Accessibilité dans les systèmes à compteurs continus: théorie et pratique

Présenté dans le cadre du cours
IFT4055 - Projet informatique honor
par Vincent Antaki

2 mai 2015

Résumé

Le problème d'accessibilité dans les *Systèmes d'addition de vecteurs avec états* (VASS), un modèle équivalent aux réseaux de Petri, est décidable mais aucune borne non primitive récursive n'est connue sur le temps requis en pire cas pour résoudre ce problème. Dans ce projet, nous étudions le problème d'accessibilité dans les systèmes à compteurs continus, nous définissons deux modèles de VASS continu et nous prouvons l'équivalence d'un de ces modèles avec les réseaux de Petri continus (CPN). Par la suite, nous présentons une implémentation de l'algorithme de Fraca et Haddad qui permet de résoudre le problème d'accessibilité dans les CPN.

1 Introduction

Les comportements des systèmes informatiques peuvent être représentés, en partie ou en totalité, par des modèles mathématiques tels que les automates finis, les grammaires hors-contextes, les nombreux systèmes de logique formelle, les machines à compteurs ou encore les machines de Turing. C'est en appliquant un processus de vérification formelle sur le modèle correspondant que nous pouvons nous assurer du comportement correct du système. Le processus peut viser à répondre à différentes questions : Le système peut-il atteindre une configuration particulière ? Peut-on affirmer qu'étant donné une configuration initiale, certains états ou certaines configurations lui sont inaccessibles ? En combien d'étapes ce système arrive-t-il au résultat souhaité ?

La difficulté à répondre à de telles questions varie selon le modèle étudié. Dans le cadre de ce projet, nous nous penchons sur un type particulier de machines à compteurs, soit les systèmes d'addition de vecteurs avec états, abrégés VASS de l'anglais «Vector addition systems with states». Les VASS sont des automates avec des compteurs où, lorsque activée, une transition du système retire ou ajoute un nombre entier de jetons dans chaque compteurs. Les règles

de transition d'un VASS dépendent du contenu de ces compteurs car un compteur ne peut jamais contenir un nombre négatif de jetons, ce qui contraint les transitions pouvant être activées. Ce système à compteurs est un outil de modélisation équivalent aux réseaux de Petri. Un des problèmes le plus central dans l'étude de ces modèles est le problème d'accessibilité. Celui-ci peut se formuler simplement par la question suivante : Étant donné un système, une configuration initiale et une configuration finale, existe-t-il un chemin dans le système débutant de la configuration initiale et se terminant à la configuration finale ? Bien qu'étudié depuis plus de trente ans, le problème de l'accessibilité dans les réseaux de Petri pose un défi particulier puisque sa décidabilité est connue depuis 1981 mais qu'aucune borne supérieure primitive récursive n'est connue sur sa complexité.

En 1979, Hopcroft et Pansiot [HP79] introduisent les VASS et démontrent leur équivalence aux VAS introduits une dizaine d'années plus tôt par Karp et Miller [KM69] pour modéliser les calculs parallèles. L'équivalence entre les VAS et les réseaux de Petri (PN) introduits par Carl Adam Petri [Pet62] étant déjà connue à l'époque, il est possible de conclure à l'équivalence entre les VASS et les PN.

La décidabilité du problème d'accessibilité pour les VASS a été prouvée et rafinée à travers une technique de décomposition [May81, Kos82, Lam92, Ler12] mais aucune borne n'était connue jusqu'à tout récemment. Dernièrement, une borne non primitive récursive à été prouvée pour la technique de décomposition [LS15], devenant donc la première borne connue sur le temps en pire cas pour le problème d'accessiblité pour les VAS, VASS et PN.

