est à dire que le produit séquentiel . est associatif) et $(S, \parallel)$ est un semi-groupe commutatif (le produit parallèle $\parallel$ est en plus commutatif). Dans le cas de cette généralisation, on peut remarquer que le pomset vide n'appartient pas à $\mathbb{SP}(\Sigma)$. En fait $\mathbb{SP}(\Sigma)$ est (isomorphe à) un algèbre de série-parallèle non vide.

Un terme d'un sp-algebre est appelé *sp-terme* et désigne donc par isomorphisme un sp-pomset. Un sp-terme est dit séquentiel s'il ne peut pas être écrit sous forme de produit parallèle, et parallèle s'il ne peut pas être écrit sous forme séquentielle. La largeur $wd(t)$ d'un sp-terme t est définie inductivement par:

- $wd(t) = 1$ si t est une lettre
- $wd(t)$=max$\{wd(t1), wd(t2)\}$ si $t = t1.t2$
- $wd(t)$=$wd(t1) + wd(t2)$ si $t = t1 \parallel t2$

On peut en d'autres termes définir $wd(t)$ comme la cardinalité maximum des antichaines du sp-pomset t. Dans l'exemple de la figure 3.4 $wd(t) = 2$.

3.2.2 Langage de séries-parallèles

Un langage de séries-parallèles ou *sp-langage* est un sous ensemble L de $\mathbb{SP}(\Sigma)$ ne contenant pas le pomset vide. Un sp-langage L est dit de largeur bornée s'il existe un $k \in \mathbb{N}$ tel que $wd(t) \leq k$ pour tout $t \in L$.

Lodaya et Weil ont introduit plusieurs classes d'expressions rationnelles pour des ensembles de sp-pomsets. Pour cela on considère un certain nombre d'opérations sur les ensembles de sp-pomsets. Soient $S, T \subseteq \mathbb{SP}(\Sigma)$, on définit:

$$S \cdot T := \{s \cdot t \mid s \in S, t \in T\},$$

$$S \parallel T := \{s \parallel t \mid s \in S, t \in T\},$$

$S + T := S \cup T$.

$S^* := \{s_1 \cdot s_2 \cdot \ \cdots \ s_n \mid n \geq 1, s_i \in S\}$:l'itération séquentielle (étoile de Kleene),

$S^{\Vdash} := \{s_1 \| s_2 \| s_3 \cdots \| s_n \mid n \geq 1, s_i \in S\}$:l'itération parallèle.

Un langage $L \subseteq \mathbb{SP}(\Sigma)$ est rationnel s'il peut être construit à partir de sous ensemble finis de $\mathbb{SP}(\Sigma)$ par les opérations $+, \cdot, \|, *$, et $\Vdash$. Il est faiblement rationnel si l'opération $\Vdash$ est appliquée aux langages de la forme $K \cdot L$, seulement. Il est enfin série-rationnel si il peut être construit à partir des lettres en utilisant des unions, des produits séquentiels, des produits parallèles et des itérations séquentielles, sans donc l'itération parallèle. Par exemple $(a \parallel a)^{\Vdash}$ est rationnel et non faiblement rationnel et $a^{\Vdash}$ est faiblement rationnel mais non séries-rationnelles.

Puisque dans la construction des langages séries-rationnelles les itérations parallèles ne peuvent pas apparaître, pour tout langage séries-rationnelles L il existe un $n \in \mathbb{N}$ avec $wd(t) \leq n$ pour tout $t \in L$, c'est-à-dire tout langage séries-rationnelles est de largeur bornée.

Il serait intéressant à ce stade de se demander s'il y a une notion de reconnaissabilité.

Dans le cadre algébrique, $REC(\mathbb{SP}(\Sigma))$ est bien défini, pour les sous-ensembles de Σ^+, c'est la notion usuelle.

Exemple 3.2.1 *$(ab)^{\Vdash}$ est reconnu par un morphisme dans le sp-algèbre de quatre éléments $\{a, b, c, 0\}$ où chaque produit séquentiel est 0, excepté pour $ab = c$; et chaque produit parallèle est 0 excepté pour $c \| c = c$.*

Les sp-langages de largeur bornée sont une classe "robuste" de *sp*-langages, dans le sens où ils ont une caractérisation algébrique: on peut décider si un langage reconnaissable L a une largeur bornée en inspectant son morphisme syntaxique (morphisme canonique dans la *sp*-algèbre reconnaissant L).

Lodaya et Weil ont montré dans [LW00] que si un langage $L \subseteq \mathbb{SP}(\Sigma)$ a une largeur bornée, alors $L \in REC(\mathbb{SP}(\Sigma))$ si et seulement si L est série-rationnelle.

Pouvons-nous trouver des caractérisations pour la reconnaissabilité sous forme d'automates? Ceci serait intéressant notamment pour établir la relation avec le théorème de Kleene.

3.2.3 Les automates de branchement

Les automates de branchement ont été introduits par Lodaya et Weil pour accepter des sp-pomsets:

Définition 3.2.2 *Un automate branchant est un tuple $\mathcal{A} = (Q, T_s, T_f, T_j, I, F)$ où:*

S est un ensemble fini d'états,

$I \subseteq Q$ et $F \subseteq Q$ sont respectivement des ensembles d'états initiaux et d'états acceptants,

$T_s \subseteq Q \times \Sigma \times Q$ est l'ensemble des transitions séquentielles,

$T_f \subseteq Q \times (Q \times Q)$ est l'ensemble des transitions d'embranchement, et

$T_j \subseteq (Q \times Q) \times Q$ est l'ensemble des transitions de jointure.

La définition initiale des automates de branchement dans [LW98] est légèrement différente de celle-ci vu qu'ils ont été introduits dans une forme légèrement plus générale permettant le branchement de plus de deux sous-processus. Mais la définition donnée ici, plus souvent utilisée par Kuske, a le même pouvoir d'expression.

Vu que les automates de branchement marchent sur les sp-pomsets, leurs exécutions peuvent être définies par induction sur la construction des sp-pomsets à partir d'un pomset singleton par produits séquentiels et parallèles.

Soient $p, r \in Q$ des états d'un automate de branchement $\mathcal{A}$ et soit a l'étiquette du pomset singleton de départ. Il y a une exécution de p à r sur a (dénoté $p \xrightarrow{a} r$)ssi $(p, a, r) \in T_s$. Maintenant soient s, t des sp-pomsets. Il y a une exécution $p \xrightarrow{s \cdot t} r$ ssi il existe un état $q \in Q$ et des exécutions $p \xrightarrow{s} q$ et $q \xrightarrow{t} r$. Il y a enfin une fonction $p \xrightarrow{s \| t} r$ ssi il existe des états $p_1, p_2, r_1, r_2 \in Q$, une transition d'embranchement $(p, (p_1, p_2)) \in T_f$, des exécutions $p_1 \xrightarrow{s} r_1$ et $p_2 \xrightarrow{t} r_2$, et une transition de jointure $((r_1, r_2), r) \in T_j$. Cette définition est visualisée dans la Figure 3.5.

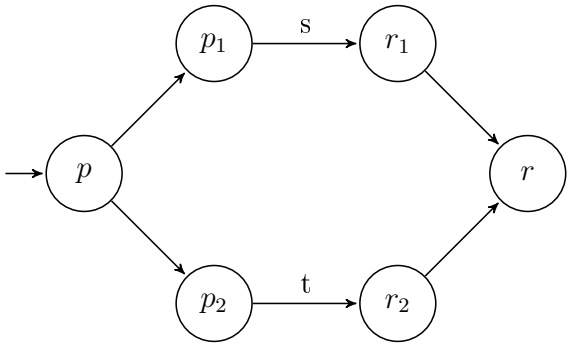


Figure 3.5: Une exécution pour $s \| t$

Un sp-pomset s est accepté par $\mathcal{A}$ ssi il existe $p \in I$ et $q \in F$ et une exécution $p \xrightarrow{s} q$. Par $L(\mathcal{A})$ nous dénotons l'ensemble des sp-pomsets acceptés par $\mathcal{A}$.

Lodaya et Weil ont montré les relations suivantes mettant en jeu les concepts introduits jusqu'ici:

Théorème 3.2.1 ([LW00]) *Soit $L \subseteq \mathbb{SP}(\Sigma)$. Alors les éléments suivants sont équivalents:*

1. *L est série-rationnelle.*
2. *L est algébriquement reconnaissable.*
3. *L peut être accepté par un automate de branchement.*
4. *L est de largeur bornée.*

A partir de là, il devient intéressant de compléter cette théorie concernant les langages reconnaissables des sp-pomsets dans l'esprit du théorème de Büchi, c'est-à-dire, voir comment rapporter la puissance expressive de la logique monadique de second ordre à cet automate de branchement.

3.2.4 Logique MSO sur les sp-pomsets

La logique MSO sur les sp-pomsets a été l'objet de nombreuses études menées par Kuske dans [Kus00b, Kus00a, Kus01, Kus02]. Nous donnons ici directement ses principaux résultats concernant la définissabilité logique et invitons le lecteur à consulter ses articles pour les détails et les preuves.

Tout d'abord, rappelons que syntaxiquement et sémantiquement, la logique MSO sur les sp-pomset rejoint celle des pomsets classiques.

Evidemment, tout ensemble fini de sp-pomsets peut être monadiquement axiomatisé. Au niveau des langages de sp-pomset qu'on a décrit précédemment, Kuske précise que tout sp-langage rationnel n'est pas forcément MSO-définissable. Ainsi, les langages rationnelles ne sont pas nécessairement monadiquement axiomatisables, par contre les faiblement rationnels le sont. D'où, la proposition suivante:

Proposition 3.2.1 ([Kus00b]) *Soit $L \subseteq \mathbb{SP}(\Sigma)$ un langage faiblement rationnel. Alors il existe une formule MSO ϕ telle que $L = \{t \in \mathbb{SP}(\Sigma) \mid t \models \phi\}$.*

Kuske démontre cette proposition en traduisant les différentes opérations des langages de série parallèles faiblement rationnels. Par exemple si S et T sont deux ensembles de sp-pomsets définis par les formules monadiques ϕ et ϕ', respectivement alors l'ensemble $S \parallel T$ consiste en tous les sp-pomsets satisfaisant:

$$\exists X(\forall x \forall y(x \in X \wedge y \notin X \rightarrow x \parallel y) \wedge \phi_{|X} \wedge \phi'_{|\bar{X}})$$

où $\phi_{|X}$ est la restriction de ϕ à l'ensemble X et $\phi'_{|\bar{X}}$ celle de ϕ' au complément de X. Il montre que l'opération $\Vdash$ est définissable du fait qu'elle est appliquée ici qu'aux langages de la forme $S = S_1 \cdot S_2$. Tout élément du langage itéré S est donc connecté (i.e. a un successeur ou un prédécesseur). Ainsi, la formule informelle

$$\varphi = \forall Z(Z \textit{ est connecte}' \rightarrow \phi_{|Z})$$

axiomatise $S^{\Vdash}$. Puisque dans la logique monadique de second ordre la fermeture transitive de la relation d'ordre $\preceq$ peut être définie, on peut exprimer qu'un ensemble est connecté. D'où le fait que $S^{\Vdash}$ peut être monadiquement défini à chaque fois que S est le produit de deux langages monadiquement axiomatisables.

Kuske a par la suite montré l'équivalence entre la définissabilité MSO et la reconnaissabilité par automate de branchement. Nous donnons ici son théorème en la matière.

Théorème 3.2.2 ([Kus00a]) *Soit $L \subseteq \mathbb{SP}(\Sigma)$ de largeur bornée. Alors les assertions suivantes sont équivalentes:*

- *L est monadiquement axiomatisable.*
- *L peut être accepté par un automate de branchement.*

Kuske dans [Kus00b] s'est aussi intéressé à la question de savoir si la correspondance de Büchi entre une logique monadique du second ordre sur les mots infinis et les ensembles reconnaissables peut être transférée à l'assemblage des sp-pomsets. Il a abouti à son principal résultat (Théorème 16), que nous reprenons ici.

Théorème 3.2.3 ([Kus00b]) *Soit $L \subseteq SP(\Sigma)$. Alors les propositions suivantes sont équivalentes:*

- *L est série-rationnelle.*
- *L est MSO-definissable et de largeur bornée.*
- *L est reconnaissable et de largeur bornée.*

Il apparaît au vu de tous ces résultats que dans le cas général la logique MSO n'a pas la même expressivité que les langages de serie parallèle. MSO est en fait moins expressive que les automates de branchement. Ce fait a motivé les tout récents travaux de Bedon dans [Bed13]. Il y définit une nouvelle logique appelée *P-MSO logic*, qui est en fait du MSO enrichi de l'arithmétique de Presburger, et montre que cette logique a le même pouvoir d'expression que les automates de branchement.

3.3 Message Sequence Chart

Les MSC (ou "Message Sequence Charts") ont rencontré un intérêt considérable pendant cette dernière décennie. C'est un langage de spécification graphique normalisé par l'ITU ("International Telecommunication Union") dès 1993 dans la recommandation Z.120. Les MSC permettent de spécifier des propriétés de systèmes de processus qui communiquent via des canaux point-à-point.

La définition par l'ITU d'opérateurs de composition (séquentielle et parallèle), afin de pouvoir exprimer des ensembles infinis de msc (HMSC) et l'intégration des travaux de Mauw et Reniers [MR94, Ren98] sur les aspects sémantiques des MSC ont donné le dernier document de la recommandation en 1999 [ITU99].

L'UML (Unified Modelling Langage) de l'OMG dispose aussi d'un diagramme proche des MSC (diagramme de séquence) et depuis sa version 2.0 en 2003 [OMG03] intégre d'autres diagrammes où on retrouve le noyau syntaxique des HMSC (diagramme de communication, diagramme de temps).

3.3.1 MSC

Pour définir les MSC nous commençons par formaliser la notion d'entité communicante souvent appelée processus (ou agent, localité,...) et de canal de communication. Pour cela, fixons un ensemble $\mathcal{P}$ de processus communicant par messages via des cannaux FIFO. On

considère $\mathcal{C}$ comme un ensemble de messages ou d'actions internes. Chaque processus de $\mathcal{P}$ peut effectuer des actions qui peuvent être:

- $p!q$ signifiant "p envoie un message à q"
- $p?q$ signifiant "p réceptionne un message provenant de q"
- $p(a)$ signifiant "p effectue l'action interne interne a"

Si on veut en plus s'intéresser aux messages envoyés ou reçus on utilise les notations $p!q(m)$ ou $p?q(m)$ avec $m \in \mathcal{C}$.

L'alphabet global des actions est ainsi défini par $\Sigma_{\mathcal{P}} = \{p!q(a), p?q(a), , p(a) \mid p, q \in \mathcal{P}, p \neq q, a \in \mathcal{C}\}$.

Chaque paire de processus dispose d'un canal pour communiquer. On note par $Ch = \{(p,q) | p \neq q\}$ l'ensemble des canaux (avec p, $q \in \mathcal{P}$).

Définition 3.3.1 *Un MSC M est un sextuplet $M = (\mathcal{P}, \mathcal{C}, E, \lambda, m, <)$ avec*

- *$\mathcal{P}$ est l'ensemble fini des processus*
- *$\mathcal{C}$ est l'ensemble d'étiquettes des messages.*
- *$E = \bigcup_{p \in \mathcal{P}} E_p$ avec E_p désignant l'ensemble des événements du processus p.*
- *$\lambda : E \longrightarrow \Sigma_{\mathcal{P}}$ est la fonction d'étiquetage, qui a un évènement associe son type. On partitionne $E = S \uplus R \uplus L$ en ensemble des envois S, réceptions R et évènements locaux L.*
- *$m : S \longrightarrow R$ est une bijection qui associe chaque envoi à sa réception. Si $m(e) = e'$, alors il existe $p, q \in \mathcal{P}$ et $a \in \mathcal{C}$ tels que $\lambda(e) = p!q(a)$ et $\lambda(e') = q?p(a)$.*
- *$< \subseteq E \times E$ est une relation sur les évènements, formée*
  - *d'un ordre total $<_p$ sur E_p, pour tout processus $p \in \mathcal{P}$*
  - *et pour tout $e, e' \in E$, $m(e) = e'$ implique que $e < e'$.*

  On demande d'autre part que la fermeture réflexive et transitive de $<$ soit acyclique (et donc qu'elle induise un ordre partiel sur E).

On note par $\mathbb{MSC}(\Sigma_{\mathcal{P}})$ l'ensemble des MSC sur l'alphabet $\Sigma_{\mathcal{P}}$ et l'ensemble des processus $\mathcal{P}$. Avec $p, q \in \mathcal{P}$, nous notons par $E_{p!q} \subseteq S$, l'ensemble des envois de message de p à q et $E_{p?q}$ l'ensemble des réceptions de messages provenant de q par p.

On peut voir alternativement un MSC comme un pomset.

Définition 3.3.2 *Soit $M = (\mathcal{P}, \mathcal{C}, E, \lambda, m, <)$ un MSC sur $\mathbb{MSC}(\Sigma_{\mathcal{P}})$. M est un pomset $(E, \preceq, \lambda)$ satisfaisant les conditions suivantes:*

(1) Chaque relation $\prec$ restreinte à $E_p \times E_p$ correspond à $<_p$ (est donc un ordre total),

(2) $p \neq q \Rightarrow |E_{p!q}| = |E_{q?p}|$ *pour tout* $p, q \in \mathcal{P}$

(3) La relation $\preceq$ *est la fermeture reflexive, et transitive de la relation* $<$.

Dans les diagrammes, les évènements d'un MSC sont présentés en ordre visuel. Les évènements de chaque processus sont arrangés dans une ligne verticale et les relations $<$ sont représentés en des flèches orientées horizontales. Nous illustrons cela avec l'exemple suivant.

Exemple 3.3.1 *Dans le MSC de la figure 3.6,* $\mathcal{P} = \{p, q, r\}$; $E_p = \{e_1\}$,$E_q = \{e_2, e'_2\}$,$E_r = \{e_3\}$; *les relations d'ordre sont* $e_1 < e_2$, $e_2 <_q e'_2$, $e'_2 < e_3$. *La fonction d'étiquetage* λ *est définie comme suit:* $\lambda(e_1) = p!q$, $\lambda(e_2) = q?p$, $\lambda(e'_2) = q?r$, $\lambda(e_3) = r!q$.

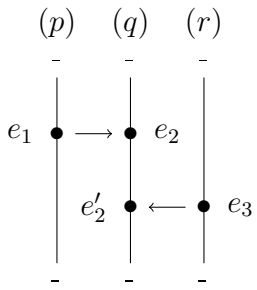


Figure 3.6: Un exemple de MSC

Pour un MSC $M \in \mathbb{MSC}(\Sigma_{\mathcal{P}})$, $LE(M)$ désigne son ensemble de linéarisation définie comme sur les pomsets c'est-à-dire les ordres totaux sur E compatible avec $\preceq$. Pour l'exemple 3.3.1, on a $LE(M) = \{p!q\ q?p\ r!q\ q?r,\ \ p!q\ r!q\ q?p\ q?r,\ \ r!q\ p!qq?p\ q?r\}$. Ce sont des mots de Σ^*. Il est possible de construire un MSC à partir de l'une de ses linéarisations.

Un langage de MSC est un sous ensemble $L \subseteq \mathbb{MSC}(\Sigma_{\mathcal{P}})$. Pour un langage de MSC L, on note par $LE(L)$ l'union des linéarisations des MSC $M \in L$.

Le modèle d'automate souvent défini pour les MSC est celui des automates communicants appelé aussi CFM (Communicating Finite-state Machine) [BZ83] ou MPA (Message Passing Automata) dans [MKS00]. Ce modèle consiste en un ensemble fini de systèmes séquentiels à événements discrets qui communiquent par échange de messages asynchrones, par l'intermédiaire de canaux de communication pas forcément bornés. Nous en donnons ici la définition formelle.

Définition 3.3.3 *Un automate communicant est une structure* $\mathcal{A} = (\{\mathcal{A}_p\}_{p \in \mathcal{P}}, \mathcal{M}, Q_I, Q_F)$ *où:*

- $\mathcal{M}$ *est un alphabet de message.*
- *pour tout processus* $p \in \mathcal{P}$ *l'automate* $\mathcal{A}_p$ *est un n-uplet* $(S_p, \Sigma_p, \longrightarrow_p)$ *consistant en un ensemble d'états locaux* S_p, *un alphabet* Σ_p *d'actions du processus p, une relation de transition locale (intégrant* $\mathcal{M}$*):* $\longrightarrow_p \subseteq S_p \times \Sigma_p \times \mathcal{M} \times S_p$.

- $Q_I, Q_F \subseteq \prod_{p\in\mathcal{P}} S_p$ *sont respectivement les ensembles finis d'états initiaux et finaux globaux.*

Le calcul de $\mathcal{A}$ *commence dans un état initial* $q^0 \in Q_I$. *Les actions de* $\mathcal{A}_p$ *sont soit des actions locales soit des envois ou réceptions de messages (avec les mêmes notations que pour les MSC).*

Définition 3.3.4 *Un automate communicant* $\mathcal{A} = (\{\mathcal{A}_p\}_{p\in\mathcal{P}}, \mathcal{M}, Q_I, Q_F)$, $A_p = (S_p, \Sigma_p, \longrightarrow_p)$ *est dit:*

- *"fini" si pour tout* $p \in \mathcal{P}$, S_p *est fini,*
- *"localement acceptant" si pour tout* $p \in \mathcal{P}$ *il existe un ensemble* $F_p \subseteq S_p$ *tel que* $Q_F = \prod_{p\in\mathcal{P}} F_p$,
- *déterministe si pour tout* $p \in \mathcal{P}$, $\longrightarrow_p$ *satisfait les conditions suivantes:*
  - *Si* $(s_0, p!q, m_1, s_1) \in \longrightarrow_p$ *et* $(s_0, p!q, m_2, s_2) \in \longrightarrow_p$ *alors* $m_1 = m_2$ *et* $s_1 = s_2$.
  - *Si* $(s_0, p?q, m_1, s_1) \in \longrightarrow_p$ *et* $(s_0, p?q, m_2, s_2) \in \longrightarrow_p$ *alors* $s_1 = s_2$.

La figure 3.7 représente un automate communicant fini non déterministe et localement acceptant.

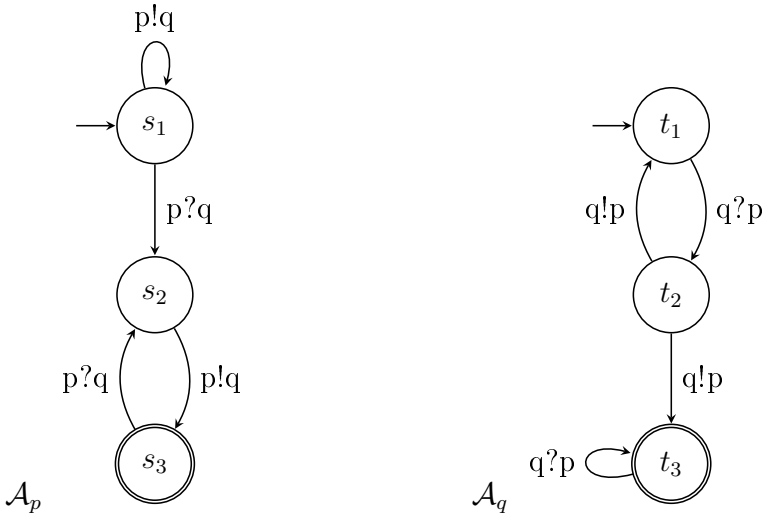


Figure 3.7: Un exemple d'automate communicant

On comprend un automate communicant en tant qu'un réseau de machines qui communiquent via des canaux infinis FIFO. La relation de transition locale $\longrightarrow_p$ montre comment le processus p envoie et reçoit des messages. La transition $(s, p!q, m, s')$ spécifie que quand p est à l'état s, il peut envoyer le message m à q et aller à l'état s'. Le message m est alors ajouté à la fin de la file du canal (p, q). Si c'est une transition $(s, p?q, m, s')$, dans ce cas la première occurence de m à partir de la tête du file du canal est supprimé.

L'ensemble des états globaux de $\mathcal{A}$ est donné par $\prod_{p\in\mathcal{P}} S_p$. Un état global est donc un $|\mathcal{P}| - uplet$ d'états locaux. Une configuration est un couple (g,χ) où g est un état global et $\chi : Ch \longrightarrow \mathcal{M}^*$ est l'état du canal de communication dénotant les messages en attente d'être lus. La configuration initiale de $\mathcal{A}$ est (g_0,χ_ε) où $g_0 \in Q_I$ est un état initial global et χ_ε exprime le vide de tous les canaux de Ch. L'ensemble des configurations finales est $Q_F \times \{\chi\}$. L'ensemble des configurations accessibles de $\mathcal{A}$ est noté $Conf_{\mathcal{A}}$.

La relation de transition globale $\longrightarrow \subseteq Conf_{\mathcal{A}} \times Conf_{\mathcal{A}}$ est définie inductivement par:

- $(g_0,\chi_\varepsilon) \in Conf_{\mathcal{A}}$.
- Si $(g,\chi) \in Conf_{\mathcal{A}}$, (g',χ') est une configuration et $(s_p, p!q, m, s'_p) \in \longrightarrow_p$ telle que:
  - $r \neq p$ implique $s_r = s'_r$ pour tout $r \in \mathcal{P}$.
  - $\chi'((p,q)) = \chi((p,q)).m$ et pour $c \neq (p,q), \chi'(c) = \chi(c)$.

  Alors $(g,\chi) \xrightarrow{p!q} (g',\chi')$ et $(g',\chi') \in Conf_{\mathcal{A}}$.
- $(g,\chi) \in Conf_{\mathcal{M}}$, (q), (g',χ') est une configuration et $(s_p, p?q, m, s'_p) \in \longrightarrow_p$ telle que:
  - $r \neq p$ implique $s_r = s'_r$ pour tout $r \in \mathcal{P}$.
  - $\chi((q,p)) = m.\chi'((q,p))$ et pour $c \neq (q,p), \chi'(c) = \chi(c)$.

  Alors $(g,\chi) \xrightarrow{p?q} (g',\chi')$ et $(g',\chi') \in Conf_{\mathcal{A}}$.

Comme d'habitude on note par $L(\mathcal{A})$ le langage des exécutions reconnues par $\mathcal{A}$.

$L(\mathcal{A})$ n'est pas forcément régulier. Considérons par exemple, CFM pour le système $producteur - consommateur$ où le producteur p envoie un nombre arbitraire de messages au consommateur q. Du moment que nous pouvons réordonner les actions $p!q$ de sorte qu'elles soient exécutées avant toutes les actions $q?p$ correspondantes, la file d'attente du canal (p,q) peut grossir de façon arbitraire. Donc, l'ensemble des configurations accessibles de ce système n'est pas borné et le langage correspondant n'est pas régulier.

Pour un ensemble d'entiers $B \in \mathbb{N}$, on dit qu'une configuration (g,χ) du CFM $\mathcal{A}$ est B-bornée si pour tout canal $c \in Ch$ on a $|\chi(c)| \leq B$. On dit qu'un CFM $\mathcal{A}$ est un automate B-borné si chacune de ses configurations accessibles est B-bornée.

On dit qu'une configuration d'un CFM est un *blocage* si aucun chemin acceptant ne part de cette configuration. Un CFM est *sans blocage* si aucune de ses configurations accessibles n'est un blocage, i.e. si tous les états globaux accessibles sont co-accessibles. Il est bien connu que les CFM sans blocage sont strictement moins expressifs que les CFM avec blocage et savoir si un CFM A est sans blocage est indécidable [BZ83].

3.3.2 HMSC

La méthode standard pour décrire des scénarios de communication multiple est de générer des collections de MSC par les voies des Hierarchical Message Sequence Charts (HMSC). Ces structures sont aussi appelées MSG [AP99, BKSS06], "MSC spécifications" [BAL97], ou "MSC-graphs" [Mus99, AEY05].

Un HMSC est un graphe où chaque sommet est étiqueté par un MSC. Les bords représentent l'opération de concaténation de MSC. La collection des MSC représentée par un HMSC consiste en tous ces MSC obtenus en traçant un chemin dans le HMSC à partir du sommet initial jusqu'au sommet final et en concaténant les MSC qui sont rencontrés le long du chemin.

On définit tout d'abord l'opération principale qui est la concaténation de MSC. Soient $M_1 = (\mathcal{P}, \mathcal{C}_1, E_1, \lambda_1, m_1, <_1)$ et $M_2 = (\mathcal{P}, \mathcal{C}_2, E_2, \lambda_2, m_2, <_2)$ deux MSC concernant le même ensemble de processus $\mathcal{P}$. Alors la concaténation de M_1 et M_2, notée $M_1.M_2$ ou M_1M_2, est un MSC $M = (\mathcal{P}, \mathcal{C}_2 \cup \mathcal{C}_2, E_1 \uplus E_2, \lambda_1 \uplus \lambda_2, m_1 \uplus m_2, <)$, où $<$ est une relation acyclique donnée par:

$$<=<_1 \cup <_2 \cup\{(e_1, e_2) \in E_1 \times E_2 \mid Processusde(e_1) = Processusde(e_2)\}$$

.

Intuitivement, on concatène M_1 et M_2 en collant les processus communs à M_1 et M_2 ensemble, les évènements de M_1 intervenant avant ceux de M_2.

Un *atome* est un MSC qui n'est pas produit de deux MSCs. L'ensemble des atomes est noté $A - \mathbb{MSC}(\Sigma_{\mathcal{P}})$. On peut voir facilement que tout MSC se décompose en un produit d'atomes et que cette décomposition est unique à permutation près d'atomes indépendants. Ainsi, MSC est (isomorphe à) un monoïde de traces, $\mathbb{T}(A - \mathbb{MSC}(\Sigma_{\mathcal{P}}), I)$.

Les MSC pouvant aussi être vus comme des pomsets, nous pouvons là encore donner une définition de la concaténation en terme de pomset. Soient deux MSC $M_1 = (E_2, \preceq_1, \lambda_1)$ et $M_2 = (E_2, \preceq_2, \lambda_2)$ avec E_1 et E_2 disjoints. On peut définir $M_1.M_2$ comme un pomset $M = (E, \preceq, \lambda)$ où:

- $E = E_1 \cup E_2$
- $\lambda = \lambda_1 \cup \lambda_2$
- $\preceq$ est la plus petite relation d'ordre partiel sur E contenant $\preceq_1$ et $\preceq_2$ et satisfaisant la condition: si $e \in E_{1p}$ et $e' \in E_{2p}$ alors $e \preceq e'$.

L'ensemble des langages rationnels de MSC sur les processus $\mathcal{P}$ est la plus petite classe qui contient les singletons MSC sur $\mathcal{P}$, close par union, concaténation et étoile de Kleene.

Définition 3.3.5 *Un HMSC est une expression rationnelle de* $\mathbb{MSC}(\Sigma_{\mathcal{P}})$.

Il existe d'autres définitions équivalentes en terme d'automate.

Définition 3.3.6 *Un HMSC sur un ensemble de processus* $\mathcal{P}$ *est un tuple* $H = (Q, T, q_0, F)$ *où* Q *est un ensemble fini d'états,* $T \subseteq Q \times \mathbb{MSC}(\Sigma_{\mathcal{P}}) \times Q$ *un ensemble fini de transitions étiquetées par des MSC finis,* $q_0 \in Q$ *est l'état initial et* $F \subseteq V$ *l'ensemble des états finaux.*

Dans d'autres définitions formelles de la littérature les éléments de $\mathbb{MSC}(\Sigma_{\mathcal{P}})$ étiquettent les états plutôt que les transitions. C'est le cas dans l'exemple de la figure 3.8.

La notation $q_i \xrightarrow{M} q_j$ signifie que $(q_i, M, q_j) \in T$. Un chemin $\rho = q_1 \xrightarrow{M_1} q_2 \xrightarrow{M_2} ... \xrightarrow{M_n} q_{n+1}$ est dit acceptant (ou exécution) si $q_1 = q_0$ et $q_n \in F$. Le MSC généré par ρ est

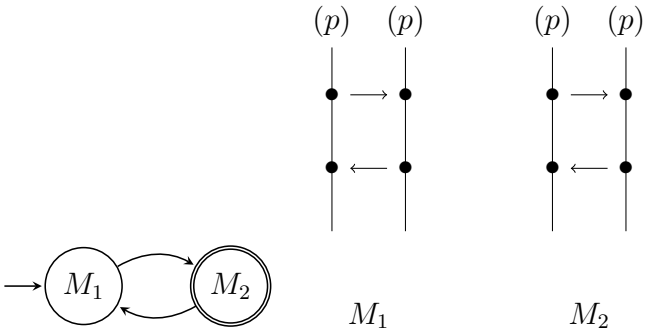


Figure 3.8: Un exemple de HMSC avec les MSC en étiquette d'état

$M = M_1.M_2.....M_n$. On dit dans ce cas (par abus de langage) que M est accepté par H. Le langage $L(H)$ d'un HMSC H est l'ensemble des MSC acceptés par H.

L'ensemble des linéarisations d'un HMSC H, noté $LE(H)$ est l'union des ensembles de linéarisations des MSC acceptés par H. Cet ensemble $LE(H)$ n'est pas rationnel en général. Pour preuve il suffit de considérer le cas $H = M^*$ où M est le MSC de la figure 3.9 qui est un exemple basique du système producteur/consommateur.

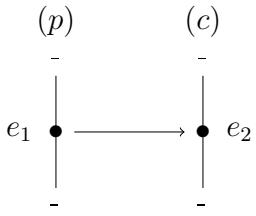


Figure 3.9: MSC pour le système producteur/consommateur

$L(H)$ n'est pas forcément régulier. Pour voir cela, nous posons $L = \{u \in \{p!q, r!s\}^* \mid u_{p!q} = u_{r!s}\}$ qui n'est pas un langage de mot régulier (car les langages réguliers sont clos sous des projections arbitraires).

3.3.3 Vérification avec les MSC et HMSCs

Les MSC peuvent s'utiliser aussi bien pour modéliser des protocoles que pour spécifier des propriétés à vérifier. Un certain nombre d'études ont été réalisé dans le domaine de la vérification pour les MSC pris individuellement en terme de leurs sémantiques et propriétés [LL95, AHP96].

Il est possible d'utiliser la logique MSO pour décrire les propriétés des MSC. Sa définition est la même que la logique sur les pomsets décrite dans la section 2.3.3 à laquelle on peut ajouter des formules atomiques de la forme $x \rightarrow y$ (signifiant x envoie un message à y) et $x \preceq_p y$ (signifiant que x et y sont des événements du processus p et x précède y dans l'ordre total des événements du processus p).

Des logiques temporelles pouvant décrire des MSC existent aussi. C'est le cas de la logique TLC considérée dans [Pel00]. Mais ces logiques peuvent en réalité être vues comme des sous-classes, de MSO d'où le fait que MSO est plus puissant.

Le champ de recherche le plus vaste a consisté à vérifier les propriétés spécifiées en tant que HMSC. Nous allons brièvement donner quelques uns de ces résultats plus populaires ici.

Etant donné deux HMSC G, H, savoir si on a $L(G) \cap L(H) = \varnothing$ ou $L(G) \subseteq L(H)$ sont des problèmes de model-checking. Le cas du model-checking négatif (le plus facile) est déjà indécidable.

Beaucoup de problèmes de décision sont indécidables avec les HMSC. Une bonne partie des résultats d'indécidabilité sont présentés dans [MPS98, Mus99, MP99]. Ceci a conduit à considérer des sous-classes de HMSC avec lesquelles les problèmes habituels sont décidables.

Henriksen, Mukund, Kumar et Thiagarajan ont considéré les HMSC réguliers [HMKT00]. Un HMSC H est dit *régulier* si $LE(H)$ est rationnel. Les problèmes de décision comme le model-checking sont bien entendu décidables pour cette classe de HMSC ($LE(H)$ étant un langage régulier de mots). Par contre on ne peut pas décider si un HMSC est régulier.

Le model-checking est notamment connu comme étant décidable pour la classe des HMSCs *globalement coopératifs* [GMSZ02]. Un HMSC est globalement coopératif si toutes ses étiquettes sont des MSCs connexes et pour tout chemin en boucle $\rho = q_1 \xrightarrow{M_1} q_2 \xrightarrow{M_2} ... \xrightarrow{M_n} q_1$ le MSC généré $M_1.M_2.....M_n$ est connexe. Un MSC est connexe si son graphe de communication (graphe dirigé dont les sommets sont les processus et les arêtes de la forme $p \to q$ si et seulement si p envoie un message à q) est faiblement connexe.

Les résultats suivants ont été démontrés dans [GMSZ02].

- Savoir si deux HMSCs globalement coopératifs ont une exécution commune est PSPACE-complet.
- Décider de l'inclusion des ensembles d'exécutions de deux HMSCs globalement coopératifs est EXPSPACE-complet.
- Quand on fixe le nombre de processus, la complexité des problèmes de model-checking pour les HMSCs localement coopératifs est PSPACE-complet.

Les autres classes de HMSC à considérer sont celles des HMSC à *canaux bornés* et les HMSC reconnaissables.

Un HMSC H est dit à canaux bornés s'il existe un entier $b \in \mathbb{N}$ tel que :

$$\forall M \in L(H), \forall (p,q) \in Ch, \forall N \; prefixe \; de \; M, (|N|_{p!q} - |N|_{q?p}) \leq b$$

. $|N|_{p!q}$ et $|N|_{q?p}$ désignent respectivement le nombre d'envois de messages de p à q et le nombre de réceptions par q des messages de p dans N.

Vu que l'ensemble des MSCs finis est isomorphe au monoïde de traces $\mathbb{T}(A-\mathbb{MSC}(\Sigma_{\mathcal{P}}), I)$, on peut définir comme à l'accoutumée la notion de langage reconnaissable. Un langage de MSC L est reconnaissable s'il existe un morphisme f de MSC dans un monoïde fini tel que $L = f^{-1}(f(L))$. Un HMSC H est dit reconnaissable si $L(H)$ est reconnaissable.

Du fait des résultats d'indécidabilité sur les traces, on peut déduire que savoir si un HMSC est reconnaissable est indécidable.

On doit à R. Morin [Mor01] les résultats suivants:

- On peut décider si un HMSC est à canaux bornés.
- Un HMSC est régulier si et seulement s'il est reconnaissable et à canaux bornés.

Genest, Muscholl et Kuske ont étendu, dans [GKM06], les résultats de Enriksen et al. dans [HMK+05] (par extension du théorème de Kleene-Büchi) sur l'equivalence entre langage régulier de MSC, HMSC globalement coopératifs, et modèles d'exécution, donnés en terme de d'automates communicants. Les équivalences suivantes ont ainsi été démontrées pour un langage L de MSC (universellement) bornés.

- L est régulier.
- L est le langage d'un CFM.
- L est le langage d'une formule MSO.
- L est le langage d'un HMSC globalement coopératif.

3.4 Les Automates Cellulaires Asynchrones pour pomsets

Dans une autre approche, il a été défini des automates pouvant accepter des pomsets ou classes de pomsets. Dans ce domaine nous pouvons citer les Automates Cellulaires Asynchrones de pomsets et les automates P-asynchrones.

La notion d'Automates Cellulaires Asynchrones (ACA) a été introduite originellement par Zielonka pour les traces. Vu que les traces peuvent être considérées comme des pomsets vérifiants des conditions particulières, Droste, Gastin et Kuske ont ensuite généralisé cette notion pour reconnaître les langages reconnaissables de pomsets et ont introduit dans ce sens les Automates Cellulaires Asynchrones pour les pomsets ($\vec{\Sigma} - ACA$). Là on s'intéresse uniquement aux pomsets sans auto-concurrence. Ils ont aussi présenté les relations entre les logiques FO, MSO et les ACA. L'essentiel de ces travaux se trouvent dans [DGK00]. Nous en donnons ci-après un résumé.

3.4.1 Présentation des $\vec{\Sigma} - ACA$

Soient $\Sigma_1, \Sigma_2, ..., \Sigma_n$ des alphabets deux à deux disjoints, et $\mathcal{P} = \{1, ..., n\}$ un ensemble d'entiers. On pose $\Sigma = \Sigma_1 \cup, ..., \cup \Sigma_n$. Nous pouvons voir $\mathcal{P}$ comme un ensemble de processus séquentiels et $\Sigma_1, ..., \Sigma_n$ comme l'ensemble des actions de ces processus séquentiels. Soit $p : \Sigma \longrightarrow \mathcal{P}$ fonction associant chaque lettre $a \in \Sigma$ à son processus $p(a) \in \mathcal{P}$, i.e. $a \in \Sigma_{p(a)}$.

Définition 3.4.1 *Un $(\Sigma_1, \Sigma_2, ..., \Sigma_n) - pomset$ ou $\vec{\Sigma} - pomset$ est un pomset $t = (E, \preceq, \lambda)$ où $\lambda^{-1}(\Sigma_i)$ est totalement ordonné pour tout $i \in \mathcal{P}$.*

Cette définition implique que deux événements concurrents ne sont jamais associés au même Σ_i dans un $\vec{\Sigma} - pomset$. On note par $\mathbb{P}(\vec{\Sigma})$ l'ensemble des $\vec{\Sigma} - pomset$.

Définition 3.4.2 *Un* $(\Sigma_1, \Sigma_2, ..., \Sigma_n) - ACA$ *(ou* $\vec{\Sigma} - ACA$*) est un tuple* $\mathcal{A} = ((Q_i)_{i\in\mathcal{P}}, (\delta_{a,J})_{a\in\Sigma, J\subseteq\mathcal{P}}, F)$ *où:*

. *Pour tout* $i \in \mathcal{P}, Q_i$ *est un ensemble fini d'états locaux au processus* i*.*

. *Pour tout* $a \in \Sigma$ *et* $J \subseteq \mathcal{P}, \delta_{a,J} : \prod_{i\in J} Q_i \longrightarrow 2^{Q_{p(a)}}$ *est la fonction de transition locale.*

. $F \subseteq \bigcup_{J\subseteq\mathcal{P}} \prod_{i\in J} Q_i$ *est un ensemble d'états acceptant (finaux).*

L'automate est déterministe si toutes les fonctions de transition sont déterministes, i.e si $|\delta_{a,J}((q_i)i \in J)| \leq 1$ pour tout $a \in \Sigma, J \subseteq \mathcal{P}$ et $q_i \in Q_i$ pour $i \in J$.

Un $\vec{\Sigma} - ACA$ accepte un $\vec{\Sigma} - pomset\ t = (E, \preceq, \lambda)$ comme suit. Le $\vec{\Sigma} - ACA$ consiste en n processus locaux dont les états locaux sont Q_i. Chaque évènement x change l'état de son processus $p(\lambda(x))$. Pour chaque évènement, il y a (au moins) deux modes de d'exécution. Soit l'évènement lit à partir des évènements qu'il suit immédiatement, notés $R^- - mode$, soit il lit à partir de tous les évènements exécutés avant, notés $R^+ - mode$. En d'autres termes plus formels:

- $R^-(x) := p(\lambda\{y \in E \mid y \lessdot x\}) = p(\lambda(max(\Downarrow x)))$
- $R^+(x) := p(\lambda\{y \in E \mid y \prec x\}) = p(\lambda(\Downarrow x))$

Concernant les critères d'acceptation, soit nous considérons les états qui exécutent au moins une action comme états d'acceptation (les états sont isomorphes aux processus qui exécutent les actions), notée F^+, soit ceux qui exécutent au moins un évènement maximal (nous pouvons car les actions sont partiellement ordonnées), noté F^-.

On considère pour la suite $\alpha, \beta \in \{+, -\}$. Pour un $\vec{\Sigma} - ACA\ \mathcal{A}$, le langage de $\vec{\Sigma} - pomsets$ accepté par $\mathcal{A}$ dans le mode $R^\alpha F^\beta$, dénoté $R^\alpha F^\beta(\mathcal{A})$ est l'ensemble de tous les $\vec{\Sigma} - pomsets$ $t \in \mathbb{P}(\vec{\Sigma})$ qui admettent une R^α-exécution F^β-réussie (acceptée).

3.4.2 Logique MSO sur les $\vec{\Sigma} - ACA$

Après la formalisation des $\vec{\Sigma} - ACA$ et de leur mode d'acceptation de pomsets, la relation avec la logique MSO a été établie. C'est bien entendu $MSO(\Sigma, \preceq)$ (MSO sur les pomsets) qui est considérée ici. C'est d'abord la MSO-definissabilité des langages acceptés par les $\vec{\Sigma} - ACA$ (déterministe ou non) qui a été montrée.

Théorème 3.4.1 ([DGK00]) *Soit* $\mathcal{A}$ *un* $\vec{\Sigma} - ACA$*, alors on peut construire une formule MSO* ϕ *existentielle de sorte que le langage de* ϕ *soit le même que celui de* $\mathcal{A}$*; i.e.* $L(\phi) = R^\alpha F^\beta(\mathcal{A})$*.*

La réciproque de ce théorème n'est pas toujours vraie. Elle l'est que pour une sous-classe spéciale de $\vec{\Sigma} - pomsets$: les $CROW - pomsets$.

Définition 3.4.3 *Un* $CROW-pomset$ *est un* $\vec{\Sigma}-pomset\ t = (E, \preceq, \lambda)$ *qui satisfait l'axiome CROW ("Concurrent Read and Exclusive Owner Write") qui est: pour tout* $x, y, z \in E$*,*

$$x \lessdot y, x \prec y \text{ et } y \parallel z \implies p(\lambda(x)) \neq p(\lambda(z))$$

.

Un CROW-pomset est en fait un $\vec{\Sigma}$-pomset pour lequel pour tout x, y, z si $x \lessdot y$, y parallèle à z et $x \prec y$, le processus sur lequel x écrit est différent de celui de z. Cela signifie que si y et z sont concurrents, si z écrit sur la mémoire du processus i, y ne lira pas à partir de ce processus.

On note par $\mathbb{CROW}(\vec{\Sigma})$ l'ensemble des $CROW - pomsets$.

Théorème 3.4.2 ([DGK00]) *Soit ϕ une formule MSO sur $\vec{\Sigma}$. Il existe un $\vec{\Sigma} - ACA$ déterministe $\mathcal{A}$ sur $\mathbb{CROW}(\vec{\Sigma})$ tel que:*

$$L(\phi) \cap \mathbb{CROW}(\vec{\Sigma}) = R^{-}F^{\beta}(\mathcal{A})$$

Droste, Gastin et Kuske ont ensuite généralisé ces deux théorèmes pour monter les équivalences finales entre les $\vec{\Sigma} - ACA$ et MSO. Ce corollaire est énoncé par un théorème.

Théorème 3.4.3 ([DGK00]) *Soit $L \subseteq \mathbb{CROW}(\vec{\Sigma})$. Les assertions suivantes sont équivalentes:*

- *L est MSO-définissable,*
- *Il existe un $\vec{\Sigma} - ACA$ $\mathcal{A}$ sur $\mathbb{CROW}(\vec{\Sigma})$ tel que $L = R^{\alpha}F^{\beta}(\mathcal{A})$.*

Ces résultats tirent leur intérêt du fait qu'ils ouvrent la voie au model-checking des systèmes distribués abstraits décrits en CROW-pomsets.

Une autre classe de $\vec{\Sigma}$-pomset a aussi été définie pour intégrer les problèmes de décision.

Définition 3.4.4 *Soient $t = (E, \leq, \lambda)$ un $\vec{\Sigma}$-pomset, k un entier positif et $C_\ell \subseteq E$ pour $1 \leq \ell \leq k$. Le tuple $(C_1, ..., C_k)$ est un k-couvrant de t si*

1. *C_ℓ est une séquence pour $1 \leq \ell \leq k$*
2. *$E = \cup_{\ell \in [k]} C_\ell$*
3. *pour tout $x, y \in E$ avec $x \prec y$ il existe $\ell \in [k]$ avec $x, y \in C_\ell$*

Un k-pomset est un $\vec{\Sigma}$-pomset qui a un k-couvrant.

Soit $\mathbb{KP}(\vec{\Sigma})$ l'ensemble des k-pomsets sur $\vec{\Sigma}$. Le premier résultat établit que $MSO(\Sigma)$ est équivalent à $\vec{\Sigma} - ACA$ non déterministes au sein de la classe des k-pomsets.

Théorème 3.4.4 ([DGK00]) *Soit $L \subseteq \mathbb{KP}(\vec{\Sigma})$. Les assertions suivantes sont équivalentes:*

- *L est définissable en $MSO(\Sigma)$.*
- *$L \in R^{\alpha}F^{\beta}(\mathbb{KP}(\vec{\Sigma}))$*

Finalement, les auteurs établissent qu'on peut décider de deux $\vec{\Sigma} - ACA$ donnés $\mathcal{A}_1$ et $\mathcal{A}_2$ si $L(\mathcal{A}_1) = L(\mathcal{A}_2)$. Ainsi donné un $\vec{\Sigma} - ACA$, on peut décider si il reconnait l'ensemble de tous les k-pomsets (universalité) ou ne reconnait rien (vide).

Remarque: Il est aussi établi dans [DGK00] une construction permettant d'associer les ensembles de CROW-pomsets aux traces Mazurkiewicz (page 16). Ceci donne la possibilité d'utiliser les résultats sur les traces pour encore résoudre des problèmes de décidabilité.

Malgré les résultats positifs très intéressants que présentent les $\vec{\Sigma} - ACA$, il y a des résultats négatifs soulignés même dans l'article [DGK00]. En général, les deux modes d'exécution, $R^- F^\beta$ et $R^+ F^\beta$, sont incomparables pour les $\vec{\Sigma} - ACA$ non déterministes. De plus, pour $n \geq 2$, l'ensemble $R^\alpha F^\beta(\vec{\Sigma})$ n'est pas clos par complément.

Il existe d'autres modèles d'automates pouvant acceptés des pomsets sans auto-concurrence. Les automates P-asynchrones ont été défini par Arnold dans [Arn91]. Ils sont une généralisation des automates asynchrones des traces. Kuske a montré dans son habilitation [Kus00a] (page 78) que le pouvoir d'acceptation des automates P-asynchrones est inclus dans celui des $\vec{\Sigma} - ACA$.

3.5 Conclusion et Discussion

Nous avons présenté dans ce chapitre différents modèles permettant de modéliser des systèmes distribués et ayant une sémantique ordre partiel. Les travaux sur ces modèles mettent souvent l'accent sur les aspects de reconnaissabilité ou de régularité permettant du coup de répondre à une difficulté qu'on rencontre avec les pomsets contrairement aux mots, qui est le manque d'ensemble fini d'opérateurs pouvant tous les générer.

Durant cette étude nous nous sommes intéressés tant à leur aspect de modélisation que les possibilités qu'ils offrent pour la vérification de systèmes concurrents. Dans le cadre de la vérification, on est souvent heurté à des problèmes d'indécidabilité du fait notamment que le model-checking est difficile pour les modèles ordre partiel. De ce fait, les modèles étudiés ont plutôt tendance à limiter le pouvoir d'expression de la logique employée ou le pouvoir d'expression des modèles même, pour obtenir des résultats de décidabilité. Cependant il n'existe que peu ou pas à notre connaissance d'outils implémentant directement ces résultats.

Les traces de Mazurkiewicz permettent de modéliser des systèmes parallèles grâce à l'utilisation d'alphabet de dépendance. Cette théorie n'est pas suffisamment puissante pour décrire certains paradigmes de la concurrence comme celui du producteur-consommateur. Des extensions des monoïdes de traces ont été proposées pour intégrer de tels problèmes. Par exemple dans [BG95] on considère des quotients du monoïde libre par des congruences qui préservent l'image commutative des mots et on etablit ensuite des conditions nécessaires et des conditions suffisantes sur les congruences pour que leurs classes d'équivalence soient représentables par ordres partiels. Il s'avère aussi que l'approche synchrone des systèmes distribués sur laquelle sont basées les traces est inappropriée lorsqu'on est dans un paradigme de communication par passage de message.