De multiples travaux traitent toutefois de la complexité du problème d'accessibilité dans des modèles moins expressifs, obtenus par exemple en limitant le nombre de compteurs ou en permettant la fluidification (franchissement fractionnaire plutôt qu'entier) des transitions du système. Ces modèles et leurs algorithmes correspondants pour résoudre le problème d'accessibilité sont une source d'informations lors de la résolution du problème d'accessibilité dans les modèles de base. L'exemple-type d'information que nous apporte un modèle relaxé est, lorsqu'on conclut la non-accessibilité dans le modèle relaxé, la non-accessibilité dans le modèle de base. Le problème d'accessibilité est NP-complet pour les VASS à un seul compteur [HKOW09] ainsi que pour les VASS sans gardes avec l'opération de remise à zéro [HH14] et PSPACE-complet pour les VASS à deux compteurs [BFG+14]. Il est P-complet pour les réseaux de Petri continus [FH13].

Il est aussi possible de tirer de l'information sur l'accessibilité en résolvant des problèmes connexes. Le problème de couverture, qui consiste à déterminer s'il est possible d'atteindre une configuration plus grande ou égale à une configuration donnée, en est un bon exemple. Encore une fois, s'il n'existe pas de solution à un problème de couverture, nous pouvons conclure à l'absence

de solution pour le problème d'accessibilité pour les mêmes configurations. La complexité du problème de couverture est dans P pour les VASS à un compteur, NP-difficile pour les VASS à deux compteurs [RY86] et dans EXPSPACE pour les VASS avec trois compteurs ou plus [RAC78].

1.1 Contribution

La composante principale du projet vise l'implantation d'algorithmes développés pour le traitement de relaxations du problème d'accessibilité dans les VASS. Le but ultime en aval de ce projet est de fournir un outil qui saura résoudre partiellement le problème d'accessibilité, en identifiant d'abord les cas simples, puis en tentant de prouver la non accessibilité en traitant itérativement des relaxations du système, puis en recourant à des heuristiques, avant de recourir à la méthode exhaustive (qui est très lente).

Une première contribution à ce projet est théorique. Les VASS continus (CVASS) n'ont pas été définis préalablement à ce projet. Puisqu'il existe une équivalence entre les VASS et les réseaux de Petri, il est donc intéressant de vérifier si une telle équivalence existe dans le cas continu. Dans la section 3 de ce document, nous suggérons deux modèles possibles pour les CVASS. Ces deux modèles peuvent emprunter leur transition de manière fractionnaire et donc accéder à des configurations inaccessibles pour un VASS. Le premier modèle possède une *fluidité* au niveau de ses états, i.e. ses états changent de manière fractionnaire selon la quantité d'une transition empruntée. Nous expliciterons l'équivalence de ce modèle avec les CPN. Dans le deuxième modèle, le système demeure à tout moment dans un seul état. Certaines questions ouvertes quant au deuxième modèle seront discutées dans la conclusion car celui-ci est une nouvelle relaxation du problème.

La seconde contribution de ce projet est pratique. Elle consiste en l'implémentation de l'algorithme de résolution du problème d'accessibilité des CPN, qui fonctionne en temps polynomial, de Fraca et Haddad [FH13]. Diverses explications relatives à l'implémentation de l'algorithme et aux difficultés surmontées seront fournies dans la section 4 du document.

1.2 Organisation du document

Ce document est divisé en cinq sections. La deuxième section présente les définitions des machines à compteurs et des problèmes d'accessibilité qui seront utilisés dans le reste du document. La troisième section introduit les deux nouveaux modèles de VASS continus et des procédures de conversion entre le premier modèle et les CPN. La section 4 discute des détails d'implémentation de l'algorithme de résolution du problème d'accessibilité pour les CPN. La section 5 présente de brèves conclusions sur le rapport et des perspectives pour des futurs projets.

2 Accessibilité dans les machines à compteurs

Dans cette section, nous définissons les modèles de machines à compteurs qui seront utilisés dans le document. Les définitions des VASS ont été reprises de [Blo14]. La définition des CPN a été calquée sur celle de [FH13].

2.1 Notation

Nous introduisons la notation utilisée dans ce document. Certaines seront introduites plus loin lorsque approprié.