Au niveau de la vérification, il n'existe pas directement d'outils implémentant les résultats théoriques des traces, cependant on peut noter l'existence d'outils basés sur les réseaux de

Petri sains et les produits synchronisés d'automates, qui sont eux des modèles en relation avec les traces [Hoo94]. Ces outils n'utilisent pas cependant des logiques ordre partiel. Les pomsets série-parallèle sont une classe de pomsets (l'une des rares, parmi les modèles, à intégrer les pomsets avec auto-concurrence) qui ont la particularité d'être $N - free$ et donc construit avec des opérations permettant de former des ensembles réguliers. De ce fait une extension du théorème de Büchi y a été étudiée [Kus00b] en intégrant la logique MSO. Cependant, de par nos recherches, son application au domaine de la modélisation et vérification de système n'est toujours pas étudiée à notre connaissance mis à part des études très peu connues et très spécifiques dans un domaine comme dans [INB99].

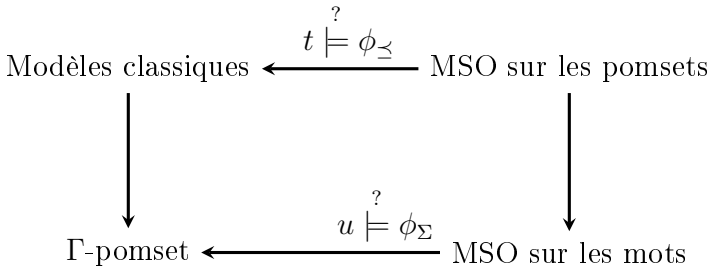
Les Message Sequence Chart permettent de modéliser des systèmes communicants, à l'aide d'expressions rationnelles. Ce sont des modèles matures et bien connus et diverses outils de modélisation de système avec des MSC existent. Au niveau vérification il y a des résultats d'indécidabilité vis à vis du model-checking classique (du style LTL) et des résultats théoriques de décidabilité en MSO comme nous l'avons montré dans le chapitre. Là encore il n'y a pas d'implémentation dans des outils de vérification.

Le constat est le même pour les Automates Cellulaire Asynchrones de pomsets. Ces derniers permettent théoriquement de faire de la vérification en utilisant la logique MSO sur les mots mais ils n'opèrent que sur les pomsets sans auto-concurrence.

Notre étude nous a permis d'autre part de nous rendre compte que les problèmes de satisfaction de formule logique sur les pomsets est en général indécidable.

Nous voulons proposer un modèle qui, à la fois, ramènerait le problème de vérification avec les logiques ordre partiel à un problème de vérification avec des logiques sur les mots (qui est décidable), et permettrait la traduction des modèles classiques étudiés et de constituer un framework commun pour une implémentation d'un outil de vérification.

Le schéma ci-dessous décrit le principe de notre approche et résume notre problématique.



Savoir s'il existe un pomset t qui satisfait une formule $\phi_{\preceq}$ de $MSO(\Sigma, \preceq)$ est en général indécidable. Certains modèles classiques étudiés dans ce chapitre donnent un certain nombre de résultat sur la vérification. Nous proposons de ramener ce problème à un problème de décidabilité sur les mots en utilisant donc la logique $MSO(\Sigma)$. Pour cela nous introduisons un nouveau modèle qui sera en même temps basé sur les mots et capable de représenter des pomsets. Nous appelons ce modèle (Γ-Pomset), et nous le présentons dans le chapitre suivant.

Chapitre 4

Γ-Pomsets

Sommaire

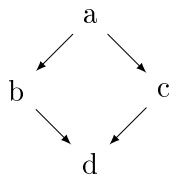
Dans ce chapitre, nous présentons notre modèle Γ-Pomset qui est une approche de modélisation d'un système parallèle basée sur les ordres partiels. Nous avons proposé ce modèle pour permettre la description d'un comportement distribué en terme de mot afin de faciliter la vérification. Après la définition formelle du modèle, nous donnerons sa sémantique par rapport aux pomsets traditionnels et des exemples de modélisation de paradigme du parallélisme. Mais tout d'abord, remettons tout à plat et présentons intuitivement notre idée de base.

4.1 Concept général

Dans le domaine de la vérification formelle, les comportements d'un système sont généralement modélisés par des mots finis ou infinis (voir section 2.1). Chaque lettre du mot, prise d'un alphabet donné, représente une action ou les valeurs des propriétés élémentaires de

l'état d'un système. Pour de tels systèmes, on parle de comportements linéaires ou séquentiels dans le sens où la suite des événements est totalement ordonnée. L'opération naturelle pour ces comportements séquentiels est la *concaténation*. Elle permet de lier deux tâches successives et modélise donc (dans le mot *..ab..*) le fait que le système peut débuter une tâche b dès qu'une autre tâche a prend fin. On parle aussi de relations de *causalité* entre événements et on note $a \rightarrow b$; l'événement "fin de a" étant la cause de l'événement "début de b". Dans un comportement linéaire, un événement ne peut être la cause que d'un seul autre événement.

Pour modéliser non seulement des comportements séquentiels, mais aussi des comportements incluant des tâches concurrentes, c'est à dire pouvant être exécutées en paralléles, nous pouvons nous baser sur les ordres partiels, l'ordre total n'étant plus approprié. C'est ainsi que les ensembles partiellement ordonnés étiquetés, pomsets ont été proposés (voir section 2.3). Dans ces structures, intervient une nouvelle opération, *l'opération parallèle* qui permet de modéliser le fait que deux tâches b et c sont incomparables dans le sens où elles peuvent être exécutées dans n'importe quel ordre après la fin d'une tâche a et avant le début d'une tâche d.



On peut naturellement constater que dans de tels systèmes un événement peut être la cause de plusieurs autres. L'ensemble des exécutions possibles ne se résume pas à un seul mot mais un ensemble de mots, pour notre exemple, *abcd* et *acbd*. On parle donc d'ensemble de *linéarisations* $LE(t)$ pour un pomset t comme on l'a vu précedemment. Cette notion de linéarisation peut nous permettre de considérer alors un pomset comme une extension de la notion de mot. Ainsi nous pouvons nous demander si les concepts de la théorie des mots et des langages, très largement répandus dans la littérature peuvent être généralisés sur les pomsets via les linéarisations. Mais nous savons qu'un ensemble de linéarisation ne permet pas toujours de retrouver les causalités réelles de départ entre les événements donc le pomset de départ. C'est le cas si dans l'exemple précédent nous remplaçons l'action c par l'action b. Nous nous trouverions dans un cas de pomset avec *auto-concurrence* où les deux événements étiquetés b ne sont pas causalement liés entre eux. L'ensemble de linéarisation se résumerait à $\{abbd\}$. Ce qui ne permet pas de retrouver l'odre partiel initial vu que d'autres pomsets peuvent avoir le même ensemble de linéarisation; $a \longrightarrow b \longrightarrow b \longrightarrow d$ par exemple.

On sait que pour certaines classes de pomsets telles celles présentées dans le chapitre précédent des équivalences avec les mots ont été établies. Une question serait de se demander si on peut trouver une généralisation pour tous les pomsets.

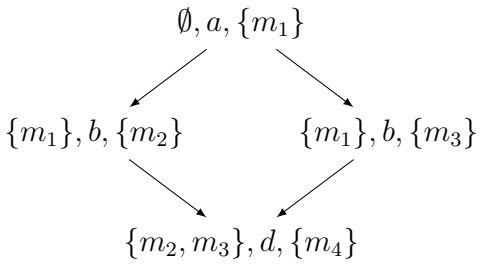
Vu qu'un pomset représente directement les liens de causalité entre événements, si nous trouvons un mécanisme permettant de retrouver à coup sûr ces causalités dans une linéarisation on peut retrouver le pomset de départ. Une idée simple serait de prendre une linéarisation et d'enrichir chacune de ses occurrences par des symboles pouvant renvoyer à un événement prédécesseur immédiat; c'est à dire la cause. Pour cela nous fixons un alphabet

et décidons d'associer deux ensembles de symboles à chaque événement i du pomset:

- un ensemble β_i permettant de marquer (noter) l'événement. Cet ensemble peut être vu comme une marque servant à identifier l'événement. Il peut être vu également comme un ensemble de *post-conditions*.

- un ensemble α_i regroupant l'ensemble des marques associées aux événements précédant immédiatement l'évènement considéré. Il est perçu en fait comme un ensemble de *pré-conditions*.

Pour illustrer cela, nous reprenons l'exemple précédent du pomset avec auto-concurrence en faisant figurer les marques des ensembles α_i et β_i.



La représentation partant d'une linéarisation donne la chaîne suivante:

$$\langle \emptyset, a, \{m_1\}\rangle\langle\{m_1\}, b, \{m_2\}\rangle\langle\{m_1\}, b, \{m_3\}\rangle\langle\{m_2, m_3\}, d, \{m_4\}\rangle$$

.

Cette chaîne peut être vue comme un mot où chaque élément (lettre) est un triplet de l'alphabet $2^{\Gamma} \times \Sigma \times 2^{\Gamma}$. Nous appellerons ces types de mots Γ-Pomsets .

Nous avons vu dans nos recherches que Twan Basten avait utilisé une représentation similaire dans [Bas97] en tant que format d'entrée pour faire du parsing de pomset. Mais il n'existe pas à notre connaissance de définition formelle de ce format pouvant permettre de l'utiliser en tant qu'outil de modélisation et de vérification de système. Dans la suite nous allons formellement présenter le modèle et ses possibilités d'utilisation dans le domaine de la vérification de systèmes parallèles.

4.2 Formalisation des Γ-Pomsets

Dans cette section, nous présentons notre modèle de base pour décrire les liens de causalités entre évènements dans un système parallèle. Ce modèle est une structure pouvant être vu comme un mot encodant des ensembles de pomsets. Cette structure s'inspire à la fois de la notion de linéarisation de pomset et de la relation de prédécesseur immédiat entre événements.

4.2.1 Définition des Γ-Pomsets

Nous allons maintenant définir un Γ-Pomset dans sa forme la plus générale avant de formaliser comment une telle structure peut permettre de générer des ensembles partiellement ordonnés.

Dans ce qui suit, nous prenons Σ comme un alphabet fini.

Définition 4.2.1 (Γ-Pomset) *Soit* Γ *un ensemble fini, éventuellement vide, de symboles appelés marques. Un* Γ*-Pomset* γ *est un mot fini ou infini* $\gamma = \langle \alpha_0, a_0, \beta_0 \rangle \cdots \langle \alpha_n, a_n, \beta_n \rangle \ldots$ *sur l'alphabet* $2^\Gamma \times \Sigma \times 2^\Gamma$.

Chaque élément $\gamma(i)$ d'un Γ-Pomset est donc triplet composé de trois parties: un sous-ensemble α_i de Γ que nous appelons marques α, un élément de l'alphabet Σ et un autre sous-ensemble β_i de Γ que nous appelons marques β.

Par simplicité d'écriture, pour un élément $\gamma(i)$, si sa marque α (resp. β) se résume à un singleton, nous écrivons $\langle m, a_i, \beta_i \rangle$ (resp. $\langle \alpha_i, a_i, m \rangle$) au lieu de $\langle \{m\}, a_i, \beta_i \rangle$ (resp. $\langle \alpha_i, a_i, \{m\} \rangle$).

Nous notons aussi par $\mathbb{P}_\Gamma(\Sigma)$ l'ensemble des Γ-Pomsets de $2^\Gamma \times \Sigma \times 2^\Gamma$. Dans tout ce qui suit Γ désignera toujours un ensemble de marques.

4.2.2 Langage de Γ-Pomsets

Les Γ-Pomsets sont des mots. De ce fait les théories classiques sur les mots et langages présentés en section 2.1 leur sont applicables. En particulier, on peut considérer des langages de Γ-Pomsets , former des expressions régulières ou des automates de Γ-Pomset . Un langage de Γ-Pomsets dénote un ensemble de Γ-Pomset donc un sous-ensemble de $\mathbb{P}_\Gamma(\Sigma)$. L'ensemble des expressions régulières $REG(\mathbb{P}_\Gamma(\Sigma))$ est le plus petit langage $L \subseteq \mathbb{P}_\Gamma(\Sigma) \cup \{., *, +, \varepsilon, \emptyset\}$ qui satisfait :

- $\varepsilon \in L$, $\emptyset \in L$ et $\gamma_i \in L$ pour γ_i un élément de l'alphabet $2^\Gamma \times \Sigma \times 2^\Gamma$.
- Si γ et γ' sont dans L alors $\gamma.\gamma'$ $\gamma + \gamma'$ et γ^* sont dans L.

Une expression régulière de Γ-Pomset renvoie à un langage L défini inductivement par:

- $L(\emptyset) = \emptyset$;
- $L(\varepsilon) = \{\varepsilon\}$;
- $L(\gamma_i) = \gamma_i$ pour $\gamma_i \in 2^\Gamma \times \Sigma \times 2^\Gamma$;
- $L(\gamma.\gamma') = L(\gamma).L(\gamma')$;
- $L(\gamma + \gamma') = L(\gamma) \cup L(\gamma')$;
- $L(\gamma^*) = L(\gamma)^*$;

Un automate de Γ-Pomset que nous appellerons $\Gamma-$*Automate* est un tuple $\mathcal{A} = (\Sigma, \Gamma, Q, \delta, I, F)$ où:

- Σ et Γ sont respectivement des alphabets d'actions et de marques
- Q est un ensemble d'états
- I et F désignent respectivement l'ensemble des états initiaux et l'ensemble des états finaux.
- $\delta : Q \times 2^{\Gamma} \times \Sigma \times 2^{\Gamma} \mapsto 2^{Q}$ est une fonction de transition

En matière de définissabilité logique nous pouvons aussi naturellement étendre le théorème de Büchi sur les Γ-Pomsets . D'où la proposition suivante:

Proposition 4.2.1 *Soit Γ un ensemble de marques et Σ un alphabet, tout langage régulier L de Γ-Pomsets est MSO definissable.*

Nous étudierons plus en détail l'application de la logique MSO sur les Γ-Pomsets dans la section 6.1.

4.2.3 Relation entre Γ-Pomsets et pomsets

Comme nous l'avons mentionné précédemment, les Γ-Pomsets seront utilisés ici pour générer des ensembles de pomsets et donc des relations d'ordre partiel. La construction de tels ensembles demande de formaliser la relation entre nos structure Γ-Pomset et les pomsets classiques. Pour cela, nous définissons la relation de couverture concernant les éléments d'un Γ-Pomset avant de parler de dépliage de Γ-Pomset .

Définition 4.2.2 (Γ-couverture $\rightarrowtail_{\Gamma}$) *Soit γ un Γ-Pomset tel que $\gamma = \langle \alpha_0, a_0, \beta_0 \rangle \dots \langle \alpha_n, a_n, \beta_n \rangle$. Pour i et j des entiers appartenant à $[0, n]$ on dit que j couvre i et on écrit $i \rightarrowtail_{\Gamma} j$ si et seulement si:*

$$i < j \wedge \exists m \in \Gamma : m \in \beta_i \wedge m \in \alpha_j \wedge (\forall k \in]i, j[: m \notin \beta_k).$$

$\rightarrowtail_{\Gamma}$ est donc une relation binaire interne à un Γ-Pomset et qui associe deux éléments d'un Γ-Pomset par leur position.

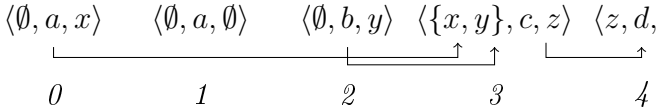
La définition 4.2.3 explique comment extraire des ensembles partiellement ordonnés depuis un Γ-Pomset donné.

Définition 4.2.3 (Dépliage de Γ-Pomset – Ψ_{Γ}) *Soit $\gamma = \langle \alpha_0, a_0, \beta_0 \rangle \dots \langle \alpha_n, a_n, \beta_n \rangle$, un Γ-Pomset de $\mathbb{P}_{\Gamma}(\Sigma)$. Nous définissons $\Psi_{\Gamma} : \mathbb{P}_{\Gamma}(\Sigma) \longmapsto \mathbb{P}(\Sigma)$ comme une application telle que $\Psi_{\Gamma}(\gamma)$ est le pomset $(E, \preceq, \lambda)$ défini comme suit:*

- $E = \{0, \dots, n\}$;
- $\lambda(i) = a_i$;
- $\preceq$ *est la fermeture reflexive et transitive de la relation de couverture $\rightarrowtail_{\Gamma}$ sur γ.*

Le dépliage consiste donc à extraire un pomset à partir d'un Γ-Pomset . Nous allons illustrer ce procédé par l'exemple suivant.

Exemple 4.2.1 *Avec* $\Gamma = \{x, y, z\}$ *et* $\Sigma = \{a, b, c, d\}$ *prenons le* Γ*-Pomset* $\gamma = \langle\emptyset, a, x\rangle\langle\emptyset, a, \emptyset\rangle\langle\emptyset, b, y\rangle\langle\{x, y\}, c, z\rangle\langle z, d, \emptyset\rangle$. *Les positions sont alors réparties comme suit:*
0 : $\langle\emptyset, a, x\rangle$
1 : $\langle\emptyset, a, \emptyset\rangle$
2 : $\langle\emptyset, b, y\rangle$
3 : $\langle\{x, y\}, c, z\rangle$
4 : $\langle z, d, \emptyset\rangle$
Avec la définition 4.2.2 on a les relations: $0 \rightarrowtail_\Gamma 3$, $2 \rightarrowtail_\Gamma 3$, $3 \rightarrowtail_\Gamma 4$. *Pour faciliter la compréhension, de manière plus visuelle on peut réécrire* γ *en dessinant les relations* Ψ_Γ.



Le Γ*-Pomset* γ *induit donc le pomset* $\Psi_\Gamma(\gamma) = (E, \preceq, \lambda)$ *tel que:*

- $E = [0, 4]$*;*
- $\lambda(0) = a, \lambda(1) = a, \lambda(2) = b, \lambda(3) = c, \lambda(4) = d$*;*
- $0 \preceq 3, 0 \preceq 4, 2 \preceq 3, 2 \preceq 4$.

Le DAG de ce pomset est représenté sur la figure 4.1

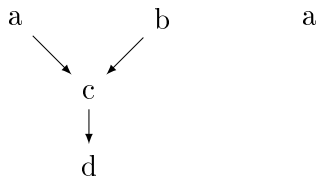


Figure 4.1: Pomset de l'exemple 4.2.1

Il est important de noter qu'en réalité plusieurs Γ-Pomsets peuvent induire le même pomset. En effet, notre définition du dépliage et celle de la relation de couverture $\rightarrowtail_\Gamma$ n'imposent pas d'avoir des marques β différentes pour chaque élément du Γ-Pomset . Nous pouvons donc réutiliser des marques ce qui nous permet potentiellement de réduire la taille de l'ensemble Γ. Ce qui fait que pour l'exemple précédent, avec $\gamma = \langle\emptyset, a, x\rangle\langle\emptyset, a, \emptyset\rangle\langle\emptyset, b, y\rangle\langle\{x, y\}, c, x\rangle\langle x, d, \emptyset\rangle$ on peut générer le même pomset réduisant ainsi Γ à $\{x, y\}$ en réutilisant la marque x. Il en découle la définition suivante.

Définition 4.2.4 (Degré-Γ d'un pomset) *Un pomset* t *admet un degré-*Γ *borné si* $\Psi_\Gamma^{-1}(t) \neq \emptyset$ *avec* $\Gamma = [0, k]$ *pour* k *un entier positif. Le degré-*Γ *du pomset* t *est donné par* $min\{k : \Gamma = [0, k]$ *et* $\Psi_\Gamma^{-1}(t) \neq \emptyset\}$.

On vient ainsi de formaliser le procédé permettant d'extraire un pomset depuis un Γ-Pomset. On peut naturellement se demander maintenant s'il est possible, partant d'un pomset d'avoir un Γ-Pomset qui l'encode. Ce qui nous pousse à énoncer la proposition suivante.

Proposition 4.2.2 *Pour tout pomset $t = (E, \preceq, \lambda)$ fini, il existe un Γ-Pomset γ tel que $\Psi_\Gamma(\gamma) = t$. L'application Ψ_Γ qui associe un Γ-Pomset à un pomset est de ce fait une fonction surjective.*

Preuve:
Il suffit de partir d'une linéarisation du pomset et de prendre comme ensemble de marques Γ les événements du pomset donc $\Gamma = E$. Ensuite pour chaque lettre de la linéarisation on prend comme marque β l'événement associé et comme marque α l'ensemble des événements prédécesseurs immédiats de l'événement associé. □

4.2.4 Autres notations

Pour un Γ-Pomset γ, nous notons par :

- Γ_γ l'ensemble des marques de Γ qui sont présentes dans γ,
- Γ_γ^α (resp. Γ_γ^β) l'ensemble des marques α (resp. β) qui sont présents dans γ.

Si L est un langage de Γ-Pomsets, nous notons par:

- Γ_L l'ensemble des marques de Γ qui sont utilisées par les Γ-Pomsets de L

Dans la section qui suit, nous allons montrer des cas d'encodage de pomset pour mettre en évidence l'aspect modélisation avec les Γ-Pomsets.

4.3 Modélisation avec des Γ-Pomsets

On peut modéliser des systèmes distribués grâce aux Γ-Pomsets. Chaque lettre $\langle \alpha_i, a_i, \beta_i \rangle$ d'un Γ-Pomset modélise un évènement étiqueté a_i. Les relations de causalité sont induites par la relation de couverture $\rightarrowtail_\Gamma$. Nous proposons dans cette section quelques exemples pour mettre en évidence les possibilités qu'offrent les Γ-Pomset en terme de modélisation.

4.3.1 Modélisation du système Producteur-Consommateur

Dans un système *Producteur-Consommateur*, une activité de consommation doit être précédée par au moins une activité de production, les productions sont ordonnées entre elles et les consommations sont ordonnées entre elles et en terme de causalité une activité de production et une activité de consommation peuvent être causalement dépendantes d'une même activité de production. Avec un alphabet $\Sigma = \{p, c\}$ p désignant une activité de production et c une activité de consommation, on peut modéliser ce système par un pomset comme on l'a vu sur la figure 2.5. Si nous nommons t ce pomset, son ensemble de linéarisation est $LE(t) = \{pppccc, ppcpcc, ppccpc, pcppcc, pcpcpc\}$.

Nous pouvons définir facilement un Γ-Pomset permettant la spécification de ce système composé de 3 productions et 3 consommations. Prenons un ensemble de marque $\Gamma = \{0, 1, 2, 3, 4, 5\}$. Le mot $\gamma = \langle\emptyset, p, 0\rangle\langle 0, c, 1\rangle\langle 0, p, 2\rangle\langle\{1, 2\},\ c, 3\rangle\langle 2, p, 4\rangle\langle\{3, 4\}, c, 5\rangle$ est un Γ-Pomset qui induit par dépliage le pomset t; c'est à dire $\Psi_\Gamma(\gamma) = t$.

Nous pouvons aussi spécifier le système avec deux marques seulement pour $\Gamma = \{0, 1\}$. Voici le Γ-Pomset γ' avec les ses relations $\rightarrowtail_\Gamma$ permettant de retrouver les relations de causalités.

$\langle 0, p, 0\rangle \quad \langle\{0, 1\}, c, 1\rangle \quad \langle 0, p, 0\rangle \quad \langle\{0, 1\}, c, 1\rangle \quad \langle 0, p, 0\rangle \quad \langle\{0, 1\}, c, 1\rangle$

Il est important de noter à ce stade que les Γ-Pomsets en tant que mots peuvent s'écrire en utilisant des opérations classiques des expressions régulières définies dans les théories des langages. De ce fait, vu qu'on a une certaine régularité dans notre exemple, on peut généraliser le système Producteur-Consommateur infini (avec un nombre indéterminé de productions et de consommations cf figure 4.2) en le spécifiant avec le Γ-Pomset $\gamma'' = (\langle 0, p, 0\rangle\langle\{0, 1\}, c, 1\rangle)^*$. Ceci illustre bien le fait qu'on peut modéliser des pomsets infinis en Γ-Pomsets .

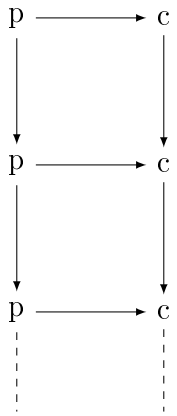


Figure 4.2: Pomset producteur-consommateur infini

4.3.2 Modélisation d'une exécution parallèle de tâches en nombre indéterminé

La figure 4.3 représente une exécution parallèle d'une action b en nombre indéterminé de fois suivie d'une action c. Ça peut modéliser un système du type *Fork and Join*.

Représenter une telle exécution en Γ-Pomset peut poser problème car l'ensemble des marques β de l'événement étiqueté c est potentiellement infini.

Pour parer à cela, nous introduisons des événements non observables (le vide ε) afin que le nombre de prédécesseurs immédiats des événements soit borné à deux. On se retrouve du coup à la représentation de la figure 4.4. La traduction en Γ-Pomset utilise alors trois marques et donne :
$\langle\emptyset, a, 0\rangle(\langle 0, b, 2\rangle\langle\{1, 2\}, \varepsilon, 1\rangle)^*\langle 1, c, \emptyset\rangle$

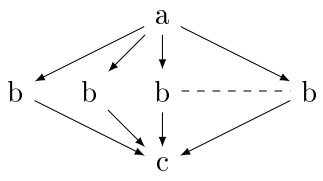


Figure 4.3: Représentation d'une exécution parallèle d'une action b

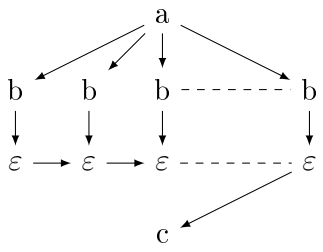


Figure 4.4: Représentation d'une exécution parallèle d'une action b avec des événements non observables

Nous remarquons donc à travers cet exemple que l'utilisation d'événements non observables peut être nécessaire pour la traduction de certains pomsets infinis. D'où la proposition suivante.

Proposition 4.3.1 *Il existe* $S \subseteq \mathbb{P}(\Sigma)$ *tel que quelque soit* $\gamma \in \mathbb{P}_\Gamma(\Sigma)$, *et quelque soit* $t \in \mathbb{P}(\Sigma)$ *si* $\Psi_\Gamma(\gamma) = t$ *alors* γ *contient des événements non observables* ε.

Preuve:
Il suffit de considérer l'exemple de la figure 4.3. Le représenter en une expression de Γ-Pomsets sans ε signifie avoir, pour l'événement étiqueté c, un ensemble de marque α non borné. Ce qui est absurde par rapport au fait que Γ soit fini.

□

L'utilisation d'événements non observables augmente donc le pouvoir d'expression des Γ-Pomsets .

4.3.3 Modélisation d'un produit synchronisé de système de transition

Nous avons parlé des automates dans la section 2.1. On peut faire des produits synchronisés d'automates (ou plus généralement de systèmes de transitions) pour décrire un système composé de plusieurs sous-systèmes. Pour cela on crée l'automate de chaque sous-système et on les fait coopérer par synchronisation pour obtenir une structure modélisant le comportement global du système. La synchronisation peut se faire de différentes manières: partage de variable d'état, synchronisation par envoie de messages etc.

Dans cette partie, nous prenons un exemple de produit synchronisé de deux systèmes de transitions pour illustrer comment on peut modéliser des causalités entre événements en passant par les Γ-Pomsets . Nous prenons comme modèle de synchronisation le modèle de Arnold-Nivat publié dans [AN82]. On montrera ensuite comment retrouver un pomset à partir du système via les Γ-Pomsets tout en restant sur un aspect purement de modélisation.

Nous considérons un modèle simplifié de gestionnaire d'impression avec une imprimante partagée entre deux utilisateurs A et B, en exclusion mutuelle. L'imprimante a deux états possibles: $s0$ (inactif) ou $s1$ (occupée). Les actions qui étiquettent les transitions sont: *imp* (imprimer) ou *dis* (rendre inactif). Ce qui nous donne les deux automates pour A et B de la figure 4.5.

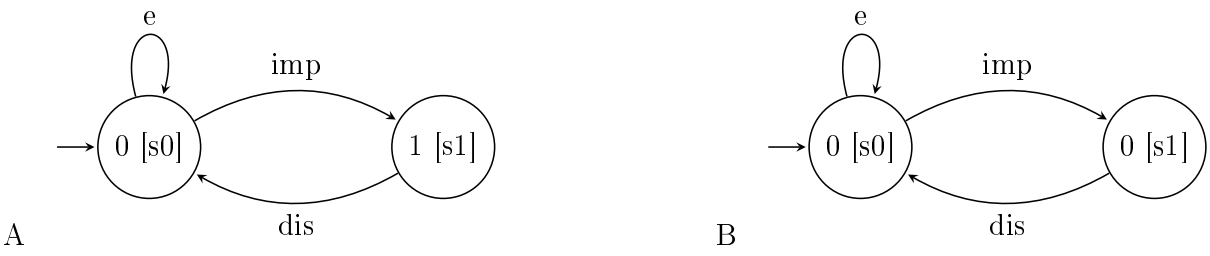


Figure 4.5: Automate gestionnaire d'imprimante pour utilisateur A et B

La présence de la boucle étiquetée e sur l'état $s0$ permet de modéliser les phénomènes asynchrones. Sa sémantique intuitive est: "je ne fais rien". Cette transition est utile quand on est dans le modèle de Arnold-Nivat pour définir les vecteurs de synchronisation qui sont des n-uplets d'actions représentant chacun une évolution globale autorisée du système.

Pour notre cas, l'ensemble des vecteurs de synchronisation est donc :
$V = \{(e, imp), (imp, e), (dis, \_), (\_, dis)\}$. Le tiret "_ " représente le fait que n'importe quelle action peut participer. Le produit de synchronisation nous donne le système de transition représenté dans la figure 4.6. Nous étiquetons les transitions par les numéros de processus qui y interviennent ($p0$ pour le processus de A et $p1$ pour celui de B).

Parmi les exécutions possible de l'automate prenons par exemple l'exécution $0 \xrightarrow{p0p1} 2 \xrightarrow{p1}$ ... (celui de gauche) qui peut se traduire en langage naturel par "B lance une impression A reste en attente jusqu'à ce que B libère l'imprimante et revienne à son état initial". Nous

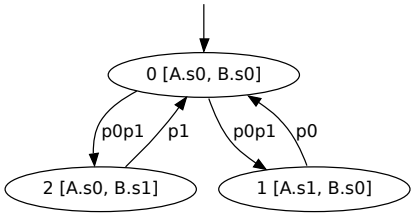


Figure 4.6: Produit synchronisé $A \times B$

pouvons représenter cette exécution sous forme de pomset où les événements désignent (sont étiquettés par) les états des processus pris indépendamment. Nous obtenons donc le pomset représenté par la figure 4.7:

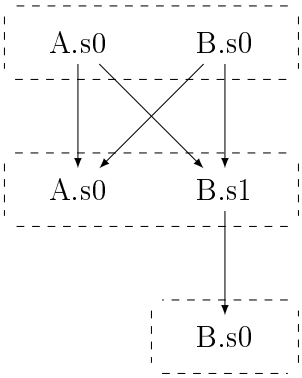


Figure 4.7: Pomset traduisant une exécution de $A \times B$

Cette exécution peut maintenant être représenté en Γ-Pomset en prenant $\Gamma = \{0, 1\}$. Ce qui donne $\gamma = (\emptyset, A.s0, 0)(\emptyset, B.s0, 1)(\{0, 1\}, A.s0, 0)(\{0, 1\}, B.s1, 1)(1, B.s0, 1)$. Remarquons que nous prenons systèmatiquement 0 comme marque β de A et 1 comme marque β de B.

Nous pouvons appliquez la même démarche pour l'autre cas d'exécution de l'automate produit synchronisé $A \times B$. Nous obtenons symétriquement, un résultat équivalent.

Nous pouvons représenter (informellement) l'automate de la figure en remplaçant les étiquettes de transition par des mots Γ-Pomset et nous obtenons la figure 4.8.

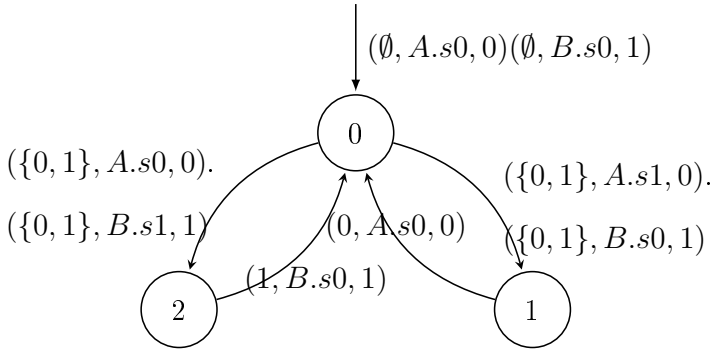


Figure 4.8: Autre représentation de $A \times B$

De manière plus simple, on peut représenter ce système par un Γ-automate en y ajoutant des états intermédiaires. Ce qui nous donne la représentation de la figure 4.9.

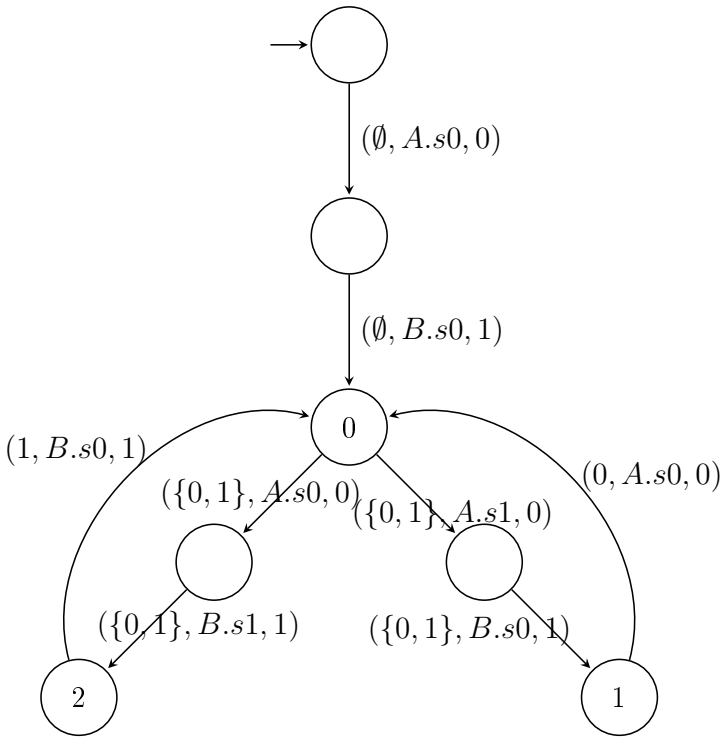


Figure 4.9: Γ-automate de $A \times B$

Chapitre 5

Relations avec les modèles classiques concurrents

Sommaire

Nous montrons dans ce chapitre les relations qui existent entre notre modèle Γ-Pomset et certains modèles classiques de la concurrence. Cette étude nous permettra d'avoir un bon aperçu sur le pouvoir d'expression de nos Γ-Pomsets.

5.1 Traces de Mazurkiewicz et Γ-Pomsets

Nous avons présenté les traces de Mazurkiewicz dans la section 3.1 en tant que modèle pouvant représenter la concurrence grâce à un alphabet de dépendance. La traduction d'une trace en Γ-Pomset est en fait triviale vu qu'une trace peut être représentée en pomset comme nous l'avons montré dans la section 3.1.4. Nous montrons ici comment à partir d'un alphabet de concurrence et un représentant u de la classe d'équivalence $[u]$ désignant une trace dans cet alphabet de concurrence, on peut associer un Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$.

Définition 5.1.1 ($\mathcal{T}_\mathcal{D}$) *Soit (Σ, D) un alphabet de concurrence. Nous définissons $\mathcal{T}_\mathcal{D}$: $\mathbb{T}(\Sigma, I) \longmapsto \mathbb{P}_\Gamma(\Sigma)$ comme une application qui à toute trace $[u]$, pour $u = u_0 \dots u_n$ associe le Γ-Pomset $\mathcal{T}_\mathcal{D}([u]) = \langle \alpha_0, u_0, \beta_0 \rangle \cdots \langle \alpha_n, u_n, \beta_n \rangle$ avec pour tout $i \in [0, n[$:*

- $\alpha_i = D_{u_i}$
- $\beta_i = \{u_i\}$

Pour rappel, $D_{u_i} = \{u_j \in \Sigma \mid (u_i, u_j) \in D\}$ (voir section 3.1.2).

Cette définition stipule que pour chaque élément (α_i, a_i, β_i) du Γ-Pomset $\mathcal{T}_{\mathcal{D}}([u])$, on a a_i qui correspond à u_i et β_i qui est restreint aussi au symbole u_i. α_i quant à lui est composé de l'ensemble des éléments de u étant en relation de dépendance avec u_i. L'ensemble de marque Γ est défini comme étant égale à Σ.

Exemple 5.1.1 *Reprenant l'alphabet de concurrence et la trace de l'exemple 3.1.1,* $D = \{(a,a),(b,b),(c,c),(a,c),(c,a),(b,c),(c,b)\}$ *et* $u = abcbac$, *on a* $\mathcal{T}_{\mathcal{D}}([u]) = \langle\{c,a\},a,a\rangle\langle\{c,b\},b,b\rangle\langle\{a,b,c\},c,c\rangle\langle\{c,b\},b,a\rangle\langle\{c,a\},a,a\rangle\langle\{a,b,c\},c,c\rangle$.

Avec cette définition nous montrons avec la proposition suivante qu'une trace $[u]$ et son Γ-Pomset associé $\mathcal{T}_{\mathcal{D}}([u])$ exprime la même chose en terme de concurrence. Autrement dit, le pomset associé à la trace $[u]$ et le pomset associé au Γ-Pomset $\mathcal{T}_{\mathcal{D}}([u])$ sont les mêmes.

Proposition 5.1.1 *Soit* (Σ, D) *un alphabet de dépendance et* u *un mot de* Σ^*. *On a* $\Psi_\Gamma(\mathcal{T}_{\mathcal{D}}([u])) = \Psi_D([u])$.

Preuve:

Prenons n'importe quel mot $u = u_0...u_n$ de Σ^*. Posons $\Psi_\Gamma(\mathcal{T}_{\mathcal{D}}([u])) = (E_\gamma, \preceq_\gamma, \lambda_\gamma)$ et $\Psi_D([u]) = (E_u, \preceq_u, \lambda_u)$. Par définition $E_\gamma = E_u$ et $\lambda_\gamma = \lambda_u$. (voir définitions 3.1.5 et 4.2.3).

$\preceq_\gamma$ est par définition la fermeture transitive de la relation $\rightarrowtail_\Gamma$, et $\preceq_u$ est la fermeture transitive de la relation $\rightarrowtail_D$ qui désigne pour rappel l'ensemble $\{(i,j) | i \leq j \wedge (u_i, u_j) \in D\}$ pour tout i et $j \in [0, |n|[$. Nous avons donc $\rightarrowtail_\Gamma \subseteq \preceq_\gamma$ et $\rightarrowtail_D \subseteq \preceq_u$. Pour prouver que $\preceq_\gamma = \preceq_u$ il nous suffit de montrer que: (1) $\preceq_u$ est aussi la fermeture transitive de $\rightarrowtail_\Gamma$ et que (2) $\preceq_\gamma$ est aussi la fermeture transitive de $\rightarrowtail_D$.

(1) Soit i, j deux éléments de $[0, |n|[$ tels que $i \rightarrowtail_\Gamma j$. On a par définition $i \leq j$. Supposons que $(u_i, u_j) \notin D$. Donc d'après la définition 5.1.1 on a $u_i \notin \alpha_j$ et $\beta_i = \{u_i\}$ dans $\mathcal{T}_{\mathcal{D}}([u])$; ce qui est contradictoire avec le fait que $i \rightarrowtail_\Gamma j$ puisqu'il n'existe pas de marque m tel que $m \in \beta_i \wedge m \in \alpha_j \wedge$. Par conséquent $(u_i, u_j) \in D$ et donc $i \rightarrowtail_D j$. Ceci démontre que $\rightarrowtail_\Gamma \subseteq \rightarrowtail_D$. $\preceq_u$ étant la plus petite relation transitive contenant $\rightarrowtail_D$ qui lui même contient $\preceq_u$ est donc la fermeture transitive de $\rightarrowtail_\Gamma$

(2) Soit i, j deux éléments de $[0, |n|[$ tels que $i \rightarrowtail_D j$. Par définition $i \leq j$ et $u_i \in D_{u_j}$. On peut raisonner par récurrence en considérant les cas suivants:

- Si $j = i + 1$ alors dans $\mathcal{T}_{\mathcal{D}}([u])$ (et d'après sa définition) on a la séquence: $\ldots\langle D_{u_i}, u_i, u_i\rangle\langle D_{u_j}, u_j, u_j\rangle\ldots$. Ce qui fait que $i \rightarrowtail_\Gamma j$ car $u_i \in D_{u_j}$ et dans ce cas $i \preceq_\gamma j$.

- S'il existe un $k \in [0, |n|[$ tel que $i + 1 = k = j - 1$ alors si $u_k \in D_{u_j}$ on a $i \rightarrowtail_\Gamma k \rightarrowtail_\Gamma j$ sinon on a directement $i \rightarrowtail_\Gamma j$ et donc $i \preceq_\gamma j$.

- Supposons que $i \preceq_\gamma i + 1 \preceq_\gamma i + 2 \ldots \preceq_\gamma k$ avec $k = j - 1$. Si $u_k \in D_{u_j}$ on a $k \rightarrowtail_\Gamma j$ sinon on il existe $l \in [i, k[$ tel que $l \rightarrowtail_\Gamma j$ du moment que $i \rightarrowtail_D j$ et donc $i \preceq_\gamma j$.

Par conséquent $\preceq_\gamma$ est aussi la fermeture transitive de $\rightarrowtail_D$ et au final avec (1) et (2) ainsi prouvés, nous avons l'égalité $\preceq_\gamma = \preceq_u$. □

Remarque: Reprenant l'exemple 5.1.1 précédent, on peut voir que nous pouvons naturellement représenter une exécution infinie de ce système par $((c,a,a)(c,b,a)(\{a,b\},c,c))^*$.

Théorème 5.1.1 *Soit $[L]_I$ un langage de trace régulier, nous pouvons lui associer un langage Π de Γ-Pomsets tel que $\Pi = \cup_{u\in L}\Psi_D(u)$ avec Π régulier.*

Preuve:
Ce résultat est une application directe de la proposition 5.1.1. En effet, l'opération Ψ_D peut être vue comme une transduction : chaque lettre u de l'alphabet de dépendance Σ est substituée par son équivalent $\langle D_u, u, u\rangle$.

□

Il existe une formulation simplifiée de la relation de dépendance dans le cas où l'alphabet est composé des parties non vides d'un ensemble donné A. L'ensemble A représente généralement un ensemble fini de processus et le sous-ensemble A attaché à une transition modélise la liste des processus réalisant conjointement la transition. La relation de dépendance se formule ainsi : deux sous-ensembles non vides $A_1, A_2 \subseteq A$ sont dépendants si $A_1 \cap A_2 \neq \emptyset$.

Dans ce modèle, la traduction d'une trace s'exprime simplement par une transduction transformant la lettre A_1 en $\langle A_1, A_1, A_1\rangle$. La modélisation du produit synchronisé présentée dans la section 4.3.3 est un exemple de traduction de cette nature. Le théorème suivant formalise cette traduction.

Proposition 5.1.2 *Soit A un ensemble fini. Posons $\Sigma = 2^A \setminus \emptyset$ l'alphabet muni de la relation de dépendance $A_1 \cap A_2 \neq \emptyset$. Posons $\Psi_{D'}$ l'homomorphisme sur les traces sur Σ vers les Γ-pomsets avec comme ensemble de marques A :*

$$\Psi_{D'}(A_1) = \langle A_1, A_1, A_1\rangle$$

Alors pour toute trace u sur Σ, $u = \Psi_{D'}(u)$ en tant que pomset. D'autre part si U est un ensemble régulier de trace, $\Psi_{D'}(U)$ est régulier et est égal à U en tant qu'ensemble de pomsets.

Preuve:
Il nous suffit de montrer que pour toute trace u, $\Psi_{D'}(u) = \Psi_D(u)$. La relation de causalité dans un Γ-pomset s'exprime en testant le vide de l'intersection du Pré et du Post de lettre du Γ-pomset. Pour les deux traductions et deux lettres A_1, A_2 de u, le test donne le même résultat :

- Pour la traduction $\Psi_{D'}$, le test est $A_1 \cap A_2 \neq \emptyset$
- Pour la traduction Ψ_D, le test est $D(A_1) \cap \{A_2\} \neq \emptyset$, équivalent à $A_2 \in D(A_1)$. Comme $D(A_1)$ est défini comme l'ensemble des parties ayant une intersection non vide avec A_1, nous avons équivalence des deux tests.

□

5.2 Pomsets série-parallèles et Γ-Pomsets

Nous avions présenté les pomsets séries-parallèles (sp-pomset) dans la section 3.2. Dans cette section nous montrons comment traduire des sp-pomsets en Γ-Pomsets. Afin de couvrir la plus large classe possible de sp-pomset, notre démarche sera de traduire directement les différentes opérations permettant de contruire des langages rationnels de sp-pomsets.

Il sera beaucoup question ici d'utiliser des événements non observables (ε-événements) qui nous servirons de "liaison" dans les opérations que nous allons traduire. Nous choisissons notamment d'encadrer les Γ-Pomsets par des ε-événements minimaux et maximaux.

L'idée de notre approche est, considérant un Γ-Pomset, d'introduire un élément en première position de sorte qu'il soit le seul minimum du Γ-Pomset et d'introduire aussi un autre élément en dernière position du Γ-Pomset en veillant à ce qu'il soit le seul maximum du Γ-Pomset considéré. Nous allons définir formellement la notion de Γ-Pomset bien construit.

Définition 5.2.1 (Γ-Pomset bien construit) *Soit γ un Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$. Soit γ' un autre Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$ tel que $\gamma' = \langle\emptyset, \varepsilon, \Gamma\rangle\gamma\langle\Gamma, \varepsilon, \emptyset\rangle$. On dit que γ est bien construit si on a:*
Propriété BC: $\forall i \in [0, |\gamma'|[,\ 0 \preceq_{\gamma'} i \preceq_{\gamma'} |\gamma'| - 1$.
au niveau du pomset induit $\Psi_\Gamma(\gamma')$.

Cette définition signifie qu'un Γ-Pomset γ est bien construit si, en mettant l'élément $\langle\emptyset, \varepsilon, \Gamma\rangle$ en première position et $\langle\Gamma, \varepsilon, \emptyset\rangle$ on n'a pas d'élément minimum à part le premier ni d'élément maximum à part le dernier; en d'autres termes, en écrivant $\gamma' = \langle\emptyset, \varepsilon, \Gamma\rangle\gamma\langle\Gamma, \varepsilon, \emptyset\rangle$ on a:

- $\forall i \in [0, |\gamma'| - 1[, \exists j \in [0, |\gamma'|[\mid i \rightarrowtail_\Gamma j$
- $\forall i \in [1, |\gamma'|[, \exists j \in [0, |\gamma'|[\mid j \rightarrowtail_\Gamma i$

Remarque: A partir de cette définition, il est évident que pour un Γ-Pomset γ bien construit on a: $\forall i \in [0, |\gamma|[,\ \alpha_i \neq \emptyset$ et $\beta_i \neq \emptyset$.

Exemple 5.2.1 *Soient $\Gamma = \{0, 1\}$ et $\Sigma = \{a, b\}$. Le Γ-Pomset $\langle 0, a, 1\rangle\langle 1, b, 0\rangle$ est bien construit. Par contre $\langle 0, a, 1\rangle\langle 0, b, 1\rangle$ n'est pas bien construit car l'élément $\langle 0, a, 1\rangle$ est un maximum dans le Γ-Pomset $\langle\emptyset, \varepsilon, \{0, 1\}\rangle\langle 0, a, 1\rangle\langle 0, b, 1\rangle\langle\{1, 0\}, \varepsilon, \emptyset\rangle$.*

Proposition 5.2.1 *Soit $\gamma \in \mathbb{P}_\Gamma(\Sigma)$ un Γ-Pomset, si $|\gamma| = 1$ alors il est bien construit.*

Preuve:
Soit $\gamma = \langle\alpha, a, \beta\rangle$ un Γ-Pomset tel que $\alpha, \beta \subseteq \Gamma$ et $a \in \Sigma \cup \{\varepsilon\}$. En posant $\gamma' = \langle\emptyset, \varepsilon, \Gamma\rangle\langle\alpha, a, \beta\rangle\langle\Gamma, \varepsilon, \emptyset\rangle$, il apparait immédiatement que: $0 \rightarrowtail_\Gamma 1 \rightarrowtail_\Gamma 2$ vu que $\alpha \subseteq \Gamma$ et $\beta \subseteq \Gamma$. La propriété BC est donc respectée, par conséquent γ est bien construit.

Définition 5.2.2 (Langage bien construit) *Un langage $L \subseteq \mathbb{P}_\Gamma(\Sigma)$ est bien construit s'il n'est formé que de Γ-Pomsets bien construits.*

Nous allons maintenant donner notre traduction des langages de sp-pomset en Γ-Pomset. Le principe sera de préserver la propriété de "bien construit".

Définition 5.2.3 ($\mathcal{T}_\mathcal{S}$) *Soit $L \subseteq \mathbb{SP}(\Sigma)$ un langage rationnel de série-parallèle et Γ un ensemble de marques, nous définissons $\mathcal{T}_\mathcal{S}(L)$ comme une transformation de L en un langage de Γ-Pomsets inductivement comme suit:*

$$\mathcal{T}_\mathcal{S}(L) = \begin{cases} \varepsilon & \text{si } L = \{\varepsilon\} \text{ (où } \varepsilon \text{ représente le pomset sans événement)} \\ \langle m, a, m\rangle & \text{si } L = \{a\} \text{ (une lettre de } \Sigma) \text{ avec } m \text{ une marque de } \Gamma \\ \mathcal{T}_\mathcal{S}(L_1)\langle\Gamma_1, \varepsilon, \Gamma_2\rangle\mathcal{T}_\mathcal{S}(L_2) & \text{si } L = L_1.L_2 \text{ avec } L_1, L_2 \subseteq \mathbb{SP}(\Sigma) \\ \mathcal{T}_\mathcal{S}(L_1)\langle\Gamma_1, \varepsilon, m_1\rangle\langle m_2, \varepsilon, \Gamma_2\rangle\mathcal{T}_\mathcal{S}(L_2) & \text{si } L = L_1 \parallel L_2 \text{ avec } L_1, L_2 \subseteq \mathbb{SP}(\Sigma), \\ & m_1, m_2 \text{ des marques de } \Gamma \text{ tels que:} \\ & m_1 \notin \Gamma_2,\ m_2 \notin \Gamma_1 \text{ et } m_1 \neq m_2. \\ (\mathcal{T}_\mathcal{S}(L_1)\langle\Gamma_1, \varepsilon, \Gamma_1\rangle)^* & \text{si } L = L_1^* \text{ avec } L_1 \subseteq \mathbb{SP}(\Sigma) \\ (\langle m_1, \varepsilon, \Gamma_1 \cup \{m_1\}\rangle\mathcal{T}_\mathcal{S}(L1)\langle\Gamma_1 \cup \{m_2\}, \varepsilon, m_2\rangle)^* & \text{si } L = L_1^{\Vdash} \text{ avec } L_1 \subseteq \mathbb{SP}(\Sigma) \\ & m_1, m_2 \text{ des marques de } \Gamma \text{ tels que:} \\ & m_1 \notin \Gamma_1,\ m_2 \notin \Gamma_1 \text{ et } m_1 \neq m_2. \\ \mathcal{T}_\mathcal{S}(L_1) \cup \mathcal{T}_\mathcal{S}(L2) & \text{si } L = L_1 + L_2 \text{ avec } L_1, L_2 \subseteq \mathbb{SP}(\Sigma). \end{cases}$$

Γ_i (avec $i \in \{1, 2\}$) désigne ici l'ensemble des marques (non vides) de Γ utilisées par $\mathcal{T}_\mathcal{S}(Li)$

Exemple 5.2.2 *Soit $L = \{(aa) \parallel b\}$. Nous prenons $\Gamma = \{0, 1, 2, 3\}$. Si nous posons $L_1 = (aa)$ et $L_2 = \{b\}$ on a $L = L_1 \parallel L_2$. Si nous posons de plus $L_3 = \{a\}$ on a $L_1 = L_3.L_3$. Par définition nous pouvons écrire $\mathcal{T}_\mathcal{S}(L_1) = \langle 0, a, 0\rangle\langle 0, \varepsilon, 0\rangle\langle 0, a, 0\rangle$ et $\mathcal{T}_\mathcal{S}(L_2) = \langle 1, b, 1\rangle$. Et pour L nous pouvons donc écrire $\mathcal{T}_\mathcal{S}(L) = \langle 0, a, 0\rangle\langle 0, \varepsilon, 0\rangle\langle 0, a, 0\rangle\langle 0, \varepsilon, 2\rangle\langle 3, \varepsilon, 1\rangle\langle 1, b, 1\rangle$.*

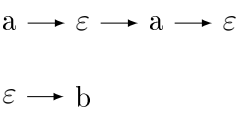


Figure 5.1: Représentation du pomset de $\mathcal{T}_\mathcal{S}(\{(aa) \parallel b\})$

Remarque:

1. Par construction, $\mathcal{T}_\mathcal{S}(L)$ est toujours non vide même pour le cas particulier où $L = \varepsilon$, on a $\mathcal{T}_\mathcal{S}(L) = \langle m, \varepsilon, m\rangle$ avec m une marque de Γ non vide.
2. Il est possible de supprimer certains $\varepsilon-$*transitions* par la suite. Pour l'exemple précédent, $\mathcal{T}_\mathcal{S}(L)$ peut se réduire en $\langle 0, a, 0\rangle\langle 0, a, 0\rangle\langle 0, \varepsilon, 2\rangle\langle 3, \varepsilon, 1\rangle\langle 1, b, 1\rangle$.