- Les opérateurs de comparaisons sont étendus aux vecteurs. Plus formellement, soit $\star \in \{<, \leq, >, \geq, =\}$, alors $\vec{u} \star \vec{v} \iff \forall i\ \vec{u}[i] \star \vec{v}[i]$.
- Les opérateurs min et max sont étendus aux vecteurs. Soit $\star \in \{min, max\}$, alors $\vec{x} = \star(\vec{u}, \vec{v}) \iff \forall i\ \vec{x}[i] = \star(\vec{u}[i], \vec{v}[i])$
- L'exposant s'applique aussi aux ensembles. Soit Q un ensemble, $Q^i = \times_{j=1}^{i} Q$.
- L'opérateur * appliqué à un ensemble Q représente l'ensemble des mots de tailles finis qu'il est possible de faire avec l'ensemble Q. Autrement dit, $Q^* = \cup_{j=0}^{\infty} Q^j$.
- Soit P, une matrice de dimension 2. $P[a, b]$ fait référence à l'élément situé à la ligne a et la colonne b, $P[a]$ fait référence au vecteur de la ligne a et $P[, b]$ fait référence au vecteur de la colonne b.
- Sur les graphes de réseaux de Petri présents dans ce document, une absence d'étiquette sur un arc représente l'étiquette "1".
- $a \in (0, 1]$ signifie que la variable a est élément de l'ensemble continu de 0 exclu à 1 inclus, autrement dit, $0 < a \leq 1$.

2.2 Systèmes d'addition de vecteurs avec états (VASS)

Définition 2.1. Soit $d \in \mathbb{N}$. Un *système d'addition de vecteurs avec états de dimension d (d-VASS)* V est une paire $V = (Q, \delta)$ où

- Q est un ensemble fini (états),
- $\delta \subseteq Q \times \mathbb{Z}^d \times Q$ est un ensemble fini (transitions),

Pour tout $d \in \mathbb{N}$, nous appelons un d-VASS simplement un *système d'addition de vecteurs avec états (VASS)*.

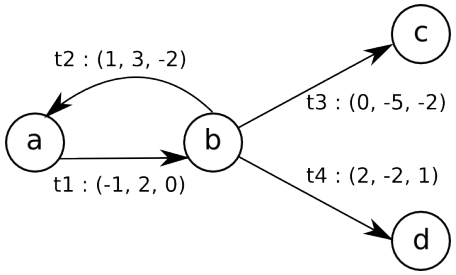
Il est possible de représenter un d-VASS $V = (Q, \delta)$ par un graphe orienté $G(V)$. Ses sommets correspondent à chaque état de Q et $G(V)$ contient une arête (p, q) étiquetée par $v \in \mathbb{Z}^d$ si $(p, v, q) \in \delta$.

Exemple 2.2. *La figure 2.3 est une représentation graphique du 3-VASS sui-*

vant :

$$\begin{aligned} V =& (Q, \delta) \\ Q =& \{a, b, c, d\} \\ \delta =& \{t_1, t_2, t_3, t_4\} \\ =& \{(a, (-1, 2, 0), b), (b, (1, 3, -2), a), \\ & (b, (0, -5, -2), c), (b, (2, -2, 1), d)\} \end{aligned}$$

Figure 2.3. *Illustration d'un 3-VASS.*



Définition 2.4. Soit $V = (Q, \delta)$ un d-VASS. Une *configuration* est une paire $(q, x) \in Q \times \mathbb{N}^d$.

Définition 2.5. Une *exécution* est une suite de configurations $p = q_1 x_1 q_2 x_2 ... q_k x_k$ où $q_i \in Q$ et $x_i \in \mathbb{N}^d$ telles que $(q_i, (x_{i+1} - x_i), q_{i+1}) \in \delta$ pour tout $i \in \{1, 2, ..., k-1\}$. Notons qu'une exécution s'assure que les transitions activées n'engendrent pas de valeurs négatives dans les compteurs. Une exécution ρ est *étiquetée* par $(q_1, (x_2 - x_1), q_2)...(q_{k-1}, (x_k - x_{k-1}), q_k) \in \delta^*$.

Définition 2.6. Une *séquence d'activation d'un VASS* est une liste de transitions $t_1, t_2, \ldots$ telles que $t_i \in \delta$ pour tout i.