Démarche de minimisation des marques:
La construction de $\mathcal{T}_\mathcal{S}$ peut quelques fois faire introduire de nouvelles marques. Ce qui peut s'avérer inefficace en terme d'optimisation. Il est possible pour le cas $L = L_1 \parallel L_2$ de limiter le nombre de nouvelles marques à employer pour m_1 et m_2. Pour cela, deux cas sont à considérer:

- Si $\Gamma_1 \neq \Gamma_2$ on peut prendre un $m_1 \in \Gamma_1$ et un $m_2 \in \Gamma_2$; de ce fait on a pas de nouvelles marques à ajouter sauf si Γ_1 est inclus dans Γ_2 ou vice et versa auxquels cas on peut ajouter au plus une marque.

- Si $\Gamma_1 = \Gamma_2$ on peut procéder à un changement de marque. La méthode est de prendre une marque de Γ_2 ensuite la renommer et la prendre en marque m_2. L'ancienne marque (de son premier nom) est ainsi prise comme marque m_1.

$\mathcal{T}_{\mathcal{S}}(L)$ de l'exemple précédent aurait pu s'écrire $\langle 0,a,0\rangle\langle 0,a,0\rangle\langle 0,\varepsilon,0\rangle\langle 1,\varepsilon,1\rangle\langle 1,b,1\rangle$ limitant ainsi Γ à $\{0,1\}$.

Exemple 5.2.3 *Soit* $L = (((aa) \parallel b)c)^*$. *Soit* $\Gamma = \{0,1\}$. *Nous avons vu avec l'exemple précédent que* $\mathcal{T}_{\mathcal{S}}(\{(aa) \parallel b\}) = \langle 0,a,0\rangle\langle 0,a,0\rangle\langle 0,\varepsilon,0\rangle\langle 1,\varepsilon,1\rangle\langle 1,b,1\rangle$. *Si nous posons* $L_1 = \{(aa) \parallel b\}$ *et* $L_2 = \{c\}$, *alors* $L = (L_1.L_2)^*$. *Par définition on peut écrire* $\mathcal{T}_{\mathcal{S}}(L_1.L_2) = \langle 0,a,0\rangle\langle 0,a,0\rangle\langle 0,\varepsilon,0\rangle\langle 1,\varepsilon,1\rangle\langle 1,b,1\rangle\langle\{0,1\},\varepsilon,0\rangle\langle 0,c,0\rangle$ *et finalement par la définition de* $\mathcal{T}_{\mathcal{S}}$ *pour* L^* *on a*
$\mathcal{T}_{\mathcal{S}}(L) = (\langle 0,a,0\rangle\langle 0,a,0\rangle\langle 0,\varepsilon,0\rangle\langle 1,\varepsilon,1\rangle\langle 1,b,1\rangle\langle\{0,1\},\varepsilon,0\rangle\langle 0,c,0\rangle\langle\{0,1\},\varepsilon,\{0,1\}\rangle)^*$.

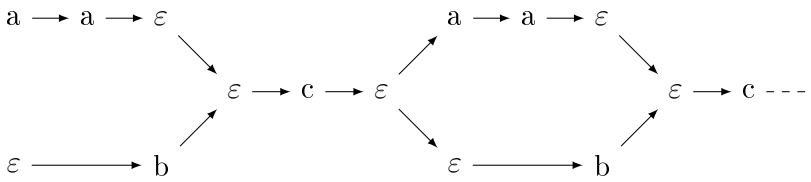


Figure 5.2: Représentation du pomset de $\mathcal{T}_{\mathcal{S}}((((aa) \parallel b)c)^*)$

Exemple 5.2.4 *Soit* $L = (a \parallel b)^{\Vdash}$. *Soit* $\Gamma = \{0,1,2,3\}$. *Posant* $L_1 = a \parallel b$ *nous pouvons écrire* $\mathcal{T}_{\mathcal{S}}(L_1) = \langle 0,a,0\rangle\langle 0,\varepsilon,0\rangle\langle 1,\varepsilon,1\rangle\langle 1,b,1\rangle$. *Par la définition de* $\mathcal{T}_{\mathcal{S}}$ *pour* $L^{\Vdash}$ *on a* $\mathcal{T}_{\mathcal{S}}(L) = (\langle 2,\varepsilon,\{0,1,2\}\rangle\langle 0,a,0\rangle\langle 0,\varepsilon,0\rangle\langle 1,\varepsilon,1\rangle\langle 1,b,1\rangle\langle\{0,1,3\},\varepsilon,3\rangle)^*$.

Maintenant il s'avère nécessaire de s'assurer que nos transformations fonctionnent bien, c'est à dire que les pomsets que codent les Γ-Pomsets issus des transformations $\mathcal{T}_{\mathcal{S}}$ sont identiques aux ensembles de pomsets découlant des langages rationnels de sp-pomset.

Théorème 5.2.1 *Soit* $L \subseteq \mathbb{SP}(\Sigma)$ *un langage rationnel de séries-parallèles, alors* $\mathcal{T}_{\mathcal{S}}(L)$ *est bien construit et* $\Psi_\Gamma(\mathcal{T}_{\mathcal{S}}(L)) = L$ *en tant qu'ensemble de pomsets.*

Preuve:
Notre transformation $\mathcal{T}_{\mathcal{S}}$ donne par définition inductive une traduction pour chaque opération de construction de langage rationnel. Montrons par induction que chaque transformation donne un langage de Γ-Pomsets bien construit et que $\Psi_\Gamma(\mathcal{T}_{\mathcal{S}}(L)) = L$.

Soit $L \subseteq \mathbb{SP}(\Sigma)$ un langage rationnel de série-parallèle.

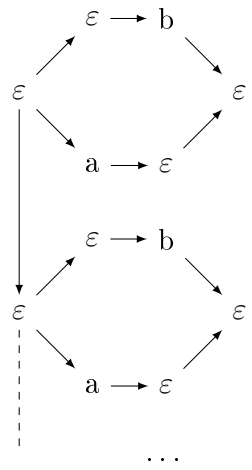


Figure 5.3: Représentation du pomset de $\mathcal{T}_{\mathcal{S}}((a \parallel b)^{\Vdash})$

1. Cas $L = \{\varepsilon\}$
   $\mathcal{T}_{\mathcal{S}}(L) = \varepsilon$. La proposition est immédiatement de vérifier.

2. Cas $L = \{a\}$ pour un $a \in \Sigma$.
   Par définition $\mathcal{T}_{\mathcal{S}}(L) = \langle m, a, m\rangle$ pour tout choix de $m \in \Gamma$. $\mathcal{T}_{\mathcal{S}}(L) = \langle m, a, m\rangle$ est immédiatement bien construit et préserve la propriété d'égalité des langages de pomset.

3. Cas $L = L_1.L_2$ avec $L_1, L_2 \subseteq \mathbb{SP}(\Sigma)$.
   Par définition $\mathcal{T}_{\mathcal{S}}(L) = \mathcal{T}_{\mathcal{S}}(L_1)\langle\Gamma_1, \varepsilon, \Gamma_2\rangle\mathcal{T}_{\mathcal{S}}(L_2)$. Soit γ un Γ-Pomset appartenant à $\mathcal{T}_{\mathcal{S}}(L)$. Montrons que γ est bien construit et appartient à L. γ s'écrit sous la forme $\gamma_1\langle\Gamma_1, \varepsilon, \Gamma_2\rangle\gamma_2$ avec γ_1 et γ_2 des Γ-Pomsets de $\mathcal{T}_{\mathcal{S}}(L_1)$ et $\mathcal{T}_{\mathcal{S}}(L_2)$ respectivement. Par récurrence, γ_1 est dans $L1$ et γ_2 est dans L_2. La structure de γ fait que les relations de dépendance dans γ_1 et dans γ_2 sont préservées. Comme γ_1 et γ_2 sont bien construits, tout événement de γ_1 est plus petit que l'événement $\langle\Gamma_1, \varepsilon, \Gamma_2\rangle$ et que de même, tout événement γ_2 est plus grand que l'événement $\langle\Gamma_1, \varepsilon, \Gamma_2\rangle$. Ceci conclut le fait que γ est la concaténation des pomsets γ_1 et γ_2 en tant que série-parallèle. Le Γ-pomset γ est de toute évidence bien construit. Posons $\gamma' = \langle\emptyset, \varepsilon, \Gamma_1 \cup \Gamma_2\rangle\gamma_1\langle\Gamma_1, \varepsilon, \Gamma_2\rangle\gamma_2\langle\Gamma_1 \cup \Gamma_2, \varepsilon, \emptyset\rangle$. Tout événement issu de γ_1 dans γ' n'est ni un plus petit événement, ni un plus grand événement car γ_1 est bien construit. Il en est de même pour les événements de γ_2 et l'événement non observable $\langle\Gamma_1, \varepsilon, \Gamma_2\rangle$. Ainsi, γ est bien construit.

   Soit π un pomset de L. Montrons que π est un pomset de $\mathcal{T}_{\mathcal{S}}(L)$. Par récurrence, il existe $\gamma_1 \in \mathcal{T}_{\mathcal{S}}(L_1)$ et $\gamma_2 \in \mathcal{T}_{\mathcal{S}}(L_2)$ tel que la concaténation (en tant que série parallèle) de γ_1 et de γ_2 donne π. Il nous suffit de vérifier que cette concaténation est égale à $\gamma_1\langle\Gamma_1, \varepsilon, \Gamma_2\rangle\gamma_2$. Nous constatons que les relations de causalité des évènements de γ_1 et de γ_2 sont préservées, et que l'évènement non observable $\langle\Gamma_1, \varepsilon, \Gamma_2\rangle$ induit une causalité entre les événements de γ_1 et γ_2.

4. Cas $L = L_1 \parallel L_2$ avec $L_1, L_2 \subseteq \mathbb{SP}(\Sigma)$.

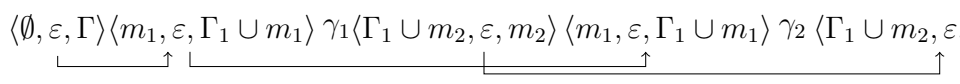
$\mathcal{T}_{\mathcal{S}}(L) = \mathcal{T}_{\mathcal{S}}(L_1)\langle\Gamma_1, \varepsilon, m_1\rangle\langle m_2, \varepsilon, \Gamma_2\rangle\mathcal{T}_{\mathcal{S}}(L2)$ par définition avec $m_1 \notin \Gamma_2$, $m_2 \notin \Gamma_1$ et $m_1 \neq m_2$. Soit γ un Γ-Pomset de $\mathcal{T}_{\mathcal{S}}(L)$. Il peut s'écrire sous la forme $\gamma_1\langle\Gamma_1, \varepsilon, m_1\rangle\langle m_2, \varepsilon, \Gamma_2\rangle\gamma_2$ avec $\gamma_1 \in \mathcal{T}_{\mathcal{S}}(L_1)$ et $\gamma_2 \in \mathcal{T}_{\mathcal{S}}(L_1)$. Montrons que γ est bien construit et est égal à $\gamma_1 \parallel \gamma_2$ en tant que série-parallèle. Les relations de causalité dans γ_1 et γ_2 sont préservées. L'introduction de l'évènement non observable $\langle m_2, \varepsilon, \Gamma_2\rangle$ avec $m_2 \notin \Gamma_1$ coupe toute possibilité de causalité entre les évènements de γ_1 et γ_2. Ceci montre que γ est bien égal $\gamma_1 \parallel \gamma_2$ en tant que série parallèle. D'autre part γ est bien construit. Posons $\gamma' = \langle\emptyset, \varepsilon, \Gamma\rangle\gamma_1\langle\Gamma_1, \varepsilon, m_1\rangle\langle m_2, \varepsilon, \Gamma_2\rangle\gamma_2\langle\Gamma, \varepsilon, \emptyset\rangle$ avec $\Gamma = \Gamma_1 \cup \Gamma_2 \cup \{m_1, m_2\}$. Notons que tout évènement de γ_1 est plus petit que l'évènement $\langle\Gamma_1, \varepsilon, m_1\rangle$ (car γ_1 est bien construit) qui lui-même est plus petit que $\langle\Gamma, \varepsilon, \emptyset\rangle$ (car $m_1 \notin \Gamma_2$). De la même façon, tout évènement de γ_2 est plus grand que $\langle\emptyset, \varepsilon, \Gamma\rangle$ grâce à l'introduction de l'évènement $\langle m_2, \varepsilon, \Gamma_2\rangle$. Nous en déduisons que γ est bien construit.

Réciproquement, soit π un pomset de L. Par récurrence, il existe γ_1 et γ_2 dans réciproquement $\mathcal{T}_{\mathcal{S}}(L_1)$ et $\mathcal{T}_{\mathcal{S}}(L_2)$ avec $\pi = \gamma_1 \parallel \gamma_2$. Posons $\gamma = \gamma_1\langle\Gamma_1, \varepsilon, m_1\rangle\langle m_2, \varepsilon, \Gamma_2\rangle\gamma_2$ avec $\gamma_1 \in \mathcal{T}_{\mathcal{S}}(L_1)$ et $\gamma_2 \in \mathcal{T}_{\mathcal{S}}(L_1)$. De toute évidence $\gamma = \pi$ en tant que pomset. En effet, les relations de causalité dans γ_1 et γ_2 sont préservées et l'évènement $\langle m_2, \varepsilon, \Gamma_2\rangle$ coupe toute possibilité de causalité entre les évènements de γ_1 et γ_2.

5. Cas $L = L_1^*$ avec $L_1 \subseteq \mathbb{SP}(\Sigma)$.
Par définition $\mathcal{T}_{\mathcal{S}}(L) = (\mathcal{T}_{\mathcal{S}}(L_1)\langle\Gamma_1, \varepsilon, \Gamma_1\rangle)^*$. En considérant $L = \cup_n L_1^n$, nous remarquons que $\mathcal{T}_{\mathcal{S}}(L) = (\cup_n \mathcal{T}_{\mathcal{S}}(L_1)^n).\langle\Gamma_1, \varepsilon, \Gamma_1\rangle$. En appliquant la proposition sur chacun des L_1^n, nous déduisons que les $\mathcal{T}_{\mathcal{S}}(L_1)^n$ sont bien construits et que $L^n = \mathcal{T}_{\mathcal{S}}(L_1)^n$ en tant qu'ensemble de pomsets. L'introduction de l'évènement $\langle\Gamma_1, \varepsilon, \Gamma_1\rangle$ ne change pas les ensembles en tant que pomsets et leurs natures bien construites. Ceci prouve que la proposition est vérifiée pour $L = L_1^*$.

6. Cas $L = L_1^{\Vdash}$ avec $L_1 \subseteq \mathbb{SP}(\Sigma)$.
$\mathcal{T}_{\mathcal{S}}(L) = (\langle m_1, \varepsilon, \Gamma_1 \cup m_1\rangle\mathcal{T}_{\mathcal{S}}(L1)\langle\Gamma_1 \cup m_2, \varepsilon, m_2\rangle)^*$ par définition avec $m_1 \notin \Gamma_1$, $m_2 \notin \Gamma_1$ et $m_1 \neq m_2$.

Soit $\gamma \in \mathcal{T}_{\mathcal{S}}(L)$. Il s'écrit donc $\langle m_1, \varepsilon, \Gamma_1 \cup m_1\rangle\gamma_1\langle\Gamma_1 \cup m_2, \varepsilon, m_2\rangle\langle m_1, \varepsilon, \Gamma_1 \cup m_1\rangle\gamma_2\langle\Gamma_1 \cup m_2, \varepsilon, m_2\rangle\gamma_3 \ldots$ avec les $\gamma_i \in \mathcal{T}_{\mathcal{S}}(L_1)$. Montrons que γ est bien construit et que $\gamma = \gamma_1 \parallel \gamma_2 \parallel \cdots$ en tant que pomset. Notons que les relations de causalité dans les γ_i sont préservés et que les événements $\langle m_1, \varepsilon, \Gamma_1 \cup m_1\rangle$ coupent toutes possibilités de relation de causalité entre les γ_i. Ainsi $\gamma = \gamma_1 \parallel \gamma_2 \parallel \cdots$. D'autre part, γ est bien construit. Posons $\gamma' = \langle\emptyset, \epsilon, \gamma \cup \{m_1, m_2\}\rangle.\gamma.\langle\gamma \cup \{m_1, m_2\}, \epsilon, \emptyset\rangle$. Les événements de chaque γ_i sont plus grands que l'évènement $\langle m_1, \varepsilon, \Gamma_1 \cup m_1\rangle$ le précédent et sont plus petits que l'événement $\langle\Gamma_1 \cup m_2, \varepsilon, m_2\rangle$ le succédant. Ceci tient au fait que $m_1, m_2 \notin \Gamma$. D'autre part, les $\langle m_1, \varepsilon, \Gamma_1 \cup m_1\rangle$ sont totalement ordonnées ainsi que les $\langle\Gamma_1 \cup m_2, \varepsilon, m_2\rangle$. Cette propriété tient compte du fait que $m_1 \neq m_2$ et que $m_1, m_2 \notin \Gamma$. Ainsi, γ est bien construit. Le diagramme suivant donne une vision claire des relation de causalité entre les évènements de γ'.



$\cdots \quad \langle \Gamma_1 \cup m_2, \varepsilon, m_2 \rangle \; \langle \Gamma, \varepsilon, \emptyset \rangle$
$\ldots$

Soit π un pomset de L. π est de la forme $\pi = \gamma_1 \parallel \gamma_2 \parallel \cdots$ avec les $\gamma_i \in \mathcal{T}_{\mathcal{S}}(L_1)$. Posons $\gamma = \langle m_1, \varepsilon, \Gamma_1 \cup m_1 \rangle \gamma_1 \langle \Gamma_1 \cup m_2, \varepsilon, m_2 \rangle \langle m_1, \varepsilon, \Gamma_1 \cup m_1 \rangle \gamma_2 \langle \Gamma_1 \cup m_2, \varepsilon, m_2 \rangle \gamma_3 \ldots$. Comme les relations de causalité dans les γ_i sont préservées et les possibilités de relation de causalité sont coupées par les $\langle m_1, \varepsilon, \Gamma_1 \cup m_1 \rangle$, nous en déduisons que $\pi = \gamma$.

7. Cas $L = L_1 + L_2$ avec $L_1, L_2 \subseteq \mathbb{SP}(\Sigma)$.
   $\mathcal{T}_{\mathcal{S}}(L) = \mathcal{T}_{\mathcal{S}}(L_1) \cup \mathcal{T}_{\mathcal{S}}(L2)$. Ici il s'agit juste d'une simple union d'ensembles de Γ-Pomsets bien construits. Ainsi l'égalité attendue en tant que pomset est préservée.

□

5.3 MSC et Γ-Pomsets

Toujours dans la lancée de notre étude comparative, nous allons nous intéresser aux Messages Sequence Charts (MSC) qui sont un modèle bien connus et souvent utilisé pour la spécification des protocoles de télécommunication. Nous avons présenté dans la section 3.3 leur formalisme et avons montré aussi comment décrire des ensembles éventuellement infinis de scénarios par des HMSC. Nous avons par ailleurs dans la même section, défini un MSC sous forme de pomset. Du moment qu'un MSC est donc un pomset, il peut être écrit en Γ-Pomset conformément à la proposition 4.2.2. Nous montrons dans cette section comment traduire un MSC en Γ-Pomset de sorte à former des ensembles clos par concaténation. Ceci nous permettra ensuite de montrer comment traduire les HMSC en Γ-Pomset.

Un Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$ dont le pomset induit est un MSC sur $\mathbb{MSC}(\Sigma_{\mathcal{P}})$ a naturellement un alphabet d'action correspondant à $\Sigma_{\mathcal{P}}$. Autrement dit $\Sigma = \Sigma_{\mathcal{P}}$. Par exemple un Γ-Pomset induisant le MSC de la figure 5.4 peut s'écrire $\langle \emptyset, p!q, 0 \rangle \langle \emptyset, q(a), 1 \rangle \langle \{0,1\}, q?p, \emptyset \rangle$

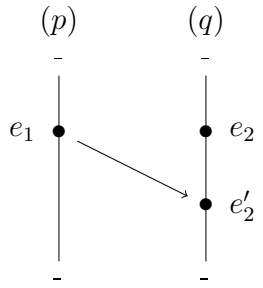


Figure 5.4: Un exemple de MSC avec $\Sigma_{\mathcal{P}} = \{p!q, q(a), q?p\}$

Parmi les nombreux Γ-Pomsets susceptibles d'induire un même MSC, nous en distingons dans ce qui suit une classe particulière que nous allons nommé et définir la structure. Il s'agit de l'ensemble des *traductions bien structurées* de MSC. En voici la définition formelle.

Définition 5.3.1 *Un Γ-Pomset $\gamma = \langle \alpha_0, a_0, \beta_0 \rangle \cdots \langle \alpha_n, a_n, \beta_n \rangle$ est une traduction bien structurée d'un MSC sur $\mathbb{MSC}(\Sigma_{\mathcal{P}})$ si les contraintes suivantes sont respectées:*

1. *$\Psi_\Gamma(\gamma)$ est un MSC sur $\mathbb{MSC}(\Sigma_{\mathcal{P}})$*
2. *$\mathcal{P} \subseteq \Gamma$*
3. *Pour tout $p \in \mathcal{P}$ et tout $i \in [0..n]$ si γ_i correspond à un événement de p (i.e $i \in E_p$) alors $\alpha_i \cap \mathcal{P} = \beta_i \cap \mathcal{P} = p$.*
4. *Pour tout $i \in [0..n]$ et tout $m \in \Gamma - \mathcal{P}$ si $m \in \alpha_i$ alors il existe une position $j < i$ telle que $m \in \beta_j$.*

Exemple 5.3.1 *Reprenant l'exemple de la figure 5.4, avec cette fois ci $\Gamma = \{p, q, 0, 1\}$, une traduction bien structurée de ce MSC est $\langle p, p!q, \{0, p\}\rangle\langle q, q(a), \{1, q\}\rangle\langle\{0, 1, q\}, q?p, q\rangle$*

Pour un MSC $M \in \mathbb{MSC}(\Sigma_{\mathcal{P}})$ nous noterons par $\mathcal{T}_{\mathcal{M}}(M) \subseteq \mathbb{P}_\Gamma(\Sigma)$ l'ensemble de ses traductions bien structurées.

Théorème 5.3.1 *Tout MSC possède une traduction bien structurée.*

Preuve:
Soit M un MSC de $\mathbb{MSC}(\Sigma_{\mathcal{P}})$. M étant un pomset, il existe un Γ-Pomset γ tel que $\Psi_\Gamma(\gamma) = M$ (application directe de la proposition 4.2.2). Posons $\gamma = \langle \alpha_0, a_0, \beta_0 \rangle \cdots \langle \alpha_n, a_n, \beta_n \rangle$ avec un ensemble de marques $\Gamma \neq \mathcal{P}$. Puis nous ajoutons l'ensemble $\mathcal{P}$ à Γ et avoir donc $\mathcal{P} \subseteq \Gamma$. Nous enrichissons ensuite les ensembles α_i et β_i de γ de la sorte: si pour tout $i \in [0..n]$, on a $\lambda^{-1}(a_i) \in E_p$ (avec $p \in \mathcal{P}$), alors on ajoute p dans α_i et dans β_i. Ainsi la contrainte 3 de la définition de la traduction bien structurée est respectée. A l'issu de ce procédé, pour chaque $i \in [0..n]$, on retire de α_i les éventuelles marques m qui ne respectent pas la contrainte 4. En effet, ces marques ne vérifiant pas cette contrainte sont inutiles et n'introduisent pas de relation de précédence. Au final γ devient une traduction bien structurée de M.

□

Avec la traduction bien structurée des MSC, nous montrons maintenant qu'il est possible de représenter la concaténation de MSC qui est une opération importante pour former des HMSC.