Pour être valide à partir d'une configuration initiale, l'exécution d'une séquence d'activation ne doit en aucun moment faire tomber un compteur dans les négatifs.

Définition 2.7. Soit $V = (Q, \delta)$ un d-VASS. Soient $(q, u), (p, v) \in Q \times \mathbb{N}^d$ deux configurations. Nous écrivons $(q, u) \xrightarrow{w} (p, v)$ s'il existe une exécution de (q, u) à (p, v) étiquetée par w. Nous écrivons $(q, u) \xrightarrow{*} (p, v)$ s'il existe une exécution de (q, u) à (p, v).

Exemple 2.8. *Soit le 3-VASS illustré à la figure 2.3, la configuration initiale* $(q_0, x_0) = (a, (2, 0, 4))$ *et la séquence d'activation* σ.

$$\begin{aligned} \sigma &= t_1 t_2 t_1 t_3 \\ &= (a, (-1, 2, 0), b)(b, (1, 3, -2), a)(a, (-1, 2, 0), b)(b, (0, -5, -2), c) \end{aligned}$$

L'exécution ρ *débutant à la configuration initiale et suivant* σ *est valide.*

$$\begin{aligned} \rho &= (q_0, x_0)(q_1, x_1)(q_2, x_2)(q_3, x_3)(q_4, x_4) \\ &= (a, (2, 0, 4))(b, (1, 2, 4))(a, (2, 5, 2))(b, (1, 7, 2))(c, (1, 2, 0)) \end{aligned}$$

Exemple 2.9. *Soit le même 3-VASS et la même séquence d'activation qu'à l'exemple 2.8. Soit la configuration initiale* $(q_0, x_0) = (a, (2, 0, 2))$. *Alors,* ρ' *n'est pas une exécution car l'activation de la transition* t_3 *fait tomber le 3e compteur sous zéro :*

$$\begin{aligned}\rho' &= (q_0, x_0)(q_1, x_1)(q_2, x_2)(q_3, x_3)(q_4, x_4) \\ &= (a, (2, 0, 2))(b, (1, 2, 2))(a, (2, 5, 0))(b, (1, 7, 0))(c, (1, 2, -2))\end{aligned}$$

2.2.1 Réseaux de Petri

Définition 2.10. Un *réseau de Petri* P est un tuple $P = (P, T, Pre, Post)$ où
- P est un ensemble fini de places (compteurs),
- T est un ensemble fini de transitions,
- Pre est une matrice de taille $|P| \times |T|$ indiquant, pour une colonne i, le coût pour activer la transition i,
- $Post$ est une matrice de taille $|P| \times |T|$ indiquant, pour une colonne i, l'ajout dans les compteurs suite à l'activation de la transition i,

Définition 2.11. Un *marquage* de PN est un vecteur $m \in \mathbb{N}^P$ où $m[q]$ est appelée le nombre de *jetons*.

Définition 2.12. Soit $PN = (P, T, Pre, Post)$ un réseau de Petri. Une *exécution* est un mot $w = m_0 m_1 m_2 ... m_k \in (\mathbb{N}^P)^*$ de configurations telles que

$$\begin{aligned}&\forall\ i \in \{0, ..., k-1\} \\ \exists j \in\ T\ &: m_i - Pre[, j] \geq \vec{0} \\ &\wedge\ m_{i+1} = m_i - Pre[, j] + Post[, j]\end{aligned}$$

Lors de l'exécution d'un réseau de Petri, une transition ne peut être activée si elle demande pour une place p un plus grand coût qu'il n'y a de jetons dans le marquage courant.

Définition 2.13. La *matrice d'incidence* C d'un PN est définie comme la différence entre la matrice Post et la matrice Pre. Elle représente l'impact des transitions sur les compteurs du système.

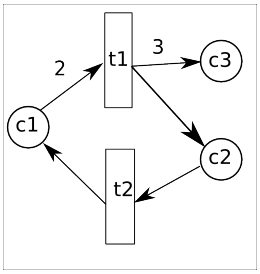
$$C \stackrel{def}{=} Post - Pre$$

Définition 2.14. Soit un réseau de Petri $PN = (P, T, Pre, Post)$, son réseau de Petri *renversé* est défini comme $PN^{-1} = (P, T, Post, Pre)$. Dans le réseau renversé, les places et les transitions coïncident mais le sens des arcs est inversé.

Exemple 2.15. *Soit le réseau de Petri N défini par les paramètres suivants :*

$$P = \{c_1, c_2, c_3\}$$
$$T = \{t_1, t_2\}$$
$$Pre = \begin{pmatrix} 2 & 0 \\ 0 & 1 \\ 0 & 0 \end{pmatrix}$$
$$Post = \begin{pmatrix} 0 & 1 \\ 1 & 0 \\ 3 & 0 \end{pmatrix}$$

Figure 2.16. *Le graphe du réseau de Petri de l'exemple 2.15.*



Définition 2.17. Étant donné une place $p \in P$ (respectivement une transition $t \in T$), le *preset*, ${}^\bullet p$ (resp. ${}^\bullet t$), est défini comme l'ensemble des transitions (resp. places) tel que :

$${}^\bullet p \stackrel{def}{=} \{t \in T \mid Post[p,t] > 0\}$$

$$(\text{resp. } {}^\bullet t \stackrel{def}{=} \{p \in P \mid Pre[p,t] > 0\})$$

Définition 2.18. Étant donné une place $p \in P$ (respectivement une transition $t \in T$), le *postset*, $p^\bullet$ (resp. $t^\bullet$), est défini comme l'ensemble des transitions (resp. places) tel que :

$$p^\bullet \stackrel{def}{=} \{t \in T \mid Pre[p,t] > 0\}$$

$$(\text{resp. } t^\bullet \stackrel{def}{=} \{p \in P \mid Post[p,t] > 0\})$$

Exemple 2.19. *Dans le réseau de Petri N de l'exemple 2.15 :*

$${}^\bullet t1 = \{c_1\}$$
$$t1^\bullet = \{c_2, c_3\}$$
$${}^\bullet c_3 = \{t_1\}$$
$$c_3^\bullet = \emptyset$$

2.3 Systèmes continus

2.3.1 Réseaux de Petri continus

Définition 2.20. Un *réseau de Petri continu* (CPN) $C = (P, T, Pre, Post)$ est défini de la même façon qu'un réseau de Petri. Par contre, son comportement diffère.

Dans un réseau de Petri continu, il est permis d'activer une transition selon une fraction $\alpha \in (0, 1]$. La quantité de jetons nécessaire ainsi que la quantité de jetons résultante de la transition sont ajustées selon le facteur α.

Définition 2.21. Un *marquage* de CPN est un vecteur $m \in \mathbb{R}_{\geq 0}^{P}$ où $m[q]$ est appelé le nombre de *jetons* même s'il peut avoir une valeur non entière.

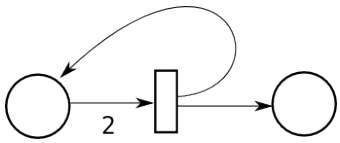
Définition 2.22. Une *séquence d'activation d'un CPN* est une liste de paires α_i, t_i où $\forall i\ t_i \in T, \alpha_i \in (0, 1]$.

Pour être valide à partir d'un marquage m_0, l'exécution d'une séquence d'activation ne doit en aucun moment faire tomber un compteur dans les négatifs.

Il est à noter que les CPN, à la différence de leur équivalent discret, peuvent avoir une exécution infinie qui converge. On parlera dans ce cas de *lim*-accessibilité.

Exemple 2.23. *La figure 2.24 est un bon exemple de la lim-accessibilité dans les CPN. Considérons le marquage initial $m_0 = (2, 0)$ et un marquage final $m = (0, 2)$ où le premier élément d'un marquage est le compteur de gauche et le deuxième le compteur de droite. Le marquage final $(0, 2)$ est accessible à partir d'une séquence d'activation infinie $\sigma = \alpha_0 t \alpha_1 t \alpha_2 t ...$ où $\alpha_0 = 1$ et $\alpha_{i+1} = \frac{\alpha_i}{2}$.*

Figure 2.24. *Le CPN suivant possède une séquence d'activation infinie telle que $(2, 0) \xrightarrow{*} (0, 2)$*



2.4 Accessibilité

L'ensemble d'accessibilité d'une paire (S, x_0) où S est un système et x_0 une configuration initiale est l'ensemble des configurations pouvant être atteintes par une exécution dans S débutant à x_0. Les problèmes qui se rapportent à l'accessibilité sont des questions de caractérisation de cet ensemble.