Proposition 5.3.1 *Soient M_1 et M_2 deux MSC de $\mathbb{MSC}(\Sigma_{\mathcal{P}})$. Soit γ_1 et γ_2 deux Γ-Pomset tels que $\gamma_1 \in \mathcal{T}_{\mathcal{M}}(M_1)$ et $\gamma_2 \in \mathcal{T}_{\mathcal{M}}(M_2)$. Alors $\gamma_1.\gamma_2 \in \mathcal{T}_{\mathcal{M}}(M_1.M_2)$.*

Preuve:
Soit $\gamma = \gamma_1.\gamma_2$. Ici le fait que $\gamma_1 \in \mathcal{T}_{\mathcal{M}}(M_1)$ et $\gamma_2 \in \mathcal{T}_{\mathcal{M}}(M_2)$ implique que les contraintes de processus (2. et 3. de la définition) sont respectées par γ. En effet si Γ_1 et Γ_2 représentent respectivement l'ensemble des marque de γ_1 et γ_2 d'une part et $\mathcal{P}_1$ et $\mathcal{P}_2$ les ensembles des processus associés à M_1 et M_2 alors:

- on a $\mathcal{P}_1 \subseteq \Gamma_1$ et $\mathcal{P}_2 \subseteq \Gamma_2$ qui provoquent que $\mathcal{P}_1 \cup \mathcal{P}_1 \subseteq \Gamma_1 \cup \Gamma_2$ et

- la contrainte 3 respectée par γ en tant que concaténation de deux pomsets respectant cette contrainte.

Vu que γ_1 et γ_2 sont bien structurés alors γ respecte la propriété 4. De ce fait il n'a pas de nouvelle relation de précédence créée entre γ_1 et γ_2 car toute marque m_2 appartenant aux ensembles de marques α de γ_2 et pouvant être candidate pour induire pour relation $\rightarrowtail_\Gamma$ avec un élément de γ_1 est inhibé par une marque m_1 appartenant à l'ensemble des marques β de γ_1. Ceci provient de la contrainte 4 de la définition. Ceci fait qu'ici $\Psi_\Gamma(\gamma_1.\gamma_2) = \Psi_\Gamma(\gamma_1).\Psi_\Gamma(\gamma_2)$ et vu que la concaténation de deux MSC est un MSC, la contrainte 1 est respectée.

Par conséquence $\gamma_1.\gamma_2 \in \mathcal{T}_\mathcal{M}(M_1.M_2)$. □

Cette propriété montre qu'on peut représenter des concaténations de MSC en Γ-Pomset . Il en découle immédiatement le corollaire suivant que l'on va énoncer en tant que théorème.

Théorème 5.3.2 *Soit L un langage de HMSC. Il existe un langage L_Γ de Γ-Pomset tel que $L_\Gamma = \mathcal{T}_\mathcal{M}(L)$.*

Rappelons que les HMSC sont des expressions rationnelles de MSC définies par l'union, la concaténation et l'étoile de Kleene $\mathbb{MSC}(\Sigma_\mathcal{P})$.

Ce théorème montre qu'on peut donc représenter un ensemble régulier de MSC en Γ-Pomsets .

Chapitre 6

Vérification avec les Γ-Pomsets

Sommaire

Nous avons montré dans le chapitre précédent qu'il est possible de manipuler des ensembles de pomsets en utilisant des structures telles que les Γ-Pomsets. Dans ce chapitre, nous nous intéressons au problème de la vérification d'une formule de logique monadique sur les pomsets en utilisant les Γ-Pomsets. Nous commencerons par formaliser la logique MSO sur les Γ-Pomsets ensuite nous étudions le problème de la satisfiabilité MSO sur les pomsets avant de montrer qu'en utilisant les Γ-Pomsets pour modéliser les systèmes parallèles, le model-checking de formule ordre partiel est décidable.

6.1 Connexion de la logique MSO aux Γ-Pomsets

A l'instar des mots sur les alphabets classiques, l'objectif de cette partie est d'établir la relation entre la logique monadique du second ordre sur les mots et les Γ-Pomsets. Il s'agira d'établir une structure permettant d'encoder les éléments constitutifs des Γ-Pomsets (les pre/post-conditions, les actions ...) par des ensembles d'entiers. Nous aurons besoin par ailleurs de traduire dans ce cadre l'importante opération de couverture $\rightarrowtail_\Gamma$, avant de formaliser le fait qu'un Γ-Pomset satisfait une formule MSO.

Dans ce qui suit, l'alphabet Σ et l'ensemble des marques Γ considérés sont supposés finis avec en notation $\Gamma = \{m_1, \ldots, m_n\}$ et $\Sigma = \{a_1, \ldots, a_k\}$.

Nous savons que, dans un Γ-Pomset, une *lettre* n'est pas un seul symbole comme dans les mots classiques. D'après la définition 4.2.1, une lettre $\langle \alpha_i, a_i, \beta_i \rangle$ est plutôt composé de trois éléments:

- α_i est l'ensemble des marques pouvant être considéré comme *pré-condition*;
- a_i est un symbole de Σ;
- β_i est un ensemble de marques pouvant être considéré comme *post-condition*.

Simuler des pré/post-conditions pour chaque position dans un Γ-Pomset par une formule de logique monadique est assez simple. Associer un symbole à une position est quant à elle classique. Nous allons présenter une série de définitions afin d'établir une structure encodant les Γ-Pomsets par seulement des entiers et des ensembles d'entiers.

Définition 6.1.1 (Γ-Pomset Symbolique) *Un Γ-Pomset Symbolique $\mathcal{S}_\Gamma$ est un tuple $\langle Pos, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma} \rangle$ où chaque élément est une variable du second-ordre, Σ un alphabet, Γ un ensemble de marques et*

- *Pos désigne un ensemble de positions (des entiers);*
- *$Pre_m \subseteq Pos$ et $Post_m \subseteq Pos$ représentent chacun un ensemble de positions liées à une marque m de Γ et respectivement en pré-condition et en post-condition.*
- *$(X_a)_{a\in\Sigma}$ représente un ensemble de positions liées à une lettre a de Σ.*

Dans la suite, nous considérons que $\mathcal{S}_\Gamma$ désigne toujours un Γ-Pomset Symbolique et donc le tuple $\langle Pos, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma} \rangle$.

Définition 6.1.2 (Instance de Γ-Pomset Symbolique) *Soit un Γ-Pomset Symbolique $\mathcal{S}_\Gamma$ et une application $\iota : VSO \mapsto 2^{\mathbb{N}}$ de l'ensemble des variables du second-ordre vers les ensembles d'entiers. Une instance de $\mathcal{S}_\Gamma$ par ι notée ${\mathcal{S}_\Gamma}^\iota$ est un tuple $\langle \iota(Pos), (\iota(Pre_m))_{m\in\Gamma}, (\iota(X_a))_{a\in\Sigma}, (\iota(Post_m))_{m\in\Gamma} \rangle$.*

Définition 6.1.3 *Soient $\gamma = \langle \alpha_0, a_0, \beta_0 \rangle \ldots \langle \alpha_n, a_n, \beta_n \rangle$ un Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$ et $\mathcal{S}_\Gamma$ un Γ-Pomset Symbolique . Nous disons qu'une instance ${\mathcal{S}_\Gamma}^\iota$ encode (ou symbolise ou traduit) un Γ-Pomset si et seulement si:*

- *$\iota(Pos) = \{0, \ldots, n\}$;*
- *pour tout $a \in \Sigma$ et tout $i \in \{0, \ldots, n\}$, $i \in \iota(X_a) \Leftrightarrow \gamma_i = \langle \alpha_i, a, \beta_i \rangle$;*
- *pour tout $m \in \Gamma$ et tout $i \in \{0, \ldots, n\}$, $i \in \iota(Pre_m) \Leftrightarrow \gamma_i = \langle \alpha_i, a_i, \beta_i \rangle \wedge m \in \alpha_i$;*
- *pour tout $m \in \Gamma$ et tout $i \in \{0, \ldots, n\}$, $i \in \iota(Post_m) \Leftrightarrow \gamma_i = \langle \alpha_i, a_i, \beta_i \rangle \wedge m \in \beta_i$.*

Pour un Γ-Pomset γ, la notation γ_S^ι désignent une instance de Γ-Pomset Symbolique qui l'encode.

Pour un Γ-Pomset symbolique γ_S^ι, l'interprétation ι peut s'exprimer en d'autres termes ainsi:

- $\iota(Pos)$ est égale à l'ensemble des positions de γ
- $\iota(X_a)$ est l'ensemble des positions où la lettre $a \in \Sigma$ apparaît.
- $\iota(Pre_m)$ est l'ensemble des positions de γ où la marque $m \in \Gamma$ apparaît en pre-condition (en marque α)
- $\iota(Post_m)$ est l'ensemble des positions de γ où la marque $m \in \Gamma$ apparaît en post-condition (en marque β)

Exemple 6.1.1 *Reprenons le Γ-Pomset* $\gamma = \langle 0,p,0\rangle\langle\{0,1\},c,1\rangle\langle 0,p,0\rangle\langle\{0,1\},c,1\rangle\langle 0,p,0\rangle\langle\{0,1\},c,1\rangle$ *représentant un système de producteur-consommateur avec* $\Gamma = \{0,1\}$ *et* $\Sigma = \{p,c\}$ *(voir section 4.3.1). Soit* $\mathcal{S}_\Gamma$ *le Γ-Pomset Symbolique suivant:* $\langle Pos, Pre_0, Pre_1, X_p, X_c, Post_0, Post_1\rangle$. *Nous pouvons donc construire la substitution* ι *telle que:*
$\iota(Pos) = \{0,1,2,3,4,5\}$, $\iota(Pre_0) = \{0,1,2,3,4,5\}$, $\iota(Pre_1) = \iota(Post_1) = \iota(X_c) = \{1,3,5\}$ *et* $\iota(Post_0) = \iota(X_p) = \{0,2,4\}$.
Le tableau suivant montre que γ_S^ι *symbolise bien* γ.

Pos	*0*	*1*	*2*	*3*	*4*	*5*
Pos_0	×	×	×	×	×	×
$Post_0$	×		×		×	
Pre_1		×		×		×
$Post_1$		×		×		×
X_p	×		×		×	
X_c		×		×		×
γ	$\langle\{0\},p,\{0\}\rangle$	$\langle\{0,1\},c,\{1\}\rangle$	$\langle\{0\},p,\{0\}\rangle$	$\langle\{0,1\},c,\{1\}\rangle$	$\langle\{0\},p,\{0\}\rangle$	$\langle\{0,1\},c,\{1\}\rangle$

Nous avions déjà énoncé dans la section 4.2.2 la proposition 4.2.1 montrant que dans l'absolu il est possible de définir un ensemble régulier de Γ-Pomsets par une formule de la logique MSO pour Σ et Γ fixé et nous venons de donner une structure permettant d'encoder les Γ-Pomsets. Une question naturelle serait maintenant de savoir s'il est possible d'extraire, depuis l'ensemble des solutions d'une formule MSO appliquée aux Γ-Pomsets, l'ensemble des pomsets qui lui sont associés. En fait, il est bien possible de le faire en traduisant par une formule MSO adéquate la relation $\rightarrowtail_\Gamma$ introduite dans la section 4.2.3 ainsi que le processus de dépliage de Γ-Pomset présenté dans la définition 4.2.3.

La définition suivante introduit la traduction de la relation $\rightarrowtail_\Gamma$.

Définition 6.1.4 ($\phi_{\rightarrowtail_\Gamma}$) *Soit* $\mathcal{S}_\Gamma$ *un Γ-Pomset Symbolique , nous définissons* $\phi_{\rightarrowtail_\Gamma}(\overrightarrow{\mathcal{S}_\Gamma})$ *où* $\overrightarrow{\mathcal{S}_\Gamma} = \langle x_i, x_j, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}\rangle$ *par la formule suivante:*

$$x_i < x_j \wedge \bigvee_{m\in\Gamma} (x_i \in Post_m \wedge x_j \in Pre_m \wedge (\forall k : (x_i < k < x_j) \Rightarrow k \notin Post_m)).$$

Partant de cette définition qui s'intéresse encore aux positions (les entiers), nous énonçons, pour un Γ-Pomset γ, la proposition suivante qui stipule que $\phi_{\rightarrowtail_\Gamma}$ simule la relation $\rightarrowtail_\Gamma$ pour tout Γ-Pomset symbolique traduisant γ.

Proposition 6.1.1 *Soit γ un Γ-Pomset, pour toute position i et j de γ, $i \rightarrowtail_\Gamma j$ si et seulement si il existe $\rho : VFO \mapsto \mathbb{N}$ et $\mu : VSO \mapsto 2^{\mathbb{N}}$ tels que $\gamma, \rho, \mu \models \phi_{\rightarrowtail_\Gamma}(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$, $\rho(x) = i$, $\rho(y) = j$ et γ_S^μ encode γ.*

Preuve:

Supposons que $i \rightarrowtail_\Gamma j$. Selon la définition 4.2.2,

$$i \rightarrowtail_\Gamma j \Leftrightarrow i < j \wedge \exists m \in \Gamma : m \in \beta_i \wedge m \in \alpha_j \wedge (\forall k \in]i, j[: m \notin \beta_k).$$

On peut déduire de manière triviale que:

$$i < j \wedge \exists m \in \Gamma : m \in \beta_i \wedge m \in \alpha_j \wedge (\forall k \in]i, j[: m \notin \beta_k) \Leftrightarrow$$
$$i < j \wedge \bigvee_{m\in\Gamma} (m \in \beta_i \wedge m \in \alpha_j \wedge (\forall k \in]i, j[: m \notin \beta_k)).$$

Soit $\mu : VSO \mapsto 2^N$ tel que γ_S^μ traduit γ.

Considérant la définition 6.1.3, on peut donc déduire que:

$$i < j \wedge \bigvee_{m\in\Gamma} (m \in \beta_i \wedge m \in \alpha_j \wedge (\forall k \in]i, j[: m \notin \beta_k)) \Leftrightarrow$$
$$i < j \wedge \bigvee_{m\in\Gamma} (i \in \mu(Post_m) \wedge j \in \mu(Pre_m) \wedge (\forall k \in]i, j[: k \notin \mu(Post_m))).$$

Et on obtient encore

$$i < j \wedge \bigvee_{m\in\Gamma} (i \in \mu(Post_m) \wedge j \in \mu(Pre_m) \wedge (\forall k \in]i, j[: k \notin \mu(Post_m))) \Leftrightarrow$$
$$i < j \wedge \bigvee_{m\in\Gamma} (i \in \mu(Post_m) \wedge j \in \mu(Pre_m) \wedge (\forall k : i < k < j \Rightarrow k \notin \mu(Post_m))).$$

Considérant ρ définie de telle sorte que $\rho(x_i) = i$ et $\rho(x_j) = j$, si on applique la définition 6.1.4 nous obtenons $\gamma, \rho, \mu \models \phi_{\rightarrowtail_\gamma}(x_i, x_j, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$.

Ce qui termine la preuve. □

Si on prend un Γ-Pomset γ, obtenir l'ensemble des pomsets depuis la formule traduisant γ peut maintenant être une tâche facile. La pierre angulaire est la construction de la relation d'odre partiel depuis la relation de couverture $\rightarrowtail_\Gamma$.

La définition suivante formalise le fait que l'ensembles des positions est stable par la relation $\rightarrowtail_\Gamma$.

Définition 6.1.5 (Fermeture de $\rightarrowtail_\Gamma$) *Soient un ensemble d'élements X et un Γ-Pomset Symbolique $\mathcal{S}_\Gamma$, nous définissons $\phi_{FP}(\overrightarrow{\mathcal{S}_\Gamma})$ avec $\overrightarrow{\mathcal{S}_\Gamma} = \langle X, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}\rangle$ comme suit:*

$$\phi_{FP}(\overrightarrow{\mathcal{S}_\Gamma}) = \quad \forall x, y : \quad (\phi_{\rightarrowtail_\Gamma}(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}) \wedge x \in X) \Rightarrow y \in X.$$

La proposition suivante assure que pour un Γ-Pomset symbolique donné, et un ensemble de positions X , la satisfiabilité de ϕ_{FP} résulte de la nature fermée de X par rapport à $\rightarrowtail_\Gamma$.

Proposition 6.1.2 *Soit γ un Γ-Pomset et μ une application $\mu : VSO \mapsto 2^{\mathbb{N}}$.*
Avec $\overrightarrow{\mathcal{S}_\Gamma} = \langle X, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}\rangle$, si $\emptyset, \mu \models \phi_{FP}(\overrightarrow{\mathcal{S}_\Gamma})$ alors $\mu(X)$ est clos par $\rightarrowtail_\Gamma$.

Preuve:
Supposons que $\emptyset, \mu \models \phi_{FP}(\overrightarrow{\mathcal{S}_\Gamma})$ et $\mu(X)$ n'est pas clos par $\rightarrowtail_\Gamma$. Donc il existe $u \in \mu(X)$ et un entier v tel que $u \rightarrowtail_\Gamma v$, $u \in \mu(X)$ et $v \notin \mu(X)$.

Selon la définition 6.1.5,

$$\forall x, y : (\phi_{\rightarrowtail_\Gamma}(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}) \wedge x \in X) \Rightarrow y \in X.$$

Par hypothèse, $\emptyset, \mu \models \phi_{FP}(\overrightarrow{\mathcal{S}_\Gamma})$. Par conséquent, v doit être un élément de $\mu(X)$. D'où l'absurdité. □

Maintenant en dernier lieu, nous donnons une définition rendant les ordres partiels définissables par une formule MSO conformément à nos structures et une proposition 6.1.3 qui va confirmer la bonne construction des ordres partiels induits par un Γ-Pomset .

Définition 6.1.6 ($\phi_\preceq$) *Soit $\mathcal{S}_\Gamma$ un Γ-Pomset Symbolique , nous définissons $\phi_\preceq(\overrightarrow{\mathcal{S}_\Gamma})$ comme suit:*

$$\phi_\preceq(\overrightarrow{\mathcal{S}_\Gamma}) = \forall X : (x \in X \wedge \phi_{FP}(X, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})) \Rightarrow y \in X)$$

avec $\overrightarrow{\mathcal{S}_\Gamma} = \langle x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}\rangle$.

Proposition 6.1.3 *Soit γ un Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$. Pour tout élément e et e' de γ respectivement à la position i et j on a: $i \preceq j$ si et seulement si il existe $\rho : VFO \mapsto \mathbb{N}$ et $\mu : VSO \mapsto 2^{\mathbb{N}}$ tel que $\gamma, \rho, \mu \models \phi_\preceq(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$, $\rho(x) = i$, $\rho(y) = j$ et γ_S^μ encode γ.*

Preuve:
Soit γ un Γ-Pomset de $\mathbb{P}_\Gamma(\Sigma)$, en suivant la définition 4.2.3, on a

$$i \preceq j \Leftrightarrow i \rightarrowtail_\Gamma^* j$$

où i et j sont deux positions de γ.

Soit μ une application telle que γ_S^μ est un Γ-Pomset symbolique traduisant γ. Etudions la formule $\phi_\preceq(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$ cas par cas selons X:

- X est vide: dans ce cas, de manière triviale, pour tout $\rho : VFO \mapsto \mathbb{N}$, $\gamma, \rho, \mu \models \phi_\preceq(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$.

- X n'est pas vide et $i \in X$: d'après la proposition 6.1.2, on peut déduire que X est clos sur $\rightarrowtail_\Gamma$. Et vu que $i \rightarrowtail_\Gamma^* j$, donc $j \in X$. Par conséquent, si on définit ρ tel que $\rho(x) = i$ et $\rho(y) = j$ nous aurons $\gamma, \rho, \mu \models \phi_\preceq(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$.
- X n'est pas vide et $i \notin X$: même chose que le premier cas.

□

Nous pouvons dire à ce stade que les définitions 4.2.1, 4.2.2 et 4.2.3 ont une traduction équivalente en logique MSO sur les mots. En d'autres termes, nous avons établi un lien entre les Γ-Pomsets et la logique MSO. Par conséquent, nous sommes maintenant en mesure de raisonner sur les pomsets en utilisant $MSO(\Sigma)$.

6.2 Satisfaction de formules MSO sur pomsets via les Γ-Pomsets

Après avoir établi, dans la section précédente, une connexion entre les pomsets et la logique MSO classique sur les mots, la prochaine étape est d'étudier le problème de satisfaction de la logique MSO sur les pomsets $MSO(\Sigma, \preceq)$. Il s'agira donc de réduire ce problème (pour rappel généralement indécidable) à un problème de satisfaction de formule sur $MSO(\Sigma)$. Notre contribution finale dans ce domaine sera de proposer un semi-algorithme pour décider ou non la satisfaction d'une formule.

6.2.1 Satisfaction de formule $MSO(\Sigma, \preceq)$

Définition 6.2.1 *Soient Σ un alphabet, Γ un ensemble de marques et ϕ une formule de $MSO(\Sigma, \preceq)$. Nous définissons ϕ^Γ comme la formule de $MSO(\Sigma)$ obtenue en appliquant la fonction* 2MSO *sur ϕ. Cette fonction est définie comme suit:*

$$\mathsf{2MSO}(\phi) = \begin{cases} \phi & si\ \phi = x \in X\ ou\ \phi = P_a(x) \\ \phi_\preceq(\overrightarrow{\mathcal{S}_\Gamma})\ avec\ \overrightarrow{\mathcal{S}_\Gamma} = & \\ \langle x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma}\rangle & si\ \phi = x \preceq y \\ \neg\mathsf{2MSO}(\psi) & si\ \phi = \neg\psi \\ \mathsf{2MSO}(\psi) \wedge \mathsf{2MSO}(\psi') & si\ \phi = \psi \wedge \psi' \\ \mathsf{2MSO}(\psi) \vee \mathsf{2MSO}(\psi') & si\ \phi = \psi \vee \psi' \\ (\forall x)\mathsf{2MSO}(\phi') & si\ \phi = (\forall x)\phi' \\ (\forall X)\mathsf{2MSO}(\phi') & si\ \phi = (\forall X)\phi' \\ (\exists x)\mathsf{2MSO}(\phi') & si\ \phi = (\exists x)\phi' \\ (\exists X)\mathsf{2MSO}(\phi') & si\ \phi = (\exists X)\phi' \end{cases}$$

Remarque: $\phi = P_a(x)$ peut être remplacé par $\phi = x \in X_a$.

Cette définition présente ainsi la transformation d'une formule de $MSO(\Sigma, \preceq)$ vers une formule de $MSO(\Sigma)$. Les solutions de la formule MSO classique sur les mots ainsi obtenus

peuvent être interprétées comme des Γ-Pomsets dont les pomsets associés sont les solutions de la formule initiale de $MSO(\Sigma, \preceq)$.

Proposition 6.2.1 *Soient* Γ *et* Σ *respectivement un ensemble de marques et un alphabet. Soit* ϕ *une formule de* $MSO(\Sigma, \preceq)$*. Si* ϕ^Γ *est satisfiable, alors* ϕ *est aussi satisfiable.*

Preuve:
La différence principale entre ϕ^Γ et ϕ se situe au niveau des positions de ϕ où les atomes de la forme $x \preceq y$ sont présents. En effet, selon la définition 6.2.1, ces atomes sont substitués dans ϕ^Γ par $\phi_\preceq(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$. Définissons deux substitutions $\rho : VFO \mapsto \mathbb{N}$ and $\mu : VSO \mapsto 2^{\mathbb{N}}$ telles que $\rho, \mu \models \phi_\preceq(x, y, (Pre_m)_{m\in\Gamma}, (X_a)_{a\in\Sigma}, (Post_m)_{m\in\Gamma})$. D'après la proposition 6.1.3, on peut déduire que $\rho(x) \preceq \rho(y)$. Ce qui fait que, $x \preceq y$ est aussi satisfiable.

Par induction, on peut facilement déduire que si ϕ^Γ est satisfiable alors ϕ l'est aussi. □

A ce stade, il est clair que la recherche de solution est dirigée par les Γ-Pomsets et plus précisément l'ensemble des marques Γ qui est directement impliqué dans la construction des ordres partiels (voir en particulier la définition 6.1.4). Par conséquence, pour un Γ donné, l'ensemble des ordres partiels pouvant être généré peut ne pas être suffisant pour la satisfaction de ϕ^Γ.

Nous allons maintenant donner notre résultat principal concernant le problème de la satisfiabilité.

Théorème 6.2.1 *Soit* ϕ *une formule* $MSO(\Sigma, \preceq)$*. S'il existe* Γ *tel que* ϕ^Γ *est satisfiable alors* ϕ *l'est aussi.*

Preuve:
Conséquence directe de la proposition 6.2.1. □

6.2.2 Semi-algorithme de décision

Partant de ce résultat, il nous est donc possible d'établir un algorithme pouvant implémenter la procédure de décision proposée pour la satisfaction d'une formule $MSO(\Sigma, \preceq)$. Nous présentons donc cet algorithme.

Avec ϕ une formule de $MSO(\Sigma, \preceq)$ donnée, $\mathsf{search}(\phi)$ est définie par l'algorithme 6.2.2.

On commence par un ensemble de marques Γ vide. On exécute $hasNoValidSolution$ en lui passant la formule ϕ^Γ (obtenu en appliquant $\mathsf{2MSO}(\phi)$).

$hasNoValidSolution$ est une fonction qui cherche, par rapport aux Γ-Pomset Symbolique , les Γ-Pomsets générés eu égard au nombre de marques choisi ($|\Gamma|$), s'il y a un Γ-Pomset (solution) satisfaisant la formule ψ reçue. Si on n'a pas de solution, on ajoute une nouvelle marque à Γ et on reprend le test $hasNoValidSolution$. On reprend le même procédé tant

Algorithm 1 search(ϕ)

Variables
(* Initialisation de l'ensemble des marques *)
$\Gamma := \emptyset$;
$\psi := \phi^{\Gamma}$;
00 ***Begin***
01 **While** (hasNoValidSolution(ψ)) **do**
02 $\Gamma := \Gamma \cup \{m_k\}$ with $k = |\Gamma|$;
03 $\psi := \phi^{\Gamma}$;
04 **EndWhile**
05 return true;
06 ***End***

qu'on n'a pas de solution valide. Le programme s'arrête quand il y a une solution. On renvoie dans ce cas la réponse positive.

Nous allons donner un exemple qui va illustrer le fonctionnement de l'algorithme afin de mieux le comprendre.

6.3 Exemple d'expérimentation

Nous avons choisi l'outil MONA[HJJ+95, KM01] pour expérimenter notre algorithme. Ce choix est naturellement guidé par le fait que cet outil implémente la logique MSO.

6.3.1 Présentation de l'outil MONA

MONA est une implémentation de procédures de décision pour les logiques *WS1S (Weak Second-order Logic of One Successor)* et *WS2S (Weak Second-order of Two Successor)* qui sont des variantes de la logique MSO [Tho90]. WS1S est une variante où on interprète les variables de premier ordre par des entiers naturels, qui peuvent être comparés et sujets à une opération d'addition. Les variables de second ordre sont interprétées comme des sous-ensembles finis de $\mathbb{N}$. WS2S est une généralisation de WS1S interprétée sur un arbre binaire fini ou infini.

MONA traduit toute formule WS1S ou WS2S en un automate fini reconnaissant les mots ou arbres qui valident la formule. Ces logiques ont été pendant longtemps connus pour être décidables, mais avec une complexité non élémentaire décourageant une implémentation utile. MONA utilise les *BDD* (Binary Decision Diagrams) [Ake78] pour représenter les fonctions de transitions d'automates; ce qui réduit la complexité car les BDD sont particulièrement adaptés aux automates [Kla98].

Un programme MONA consiste en un ensemble de déclarations, de formules et de macros écrit sur un fichier .mona qui décrit une formule WS1S ou WS2S. Dans ce fichier certains symboles mathématiques et connecteurs logiques tels que: $\neg, \wedge, \vee, \Rightarrow, \Leftrightarrow, \exists x, \forall x, \exists X, \forall X, \leq, \geq, \subseteq, \in, \notin, \emptyset, \cup, \cap$ sont traduits en leur équivalent texte, soit respectivement: $\sim, \&, |, =>$

, <=>, ex1 x, all1 x, ex2 X, all2 X, <=, >=, sub, in, notin, empty,
union, inter.

Pour simplifier l'écriture de formules complexes, MONA propose des macros ou des prédicats. Les prédicats sont précédés par le mot clé *pred*. Il est possible d'écrire des commentaires (non interprétés donc) dans un programme MONA en les faisant précéder du caractère #.

Le résultat de la compilation est un automate, qui peut être utilisé selon les désirs du programmeur MONA. Si la formule en entrée est satisfaite des exemples satisfaisants sont générés sinon des contres exemples sont présentés.

Voici un petit exemple pour illustrer les fonctionnalités de base de MONA.

Exemple 6.3.1 *Le programme de la figure ci-dessous (qu'on peut écrire dans un fichier* simple.mona*) se voudrait donner une formule qui soit vraie si les variables X_a et X_b modélisent bien un mot de la forme $a * b$.*

```
#Déclaration des variables libres :
# - var1 pour les variables du premier ordre
# - var2 pour les variables du second ordre
var2 Xa, Xb;

#Déclaration et définition du prédicat
pred formeDesMotsAcceptes(var2 X, Y) =
ex1 y: y in Y & y notin X
& all1 x: (
((x < y) => (x in X))
&
((x >y) => (x notin (Xa union Xb)))
)
& X inter Y = empty;

#la formule à tester avec la contrainte que Xb={3}.
formeDesMotsAcceptes(Xa,Xb) & Xb={3} ;
```

Le langage associé à simple.mona *est l'ensemble des mots finis (qui sont en fait des suites binaires) correspondant à des interprétations satisfaisantes. Ce langage est régulier, et peut donc être reconnu par un automate état-fini.*

MONA est capable d'analyser simple.mona *automatiquement en le traduisant dans un automate minimum reconnaissant l'ensemble des interprétations satisfaisantes. La commande* mona simple.mona *produit l'automate, l'analyse et génère la sortie suivante:*

```
100% completed
Time: 00:00:00.00

Automaton has 7 states and 16 BDD-nodes

ANALYSIS

A counter-example of least length (0) is:
Xa              X
Xb              X
```

```
Xa = {}
Xb = {}

A satisfying example of least length (4) is:
Xa              X 1110
Xb              X 0001

Xa = {0,1,2}
Xb = {3}

Total time: 00:00:00.00
```

Cette analyse nous dit qu'un contre exemple de cette formule est obtenu en interprétant à la fois X_a et X_b comme des ensembles vides. MONA a aussi calculé une interprétation satisfaisante qui est $X_a = \{0, 1, 2\}$, $X_b = \{3\}$. La formule est donc satisfiable.

6.3.2 Expérimentation de l'algorithme sous MONA

Nous donnons un exemple d'implémentation de l'algorithme 6.2.2. On va donc regarder les nombre de marques pouvant suffire pour décider qu'une formule $MSO(\Sigma, \preceq)$ est satisfiable.

Considérons pour cela la formule suivante:

$$\phi = \exists x_1, x_2, x_3, x_4 . x_1 \lessdot x_4 \wedge x_2 \lessdot x_4 \wedge x_3 \lessdot x_4 \wedge \bigwedge_{i,j \in \{1,\ldots,4\} \wedge i \neq j} (x_i \neq x_j).$$

Elle traduit simplement l'existence d'un pomset avec 3 événements parallèles immédiatement suivis d'un autre événement (une sorte de *join*). Notons que l'alphabet Σ n'est pas important dans cet exemple, c'est pourquoi nous ignorons l'utilisation de variables qui lui seraient liées.

On commence d'abord par traduire ϕ en ϕ^Γ en appliquant $2MSO(\phi)$ (définition 6.2.1). Nous choisissons de commencer par deux marques. Ensuite on soumet ϕ^Γ à MONA. Ce qui donne le programme joinSystem2Mark.mona de la figure 6.1. Dans ce programme nous avons défini un certain nombre de prédicat pour faciliter l'écriture de notre formule. Nous avons ainsi les prédicats:

- *relationGPomset* qui est une implémentation de la traduction de la relation de couverture sur les Γ-Pomsets i.e. la relation $\phi_{\rightarrowtail_\Gamma}$ présentée dans la définition 6.1.4,
- *partialOrder* une implémentation de la traduction de la relation d'ordre partiel déjà défini en tant que fermeture transitive de la relation $\phi_{\rightarrowtail_\Gamma}$ (voir en définition 6.1.6 la relation $\phi_\preceq$),
- *cover* qui est la traduction de la relation de couverture sur les pomsets i.e. la relation $\lessdot$ utilisée dans la formule à tester.

Ces prédicats seront réutilisés pour les autres programmes. Le dernier prédicat *joinsystem* implémente une traduction de la formule ϕ à vérifier en utilisant les prédicats déclarés.

```
var2 Pre0; # L'ensemble des positions où la marque 0 apparaît en pré-condition
var2 Pre1; # L'ensemble des positions où la marque 1 apparaît en pré-condition
var2 Post0; # L'ensemble des positions où la marque 0 apparaît en post-condition
var2 Post1; # L'ensemble des positions où la marque 1 apparaît en post-condition


# Traduction de la relation de couverture sur les |-Pomset
pred relationGPomset(var1 x, y, var2 Pr0, Pr1, Pt0, Pt1) =
 x < y &
 (
   (x in Pt0 & y in Pr0 & (all1 z:(z>x & z<y)=> z notin Pt0)) |
   (x in Pt1 & y in Pr1 & (all1 z:(z>x & z<y)=> z notin Pt1))
 )
;

#Traduction de la relation d'ordre partiel
pred partialOrder(var1 x, y, var2 Pr0, Pr1, Pt0, Pt1) =
 all2 X:((x in X & (all1 y,z:(relationGPomset(y,z,Pr0,Pr1,Pt0,Pt1) & y in X)
                      => z in X)) => y in X)
;

#Traduction de la relation de couverture sur les pomsets
pred cover(var1 x, var1 y, var2 Pr0, var2 Pr1, var2 Pt0, var2 Pt1) =
partialOrder(x,y,Pr0,Pr1,Pt0,Pt1) & ~(ex1 z : z ~= x & z ~=y &
                 (partialOrder(x,z,Pr0,Pr1,Pt0,Pt1) &
                   partialOrder(z,y,Pr0,Pr1,Pt0,Pt1)
                  )
         );

#Traduction de la formule considérée
pred joinsystem(var2 Pr0, Pr1, Pt0, Pt1) =
 ex1 x0,x1,x2,x3:(x1~=x2 & x1~=x3 & x2~=x3  & x1~=x0 & x0~=x2 & x0~=x3 &
              cover(x1,x0,Pr0,Pr1,Pt0,Pt1) &
              cover(x2,x0,Pr0,Pr1,Pt0,Pt1) &
              cover(x3,x0,Pr0,Pr1,Pt0,Pt1)
             )
;

joinsystem(Pre0,Pre1,Post0,Post1);
```

Figure 6.1: Spécification Mona de l'exemple avec deux marques

Une fois exécuté, MONA donne, pour l'automate généré par notre programme, l'analyse suivante:

```
AUTOMATON CONSTRUCTION


100% completed
Time: 00:00:00.00
```

```
Automaton has 1 state and 1 BDD-node

ANALYSIS
Formula is unsatisfiable

A counter-example of least length (0) is:
Pre0            X
Pre1            X
Post0           X
Post1           X

Pre0 = {}
Pre1 = {}
Post0 = {}
Post1 = {}

Total time: 00:00:00.00
```

On constate que la formule est insatisfiable, autrement dit qu'il n'y a pas de Γ-Pomset qui engendre un pomset satisfaisant la formule. Nous ajoutons donc une troisième marque à Γ conformément l'algorithme 6.2.2 et demandons à nouveau à MONA une solution. Le programme est celui de la figure 6.2 (joinSystem3Mark.mona). Les prédicats sont les mêmes sauf qu'on intègre les paramètres de la nouvelle marque ajoutée.

La formule ϕ^{Γ} est maintenant satisfiable avec ce programme. Nous donnons ci-dessous l'analyse MONA de l'automate produit.

```
MONA v1.4-13 for WS1S/WS2S
Copyright (C) 1997-2008 BRICS

AUTOMATON CONSTRUCTION
100% completed
Time: 00:00:00.01

Automaton has 48 states and 787 BDD-nodes

ANALYSIS

Pre0 = {3}
Pre1 = {3}
Pre2 = {3}
Post0 = {2}
Post1 = {1}
Post2 = {0}
```

En interprétant ce résultat, on peut construire le Γ-Pomset suivant:

$$\langle\emptyset, 0, \{2\}\rangle\langle\emptyset, 1, \{1\}\rangle\langle\emptyset, 2, \{0\}\rangle\langle\{0, 1, 2\}, 3, \emptyset\rangle.$$

A noter que par simple convenance, nous avons mis les positions pour remplacer les symboles de Σ qui sont, rappelons le, non considérés ici.

D'après le théorème 6.2.1, ϕ est aussi satisfiable. Figure 6.3 donne une représentation du pomset généré par le Γ-Pomset défini précédemment.

```
var2 Pre0; # L'ensemble des positions où la marque 0 apparaît en pré-condition
var2 Pre1; # L'ensemble des positions où la marque 1 apparaît en pré-condition
var2 Pre2; # L'ensemble des positions où la marque 2 apparaît en pré-condition
var2 Post0; # L'ensemble des positions où la marque 0 apparaît en post-condition
var2 Post1; # L'ensemble des positions où la marque 1 apparaît en post-condition
var2 Post2; # L'ensemble des positions où la marque 2 apparaît en post-condition

pred relationGPomset(var1 x, y, var2 Pr0, Pr1, Pr2, Pt0, Pt1, Pt2) =
 x < y &
 (
   (x in Pt0 & y in Pr0 & (all1 z:(z>x & z<y)=> z notin Pt0)) |
   (x in Pt1 & y in Pr1 & (all1 z:(z>x & z<y)=> z notin Pt1)) |
   (x in Pt2 & y in Pr2 & (all1 z:(z>x & z<y)=> z notin Pt2))
 )
;

pred partialOrder(var1 x, y, var2 Pr0, Pr1, Pr2, Pt0, Pt1, Pt2) =
 all2 X:((x in X & (all1 y,z:(relationGPomset(y,z,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2) & y in X)
                      => z in X)) => y in X)
;

pred cover(var1 x, y, var2 Pr0, Pr1, Pr2, Pt0, Pt1, Pt2) =
partialOrder(x,y,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2) & ~(ex1 z : z ~= x & z ~=y &
                 (partialOrder(x,z,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2) &
                   partialOrder(z,y,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2)
                  )
         );

pred joinsystem(var2 Pr0, Pr1, Pr2, Pt0, Pt1, Pt2) =
 ex1 x0,x1,x2,x3:x1~=x2 & x1~=x3 & x2~=x3 & x1~=x0 & x0~=x2 & x0~=x3 &
              cover(x1,x0,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2) &
              cover(x2,x0,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2) &
              cover(x3,x0,Pr0,Pr1,Pr2,Pt0,Pt1,Pt2)
;

joinsystem(Pre0,Pre1,Pre2,Post0,Post1,Post2);
```

Figure 6.2: Spécification Mona de l'exemple avec trois marques

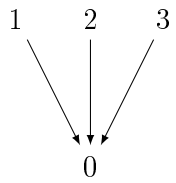


Figure 6.3: Une solution de la formule ϕ de l'exemple

Il est bien sûr possible de coder dans un langage de programmation l'automatisation du processus d'incrémentation des marques et la soumission à la spécification MONA à chaque tour jusqu'à obtention d'un résultat positif. Pour notre part, nous avons réalisé un programme en langage JAVA (de 120 lignes) pour cela. Bien entendu, ce programme ne termine pas toujours puisqu'on est dans une implémentation d'un semi-algorithme.

6.4 Application à la vérification de système

Au-delà de la portée de la logique, MSO constitue un outil pour l'analyse et la vérification de systèmes. En effet, les logiques sont adaptées pour concevoir aussi bien des propriétés que le système lui même. La connexion établie entre MSO et la théorie des automates a fourni beaucoup de techniques pour la vérification des systèmes.

Maintenant considérons les Γ-Pomsets comme un outil de spécification. En effet, il est bien connu que les ordres partiels (plus précisément les pomsets) sont utiles pour la spécification des systèmes concurrents. Supposons un système S spécifié par θ une expression régulière des Γ-Pomsets.

Soit p une propriété spécifiée par une formule ϕ dans $MSO(\Sigma, \preceq)$. La question est $S \stackrel{?}{\models} p$ (est-ce que S satisfait la propriété p?). En réalité, $S \models p$ si pour tout pomset $(E, \preceq, \lambda)$ généré par θ, $(E, \preceq, \lambda) \models \phi$.

Naturellement, dans ce cas, le problème de la vérification est décidable avec la logique MSO puisque l'ensemble des pomsets est défini par le système lui-même ainsi spécifié en Γ-Pomsets. Ce résultat est donné dans le théorème suivant.

Théorème 6.4.1 *Soient Γ un ensemble fini de marques et Σ un alphabet fini. Soit p une propriété spécifiée dans $MSO(\Sigma, \preceq)$. Soit S un système spécifié par un Γ-Pomset dans Σ. Le problème de vérification $S \stackrel{?}{\models} p$ est décidable.*

Preuve:
On sait que tout Γ-Pomset σ peut être défini en MSO (depuis la proposition 4.2.1). Soit Γ pris de sorte que $\Gamma = \{m_1, \ldots, m_n\}$. Supposons S défini par la formule MSO $\phi(Pos, (Pre_{m_i})_{i=1,\ldots,n}, (X_a)_{a\in\Sigma}, (Post_{m_i})_{i=1,\ldots,n})$.

Considérons maintenant la formule p^Γ obtenue avec la définition 6.2.1. Cette formule est aussi définie sur le même ensemble de variables libres. Par conséquence, si $\phi(Pos, (Pre_{m_i})_{i=1,\ldots,n}, (X_a)_{a\in\Sigma}, (Post_{m_i})_{i=1,\ldots,n}) \wedge \neg p^\Gamma$ est insatisfiable alors il n'existe pas un pomset $\langle E, \preceq, \lambda\rangle$ généré par σ de sorte que $\langle E, \preceq, \lambda\rangle \models \neg p$. Ainsi, S satisfait p.

Supposons maintenant que $\phi(Pos, (Pre_{m_i})_{i=1,\ldots,n}, (X_a)_{a\in\Sigma}, Post_{m_i})_{i=1,\ldots,n}) \wedge \neg p^\Gamma$ est satisfiable. Alors, il existe un pomset $\langle E, \preceq, \lambda\rangle$ généré par σ de sorte que $\langle E, \preceq, \lambda\rangle \models \neg p$. Par conséquence, on peut déduire que S ne satisfait pas p.

□

Remarque: Vu que nous nous ramenons à la logique MSO sur les mots, la complexité est non-élémentaire. Il est cependant possible de se restreindre à des fragments MSO dont la complexité est PSPACE. Nous n'étudions pas cet aspect ici.

Pour illustrer notre technique, nous reprenons l'exemple bien connu du système producteur-consommateur. Un système producteur-consommateur peut être modélisé en pomset comme nous l'avons vu précédemment (exemple 2.3.2). Comme nous l'avons montré ensuite dans la section 4.3.1 il est possible de généraliser la spécification d'un tel système à tout nombre de consommateurs et producteurs en utilisant un Γ-Pomset avec deux marques $\Gamma = \{0, 1\}$. Rappelons encore que l'expression régulière encodant ce système est la suivante: $(\langle 0, p, 0\rangle\langle\{0, 1\}, c, 1\rangle)^*$.

Selon la proposition 4.2.1, cette expression régulière peut être encodée par la formule MSO ϕ_S donnée juste en dessous. Les variables P et C spécifient respectivement les évènements de production et ceux de consommation. $\langle Pos, Pr_0, Pr_1, P, C, Pt_0, Pt_1\rangle$ est un Γ-Pomset Symbolique.

$$
\begin{aligned}
&\phi_S(Pos, Pr_0, Pr_1, P, C, Pt_0, Pt_1) = \\
&\quad (0 \in Pos \Leftrightarrow 0 \in P) \wedge \\
&\quad (\forall i : (i \in P \wedge i \in Pos) \Rightarrow ((i+1) \in Pos \wedge (i+1) \in C)) \wedge \\
&\quad (\forall i : (i \in C \wedge i + 1 \in Pos) \Rightarrow (((i+1) \in P)) \wedge \\
&\quad (\forall i : (i \in P \wedge i \in Pos) \Rightarrow (i \in (Pr_0 \cap Pt_0) \wedge i \notin (Pr_1 \cup Pt_1))) \wedge \\
&\quad (\forall i : (i \in C) \Rightarrow (i \in (Pr_0 \cap Pr_1 \cap Pt_1) \wedge i \notin Pt_0)) \wedge \\
&\quad (O \in Pos \Leftrightarrow \exists f.f \in C \wedge (\forall x : x > f \Rightarrow x \notin P \cup C))
\end{aligned}
$$

Maintenant que le système est spécifié, il est possible d'en vérifier des propriétés. Un exemple de propriété pourrait être celui-ci: "deux productions sont consommées selon leur ordre de production". Logiquement parlant, on peut l'exprimer par la formule $MSO(\Sigma, \preceq)$ suivante:

$$
\begin{aligned}
&prop1 = \forall x_1, x_2, y_1, y_2. \\
&\quad x_1 \lessdot y_1 \wedge x_2 \lessdot y_2 \wedge \lambda(x_1) = p \wedge \lambda(x_2) = p \wedge \lambda(y_1) = c \wedge \lambda(y_2) = c \\
&\quad \Rightarrow ((x_1 \preceq x_2 \wedge y_1 \preceq y_2) \vee (x_2 \preceq x_1 \wedge y_2 \preceq y_1)).
\end{aligned}
$$

Par rapport à la définition 6.2.1, on peut générer la formule sur des Γ-Pomsets donnés ci-dessous:

$$
\begin{aligned}
&\phi_p(Pr_0, Pr_1, P, C, Pt_0, Pt_1) = \forall x_1, x_2, y_1, y_2. \\
&\quad (\phi_{\lessdot}(x_1, y_1, Pr_0, Pr_1, P, C, Pt_0, Pt_1) \wedge \phi_{\lessdot}(x_2, y_2, Pr_0, Pr_1, P, C, Pt_0, Pt_1) \wedge \\
&\quad x_1 \in P \wedge x_2 \in P \wedge y_1 \in C \wedge y_2 \in C) \\
&\quad \Rightarrow \\
&\quad ((\phi_{\preceq}(x_1, x_2, Pr_0, Pr_1, P, C, Pt_0, Pt_1) \wedge \phi_{\preceq}(y_1, y_2, Pr_0, Pr_1, P, C, Pt_0, Pt_1)) \vee \\
&\quad (\phi_{\preceq}(x_2, x_1, Pr_0, Pr_1, P, C, Pt_0, Pt_1) \wedge \phi_{\preceq}(y_2, y_1, Pr_0, Pr_1, P, C, Pt_0, Pt_1)))
\end{aligned}
$$

avec $\phi_{\lessdot}(x, y, Pr0, Pr1, P, C, Pt0, Pt1) = \phi_{\preceq}(x, y, Pr0, Pr1, P, C, Pt0, Pt1) \wedge \neg\exists z.z \neq x \wedge z \neq y \wedge \phi_{\preceq}(x, z, Pr0, Pr1, P, C, Pt0, Pt1) \wedge \phi_{\preceq}(y, z, Pr0, Pr1, P, C, Pt0, Pt1)$.

Grâce à Mona, on peut montrer que la formule suivante

$$\phi_S(Pos, Pr_0, Pr_1, P, C, Pt_0, Pt_1) \wedge \neg(\phi_p(Pos, Pr_0, Pr_1, P, C, Pt_0, Pt_1))$$

n'est pas satisfiable. Conséquemment, la propriété p est satisfaite par le système S.

```
var2 Pre0; # L'ensemble des positions où la marque 0 apparaît en pré-condition
var2 Pre1; # L'ensemble des positions où la marque 1 apparaît en pré-condition
var2 Post0; # L'ensemble des positions où la marque 0 apparaît en post-condition
var2 Post1; # L'ensemble des positions où la marque 1 apparaît en post-condition
var2 P; # L'ensemble des positions où l'événement P apparaît
var2 C; # L'ensemble des positions où l'événement P apparaît
var2 Pos; # L'ensemble de toutes les positions

allpos Pos;


pred relationGPomset(var1 x, y, var2 Pr0, Pr1, Pt0, Pt1) =
 x < y &
 (
   (x in Pt0 & y in Pr0 & (all1 z:(z>x & z<y)=> z notin Pt0)) |
   (x in Pt1 & y in Pr1 & (all1 z:(z>x & z<y)=> z notin Pt1))
 )
;

pred partialOrder(var1 x, y, var2 Pr0, Pr1, Pt0, Pt1) =
 all2 X:((x in X & (all1 y,z:(relationGPomset(y,z,Pr0,Pr1,Pt0,Pt1) & y in X)
                      => z in X)) => y in X)
;

pred cover(var1 x, var1 y, var2 Pr0, var2 Pr1, var2 Pt0, var2 Pt1) =
partialOrder(x,y,Pr0,Pr1,Pt0,Pt1) & ~(ex1 z : z ~= x & z ~=y &
                 (partialOrder(x,z,Pr0,Pr1,Pt0,Pt1) &
                   partialOrder(z,y,Pr0,Pr1,Pt0,Pt1)
                  )
         )
;

pred prop1(var2 P, C, Pr0, Pr1, Pt0, Pt1) =
all1 x1,x2,y1,y2 :
     ((x1 in P ) & (x2 in P) & (y1 in C)  & (y2 in C) &
                cover(x1,y1,Pr0,Pr1, Pt0,Pt1) & cover(x2,y2,Pr0,Pr1, Pt0,Pt1)) =>
 ((partialOrder(x1,x2,Pr0,Pr1,Pt0,Pt1) &    partialOrder(y1,y2,Pr0,Pr1,Pt0,Pt1)) |
       (partialOrder(x2,x1,Pr0,Pr1,Pt0,Pt1) & partialOrder(y2,y1,Pr0,Pr1,Pt0,Pt1)))

;

pred pcsystem(var2 P, C, Pr0, Pr1, Pt0, Pt1) =
 (0 in Pos <=> 0 in P) &
 (all1 i: (i in P & i in Pos) => (i+1 in Pos & i+1 in C)) &
 (all1 i: (i+1 in Pos & i in C) => i+1 in P) &
 (all1 i: i in P & i in Pos  => (i in (Pr0 inter Pt0)
                                & ~(i in (Pr1 union Pt1)))) &
 (all1 i: i in C => (i in (Pr0 inter (Pr1 inter Pt1)) & ~(i in Pt0)))&
 (0 in Pos <=> ex1 i: i in C & (all1 x: x>i =>
         x notin (P union C union Pr0 union Pr1 union Pt0 union Pt1)))
;


pcsystem(P,C,Pre0,Pre1,Post0,Post1) & ~prop1(P,C,Pre0,Pre1,Post0,Post1);
```

Figure 6.4: Specification Mona Producteur/Consommateur

La totalité de la spécification MONA est donnée dans la figure 6.4.

Le prédicat *pcsystem* est la traduction de la spécification du système alors que *prop*1 est celle de la propriété à vérifier.

Chapitre 7

Conclusion et perspectives

Sommaire

7.1 Synthèse de nos travaux

Nos travaux dans cette thèse ont concerné le domaine de la modélisation et vérification de systèmes concurrents. Nous nous sommes spécialement intéressés aux ensembles partiellement ordonnés étiquetés (pomsets) et aux problèmes de vérification qu'ils ont soulevés et principalement au problème d'indécidabilité de la satisfaction d'une formule MSO sur pomset (que nous avons prouvé dans le premier chapitre).

Nous avons proposé un nouveau modèle appelé Γ-Pomset qui est une structure pouvant encoder tout pomset fini en mots. Nous avons défini formellement un Γ-pomset comme un mot sur un alphabet $2^\Gamma \times \Sigma \times 2^\Gamma$ où Γ est un ensemble fini de *marques* et Σ est l'alphabet étiquetant les événements des pomsets et de telle sorte que les relations d'ordres puissent être encodées. C'est une méthode assez compacte pour encoder des ensembles finis ou infinis d'ordres partiels dont les linéarisations n'ont pas forcément besoin de former des langages réguliers.

Notre structure peut être utilisée purement pour la modélisation de systèmes concurrents et nous avons établi ses équivalences avec les modèles classiques de la concurrence qui sont liés aux ordres partiels. L'étude de ces équivalences nous a permis de constater que les Γ-Pomset ont un bon pourvoir d'expression parce que capable de traduire des ensembles des modèles classiques telles que les traces de Mazurkiewicz, les Message Sequence Charts et les pomsets série-parallèle.

L'avantage des Γ-Pomsets est de permettre de ramener le problème de vérification avec les pomsets à un problème de vérification avec les mots. Dans ce domaine, nous nous sommes intéressés au model-checking et à la satisfiabilité de formules MSO sur pomset (qui est indécidable en général).

Nous avons mis en oeuvre une méthode de transformation de formule MSO sur ordre partiel en MSO sur les mots. Nous avons ensuite donné une condition suffisante pour la satisfaction d'une formule MSO ϕ sur pomset: s'il existe un nombre de marques pour lequel la satisfaction d'une formule MSO sur les mots ϕ^Γ (construite à partir de ϕ pour les Γ-Pomsets) est vérifiée alors ϕ est satisfiable. Ce résultat nous a permis de donner un semi-algorithme qui augmente tour à tour l'ensemble des marques pour vérifier la satisfaction de formule.

Les Γ-Pomsets étant aussi des mots pouvant être utilisés comme langage de spécification de systèmes concurrents, on a établi le fait que pour un système S spécifié en Γ-Pomset γ et une formule logique ordre partiel ϕ dénotant une propriété p, le problème de vérification $S \stackrel{?}{\models} p$ devient décidable.

Les systèmes décrits à l'aide d'autres modèles classiques de la concurrence tels que les MSC, peuvent aussi être vérifiés avec les Γ-Pomsets via les transformations que nous avons données.

Nous avons enfin utilisé l'outil MONA pour implémenter notre procédure et tester le model-checking de formule ordre partiel sur les systèmes concurrents.

Nos travaux constituent une base ouvrant beaucoup de nouvelles pistes de recherche à explorer. Nous les décrivons dans la section suivante.

7.2 Perspectives envisagées

Les perspectives envisagées pour la suite de nos travaux s'inscrivent aussi bien dans le cadre théorique que pratique.

Tout d'abord nous savons qu'on peut encoder tout pomset fini en Γ-Pomset . A partir de là, il est naturel de se demander quelles sont les classes de pomsets infinis qu'on peut représenter. D'où un travail de caractérisation de ces classes. Nos résultats en vérification resteront valables par ailleurs pour les pomsets infinis ainsi traduits en Γ-Pomset ; MSO étant décidable aussi bien sur les mots finis que sur les mots infinis.

Nous avons certes traduit quelques modèles de la concurrence en Γ-Pomset , mais nous pensons qu'on pourrait en traduire davantage, notamment les modèles à base d'automates asynchrones comme les $\vec{\Sigma} - ACA$. Nous n'avons pas abordé le modèle des réseaux de Petri mais nous pensons que tout au moins le cas des réseaux de Petri sains est facile à traduire; ces derniers sont par ailleurs encore liés aux traces.

L'une des perspectives les plus intéressantes et immédiates est la réalisation d'un model-checker pour la vérification automatique de systèmes. Nos travaux ont posé les bases et le prototype pour un tel outil. Pour cela il y a certes un travail d'implémentation à faire mais aussi un travail d'optimisation. Le fait de maintenir la logique MSO ne pourra, dans ce contexte, pas être adéquat car on sera vite freiné par des problèmes de complexité trop grande. Il faudrait plutôt se retourner vers des logiques temporelles sur lesquelles on va pouvoir améliorer les constructions d'automates et se ramener à une complexité PSPACE. Il est probable que les récents travaux de Paul Gastin et Dietrich Kuske sur la satisfiabilité LTL sur les traces [GK10] puissent être généralisés à cet effet sur les Γ-Pomsets . Il est aussi envisageable, toujours pour la performance, d'intégrer des techniques d'optimisation telles que les calculs symboliques sur les modèles, et les méthodes de réduction ordre partiel. Nous

pensons que notre modèle est bien adapté à ces techniques puisqu'il traite des ordres partiels.

Un model-checker ainsi réalisé pourrait aussi fonctionner pour chacun des modèles classiques traduisibles en Γ-Pomset . Pour cela il faudrait implémenter les mécanismes de traduction proposés. L'amélioration éventuelle de ces mécanismes est aussi permises.

Nous estimons de même qu'il serait intéressant d'intégrer nos travaux dans des model-checker plus populaires tels que SPIN ou MEC qui ne sont pas à l'origine basés sur des ordres partiels.

Dans [CE12], Courcelle et Engelfriet ont montré que le problème de la vérification d'une formule MSO sur les graphes générés par une grammaire hors contexte est décidable. Ce résultat donne un autre point de vue de la vérification où les pomsets sont vu comme des graphes et les grammaires utilisés pour décrire des systèmes. Cette approche peut être utilisée pour la vérification de série-parallèle en particulier. Nous pensons qu'il soit intéressant de conjurer cette technique avec le notreaussi bien d'un point de vue théorique que d'un point de vue implémentation.

Enfin, nous pensons que nos travaux peuvent avoir des domaines d'application réels dans le milieu industriel ou des logiciels. Nous envisageons par exemple une application dans la vérification des compositions de services web.

Bibliographie

[AEY05] Rajeev Alur, Kousha Etessami, and Mihalis Yannakakis. Realizability and verification of msc-graphs, 2005.

[AHP96] Rajeev Alur, Gerard J. Holzmann, and Doron Peled. An analyzer for message sequence charts. In *SOFTWARE CONCEPTS AND TOOLS*, pages 304–313, 1996.

[Ake78] B. Akers. Binary decision diagrams. *IEEE Transactions on Computers*, 27(6):509–516, 1978.

[AMP98] Rajeev Alur, Kenneth L. McMillan, and Doron Peled. Deciding global partial-order properties. In Kim Guldstrand Larsen, Sven Skyum, and Glynn Winskel, editors, *ICALP*, volume 1443 of *Lecture Notes in Computer Science*, pages 41–52. Springer, 1998.

[AN82] A. Arnold and M. Nivat. Comportements de processus. In Colloque AFCET, pages 35–68, 1982.

[AP99] Rajeev Alur and Doron Peled. Undecidability of partial order logics. *Inf. Process. Lett.*, 69(3):137–143, February 1999.

[APP95] Rajeev Alur, Doron Peled, and Wojciech Penczek. Model-checking of causality properties. In *Proceedings of the 10th Annual IEEE Symposium on Logic in Computer Science*, LICS '95, pages 90–, Washington, DC, USA, 1995. IEEE Computer Society.

[Arn91] A. Arnold. An extension of the notions of traces and of asynchronous automata. *RAIRO - Theoretical Informatics and Applications - Informatique Théorique et Applications*, 25(4):355–393, 1991.

[BAL97] Hanene Ben-Abdallah and Stefan Leue. Syntactic detection of process divergence and non-local choice in message sequence charts. In *Proc. Intl. Conf. on Tools and Algorithms for the Construction and Analysis of Systems TACAS'97*, pages 259–274. Springer, 1997.

[Bas97] Twan Basten. Parsing partially ordered multisets. *International Journal of Foundations of Computer Science*, 08(04):379–407, 1997.

[Bed13] Nicolas Bedon. Logic and branching automata. In Krishnendu Chatterjee and Jirí Sgall, editors, *Mathematical Foundations of Computer Science 2013*, volume 8087 of *Lecture Notes in Computer Science*, pages 123–134. Springer Berlin Heidelberg, 2013.

[BG95] Serge Bauget and Paul Gastin. On congruences and partial orders. In Jirí Wiedermann and Petr Hájek, editors, *Proceedings of the 20th International Symposium on Mathematical Foundations of Computer Science (MFCS'95)*, volume 969 of *Lecture Notes in Computer Science*, pages 434–443, Prague, Czech Republic, August 1995. Springer.

[BHR84] S. D. Brookes, C. A. R. Hoare, and A. W. Roscoe. A theory of communicating sequential processes. *J. ACM*, 31(3):560–599, June 1984.

[BK91] B. Bloom and M. Kwiatkowska. Trade-offs in true concurrency: Pomsets and Mazurkiewicz traces. In *Proc. 7th International Conference on Mathematical Foundations of Programming Semantics (MFPS'91)*, volume 598 of *LNCS*, pages 350–375. Springer, 1991.

[BKSS06] Benedikt Bollig, Carsten Kern, Markus Schlütter, and Volker Stolz. Mscan: A tool for analyzing msc specifications. In *In TACAS 2006, volume 3920 of LNCS*, pages 455–458, 2006.

[BL69] J. Richard Büchi and Lawrence H. Landweber. Definability in the monadic second-order theory of successor. *J. Symb. Log.*, 34(2):166–170, 1969.

[Bry86] R. Bryant. Graph-based algorithms for boolean function manipulation. *IEEE Transactions on Computers*, 35(8):677–691, 1986.

[Büc62] Julius R. Büchi. On a decision method in restricted second order arithmetic. In Ernest Nagel, Patrick Suppes, and Alfred Tarski, editors, *Proceedings of the 1960 International Congress on Logic, Methodology and Philosophy of Science (LMPS'60)*, pages 1–11. Stanford University Press, June 1962.

[BZ83] Daniel Brand and Pitro Zafiropulo. On communicating finite-state machines. *J. ACM*, 30(2):323–342, April 1983.

[CE12] Professor Bruno Courcelle and Dr Joost Engelfriet. *Graph Structure and Monadic Second-Order Logic: A Language-Theoretic Approach*. Cambridge University Press, New York, NY, USA, 1st edition, 2012.

[CES83] Edmund M. Clarke, E. Allen Emerson, and A. Prasad Sistla. Automatic verification of finite state concurrent systems using temporal logic specifications: A practical approach. In *POPL*, pages 117–126, 1983.

[CG95] Christian Choffrut and Leucio Guerra. Logical definability of some rational trace languages. *Mathematical Systems Theory*, 28(5):397–420, 1995.

[Cou99] J.-M. Couvreur. On-the-fly verification of linear temporal logic. In *FM'99—Formal Methods, Volume I*, volume 1708 of *LNCS*, pages 253–271. Springer Verlag, 1999.

[DG01] Volker Diekert and Paul Gastin. Local temporal logic is expressively complete for cograph dependence alphabets. In Robert Nieuwenhuis and Andrei Voronkov, editors, *Logic for Programming, Artificial Intelligence, and Reasoning*, volume 2250 of *Lecture Notes in Computer Science*, pages 55–69. Springer Berlin Heidelberg, 2001.

[DG02] Volker Diekert and Paul Gastin. {LTL} is expressively complete for mazurkiewicz traces. *Journal of Computer and System Sciences*, 64(2):396 – 418, 2002.

[DG06] Volker Diekert and Paul Gastin. Pure future local temporal logics are expressively complete for mazurkiewicz traces. *Information and Computation*, 204(11):1597 – 1619, 2006.

[DGK00] Manfred Droste, Paul Gastin, and Dietrich Kuske. Asynchronous cellular automata for pomsets. *Theoretical Computer Science*, 247(1-2):1–38, September 2000.

[Die95] Volker Diekert. *The Book of Traces*. World Scientific Publishing Co., Inc., River Edge, NJ, USA, 1995.

[DS02] Stéphane Demri and Philippe Schnoebelen. The complexity of propositional linear temporal logics in simple cases. *Information and Computation*, 174(1):84 – 103, 2002.

[Eil74] S. Eilenberg. *Automata, languages, and machines*. Pure and Applied Mathematics. Elsevier Science, 1974.

[Elg61] Calvin C. Elgot. Decision problems of finite automata design and related arithmetics. pages 21–51, 1961.

[EM93] Werner Ebinger and Anca Muscholl. Logical definability on infinite traces. In Andrzej Lingas, Rolf Karlsson, and Svante Carlsson, editors, *Automata, Languages and Programming*, volume 700 of *Lecture Notes in Computer Science*, pages 335–346. Springer Berlin Heidelberg, 1993.

[Gis85] Jay Loren Gischer. *Partial orders and the axiomatic theory of shuffle (pomsets)*. PhD thesis, Stanford, CA, USA, 1985. AAI8506191.

[Gis88] Jay L. Gischer. The equational theory of pomsets. *Theor. Comput. Sci.*, 61(2-3):199–224, November 1988.

[GK03] Paul Gastin and Dietrich Kuske. Satisfiability and model checking for mso-definable temporal logics are in pspace. In Roberto Amadio and Denis Lugiez, editors, *CONCUR 2003 - Concurrency Theory*, volume 2761 of *Lecture Notes in Computer Science*, pages 222–236. Springer Berlin Heidelberg, 2003.

[GK05] Paul Gastin and Dietrich Kuske. Uniform satisfiability problem for local temporal logics over mazurkiewicz traces. In *CONCUR*, pages 533–547, 2005.

[GK10] Paul Gastin and Dietrich Kuske. Uniform satisfiability problem for local temporal logics over mazurkiewicz traces. *Inf. Comput.*, 208(7):797–816, July 2010.

[GKM06] Blaise Genest, Dietrich Kuske, and Anca Muscholl. A kleene theorem and model checking algorithms for existentially bounded communicating automata. *Information and Computation*, 204(6):920 – 956, 2006.

[GMSZ02] Blaise Genest, Anca Muscholl, Helmut Seidl, and Marc Zeitoun. Infinite-state high-level mscs: Model-checking and realizability. In Peter Widmayer, Francisco Triguero Ruiz, Rafael Morales Bueno, Matthew Hennessy, Stephan Eidenbenz, and Ricardo Conejo, editors, *Automata, Languages and Programming*, volume 2380 of *Lecture Notes in Computer Science*, pages 657–668. Springer, 2002.

[GPVW95] R. Gerth, D. Peled, M. Y. Vardi, and P. Wolper. Simple on-the-fly automatic verification of linear temporal logic. In *PSTV'95*. North-Holland, 1995.

[Gra81] J. Grabowski. On partial languages. *Fundam. Inform.*, 4(2):427–, 1981.

[HJJ+95] J.G. Henriksen, J. Jensen, M. Jørgensen, N. Klarlund, B. Paige, T. Rauhe, and A. Sandholm. Mona: Monadic second-order logic in practice. In *Tools and Algorithms for the Construction and Analysis of Systems, First International Workshop, TACAS '95, LNCS 1019*, 1995.

[HMK+05] Jesper G. Henriksen, Madhavan Mukund, K. Narayan Kumar, Milind A. Sohoni, and P. S. Thiagarajan. A theory of regular msc languages. *Information and Computation/information and Control*, 202:1–38, 2005.

[HMKT00] JesperG. Henriksen, Madhavan Mukund, K.Narayan Kumar, and P.S. Thiagarajan. Regular collections of message sequence charts. In Mogens Nielsen and Branislav Rovan, editors, *Mathematical Foundations of Computer Science 2000*, volume 1893 of *Lecture Notes in Computer Science*, pages 405–414. Springer Berlin Heidelberg, 2000.

[Hol97] Gerard J. Holzmann. The model checker spin. *IEEE Trans. Software Eng.*, 23(5):279–295, 1997.

[Hol03] G. J. Holzmann. *The SPIN MODEL CHECKER*. Addison-Wesley, 2003.

[Hoo94] P. W. Hoogers. *Behavioural aspects of Petri nets*. PhD thesis, Leiden, Rijksuniversiteit, 1994. Th. : Ph. D.

[INB99] L. Ivanov, R. Nunna, and S. Bloom. Modeling and analysis of noniterated systems: an approach based upon series-parallel posets. In *International Symposium on Circuits and Systems (ISCAS 1999), May 30 - June 2, 1999, Orlando, Florida, USA*, pages 404–406, 1999.

[ITU99] ITU. *Recommendation Z.120: Message Sequence Chart (MSC)*. Haugen (ed.), Geneva, 1999.

[Kla98] Nils Klarlund. Mona & fido: The logic-automaton connection in practice. In Mogens Nielsen and Wolfgang Thomas, editors, *Computer Science Logic*, volume 1414 of *Lecture Notes in Computer Science*, pages 311–326. Springer Berlin Heidelberg, 1998.

[Kle56] S. C. Kleene. Representation of events in nerve nets and finite automata. In Claude Shannon and John McCarthy, editors, *Automata Studies*, pages 3–41. Princeton University Press, Princeton, NJ, 1956.

[KM01] Nils Klarlund and Anders Møller. *MONA Version 1.4 User Manual*. BRICS, Department of Computer Science, Aarhus University, January 2001. Notes Series NS-01-1. Available from `http://www.brics.dk/mona/`. Revision of BRICS NS-98-3.

[Kus00a] D. Kuske. *Contributions to a Trace Theory Beyond Mazurkiewicz Traces*. 2000.

[Kus00b] Dietrich Kuske. Infinite series-parallel posets: Logic and languages. In *ICALP*, volume 1853 of *Lecture Notes in Computer Science*, pages 648–662. Springer, 2000.

[Kus01] Dietrich Kuske. A model theoretic proof of büchi-type theorems and first-order logic for n-free pomsets. In *Proceedings of the 18th Annual Symposium on Theoretical Aspects of Computer Science*, STACS '01, pages 443–454, London, UK, UK, 2001. Springer-Verlag.

[Kus02] Dietrich Kuske. Recognizable sets of n-free pomsets are monadically axiomatizable. In Werner Kuich, Grzegorz Rozenberg, and Arto Salomaa, editors, *Developments in Language Theory*, volume 2295 of *Lecture Notes in Computer Science*, pages 206–216. Springer Berlin Heidelberg, 2002.

[LL95] Peter B. Ladkin and Stefan Leue. Interpreting message flow graphs. *Formal Aspects of Computing*, 7, 1995.

[LW98] K. Lodaya and P. Weil. Series-parallel posets: Algebra, automata and languages. In *STACS98, Lecture Nodes in Computer Science*, pages 555–565. Springer, 1998.

[LW00] Kamal Lodaya and Pascal Weil. Series-parallel languages and the bounded-width property. *Theor. Comput. Sci.*, 237(1-2):347–380, 2000.

[Maz77] Antoni Mazurkiewicz. Concurrent program schemes and their interpretations. DAIMI PB 78, 1977.

[MKS00] Madhavan Mukund, K.Narayan Kumar, and Milind Sohoni. Synthesizing distributed finite-state systems from mscs. In Catuscia Palamidessi, editor, *CONCUR 2000 - Concurrency Theory*, volume 1877 of *Lecture Notes in Computer Science*, pages 521–535. Springer Berlin Heidelberg, 2000.

[Mor01] Rémi Morin. On regular message sequence chart languages and relationships to mazurkiewicz trace theory. In Furio Honsell and Marino Miculan, editors, *Foundations of Software Science and Computation Structures*, volume 2030 of *Lecture Notes in Computer Science*, pages 332–346. Springer Berlin Heidelberg, 2001.

[Mor02] Remi Morin. Recognizable sets of message sequence charts. In *STACS 2002, LNCS 2030*, pages 523–534. Springer, 2002.

[MP99] Anca Muscholl and Doron Peled. Message sequence graphs and decision problems on mazurkiewicz traces. In *In Proc. of MFCS'99, LNCS 1672*, pages 81–91. Springer, 1999.

[MPS98] Anca Muscholl, Doron Peled, and Zhendong Su. Deciding properties for message sequence charts, 1998.

[MR94] S. Mauw and M.A. Reniers. An algebraic semantics of basic message sequence charts. *The Computer Journal*, 37:269–277, 1994.

[Mus99] Anca Muscholl. Matching specifications for message sequence charts. In *Proc. FOSSACS '99, LNCS 1578, Springer-Verlag*, pages 273–287, 1999.

[Och85] Edward Ochmanski. Regular behaviour of concurrent systems. *Bulletin of the EATCS*, 27:56–67, 1985.

[OMG03] OMG. *Unified Modeling Language Specification (version 2.0)*, 2003. OMG Final Adopted Specification doc. ptc/03-09-01.

[Pel00] Doron Peled. Specification and verification of message sequence charts. In *Proceedings of the FIP TC6 WG6.1 Joint International Conference on Formal Description Techniques for Distributed Systems and Communication Protocols (FORTE XIII) and Protocol Specification, Testing and Verification (PSTV XX)*, FORTE/PSTV 2000, pages 139–154, Deventer, The Netherlands, The Netherlands, 2000. Kluwer, B.V.

[Pen92] Wojciech Penczek. On undecidability of propositional temporal logics on trace systems. *Inf. Process. Lett.*, 43(3):147–153, September 1992.

[Pet62] Carl Adam Petri. *Kommunikation mit Automaten.* PhD thesis, 1962.

[Pnu77] Amir Pnueli. The temporal logic of programs. In *Foundations of Computer Science, 1977., 18th Annual Symposium on*, pages 46–57, 1977.

[Pos46] Emil L. Post. A variant of a recursively unsolvable problem. *Bulletin of the American Mathematical Society*, 52(4):264–268, 04 1946.

[PP04] D. Perrin and J.E. Pin. *Infinite Words: Automata, Semigroups, Logic and Games*. Number v. 141 in Infinite words: automata, semigroups, logic and games. Elsevier, 2004.

[Pra86] Vaughan R. Pratt. Modelling concurrency with partial orders. *International Journal of Parallel Programming*, 15(1):33–71, 1986.

[Ren98] Door Michel Adriaan Reniers. Message sequence chart: Syntax and semantics. Technical report, Faculty of Mathematics and Computing, 1998.

[Sak03] Jacques Sakarovitch. *Eléments de théorie des automates*. Les classiques de l'informatique. Vuibert, Paris, 2003.

[SC85] A. P. Sistla and E. M. Clarke. The complexity of propositional linear temporal logics. *J. ACM*, 32(3):733–749, July 1985.

[Sto74] Larry J. Stockmeyer. *The Complexity of Decision Problems in Automata Theory and Logic*. PhD thesis, MIT, Cambridge, Massasuchets, USA, 1974.

[Tau03] H. Tauriainen. *On Translating Linear Temporal Logic into Alternating and Nondeterministic Automata*. Research reports 83, Helsinki University of Technology, Laboratory for Theoretical Computer Science, Espoo, Finland, 2003.

[Tho90] W. Thomas. Automata on infinite objects. In J. van Leeuwen, editor, *Handbook of Theoretical Computer Science: Volume B: Formal Models and Semantics*, pages 133–191. Elsevier, Amsterdam, 1990.

[Tho96] Wolfgang Thomas. Languages, automata, and logic. In *Handbook of Formal Languages*, pages 389–455. Springer, 1996.

[TW02] P.S. Thiagarajan and I. Walukiewicz. An expressively complete linear time temporal logic for mazurkiewicz traces. *Information and Computation*, 179(2):230 – 249, 2002.

[vN51] John von Neumann. *The general and logical theory of automata*, pages 1–41. Wiley, Pasadena CA, 1951.

[VTL79] Jacobo Valdes, Robert E. Tarjan, and Eugene L. Lawler. The recognition of series parallel digraphs. In *Proceedings of the Eleventh Annual ACM Symposium on Theory of Computing*, STOC '79, pages 1–12, 1979.

[Wal98] Igor Walukiewicz. Difficult configurations - on the complexity of ltrl. In KimG. Larsen, Sven Skyum, and Glynn Winskel, editors, *Automata, Languages and Programming*, volume 1443 of *Lecture Notes in Computer Science*, pages 140–151. Springer Berlin Heidelberg, 1998.

[Wol00] P. Wolper. Constructing Automata from Temporal Logoc Formulas : A tutorial. In *FMPA 2000*, volume 2090 of *LNCS*, pages 261–277. Springer Verlag, 2000.

Mouhamadou Tafsir SAKHO

Γ-pomset pour la modélisation et la vérification de systèmes parallèles

Résumé

Un comportement distribué peut être décrit avec un multi-ensemble partiellement ordonné (pomset). Bien que compacts et très intuitifs, ces modèles sont difficiles à vérifier. La principale technique utilisée dans cette thèse est de ramener les problèmes de décision de la logique MSO sur les pomsets à des problèmes de décision sur les mots. Les problèmes considérés sont la satisfiabilité et la vérification. Le problème de la vérification pour une formule donnée et un pomset consiste à décider si une interprétation est vraie, et le problème de satisfiabilité consiste à décider si un pomset répondant à la formule existe. Le problème de satisfiabilité de MSO sur pomsets est indécidable. Une procédure de semi-décision peut apporter des solutions pour de nombreux cas, en dépit du fait qu'elle peut ne pas terminer. Nous proposons un nouveau modèle, que l'on appelle Γ-Pomset , pouvant rendre l'exploration des pomsets possible. Par conséquent, si une formule est satisfiable alors notre approche mènera éventuellement à la détection d'une solution. De plus, en utilisant les Γ-Pomsets comme modèles pour systèmes concurrents, le model-checking de formules ordre partiel sur systèmes concurrents est décidable. Certaines expérimentations ont été faites en utilisant l'outil MONA. Nous avons comparé aussi la puissance expressive de certains modèles classiques de la concurrence comme les traces de Mazurkiewicz avec les Γ-Pomsets .
Mots clés: Pomset, Model-checking, Satisfiabilité, traces.

Γ-pomset for modelling and verifying parallel systems

Abstract

Multiset of partially ordered events (pomset) can describe distributed behavior. Although very intuitive and compact, these models are difficult to verify. The main technique used in this thesis is to bring back decision problems for MSO over pomsets to problems for MSO over words. The problems considered are satisfiability and verification. The verification problem for a formula and a given pomset consists in deciding whether such an interpretation exists, and the satisfiability problem consists in deciding whether a pomset satisfying the formula exists. The satisfiability problem of MSO over pomsets is undecidable. A semi-decision procedures can provide solutions for many cases despite the fact that they may not terminate. We propose a new model, so called Γ-Pomset , making the exploration of pomsets space possible. Consequently, if a formula is satisfiable then our approach will eventually lead to the detection of a solution. Moreover, using Γ-Pomsets as models for concurrent systems, the model checking of partial order formulas on concurrent systems is decidable. Some experiments have been made using MONA. We compare also the expressive power of some classical model of concurrency such as Mazurkiewicz traces with our Γ-Pomset .
Keywords: Pomset, Satisfiability, Trace, Model checking.