2.4.1 Problème d'accessibilité

Le problème d'accessibilité est central dans l'étude des machines à compteurs. Celui-ci pose la question à savoir s'il existe une exécution permettant à

un système S d'aller d'une configuration initiale x_0 jusqu'à une configuration finale x. En d'autre mots, est-ce que x est un élément de l'ensemble d'accessibilité de (S, x_0) ?

Accès$_d$(VASS)
— Entrée : Un d-VASS $V = (Q, \delta)$, une configuration initiale $(p, v) \in Q \times \mathbb{N}^d$ et une configuration finale $(q, u) \in Q \times \mathbb{N}^d$.
— Question : $\exists \sigma \in \delta^*$ tel que $(p, v) \xrightarrow{\sigma} (q, u)$?
Accès(PN)
— Entrée : Un réseau de Petri $N = (P, T, Pre, Post)$, un marquage initial $m_0 \in \mathbb{N}^{|P|}$ et un marquage final $m \in \mathbb{N}^{|P|}$.
— Question : $\exists \sigma \in T^*$ tel que $m_0 \xrightarrow{\sigma} m$?
Accès(CPN)
— Entrée : Un réseau de Petri $N = (P, T, Pre, Post)$, un marquage initial $m_0 \in \mathbb{N}^{|P|}$ et un marquage final $m \in \mathbb{N}^{|P|}$.
— Question : $\exists \sigma \in ((0, 1] \times T)^*$ tel que $m_0 \xrightarrow{\sigma} m$?

2.4.2 Problème de couverture

Le problème de couverture est un problème moins contraignant que le problème d'accessibilité défini précédemment. Les paramètres en entrée sont un système, une configuration initiale et une borne inférieure x sur la configuration finale. Le problème formulé est le suivant : Existe-t-il une exécution dans le système partant de la configuration initiale et se rendant à une configuration finale plus grande ou égale à x.

Couverture$_d$(VASS)
— Entrée : Un d-VASS $V = (Q, \delta)$, une configuration initiale $(p, v) \in Q \times \mathbb{N}^d$ et une configuration $(q, u) \in Q \times \mathbb{N}^d$.
— Question : $\exists \sigma \in \delta^*$ et $y \geq u$ tel que $(p, v) \xrightarrow{\sigma} (q, y)$?
Couverture(PN)
— Entrée : Un réseau de Petri $N = (P, T, Pre, Post)$, un marquage initial $m_0 \in \mathbb{N}^{|P|}$ et un marquage $m \in \mathbb{N}^{|P|}$.
— Question : $\exists \sigma \in T^*$ et $y \geq m$ tel que $m_0 \xrightarrow{\sigma} y$?
Couverture(CPN)
— Entrée : Un réseau de Petri $N = (P, T, Pre, Post)$, un marquage initial $m_0 \in \mathbb{N}^{|P|}$ et un marquage $m \in \mathbb{N}^{|P|}$.
— Question : $\exists \sigma \in ((0, 1] \times T)^*$ et $y \geq m$ tel que $m_0 \xrightarrow{\sigma} y$?

Si on ne peut pas couvrir m à partir de m_0 dans un système S, on ne pourra pas atteindre m à partir de m_0 dans S.

2.4.3 Complexité

Voici deux brefs tableaux qui présentent la complexité connue à ce jour des problèmes d'accessibilité et de couverture.

Problème	Dimension		
	1	2	≥ 3
Accès	NP-complet [HKOW09]	PSPACE-complet [BFG+14]	$\in F_{\omega^3}$[LS15]
Couverture	$\in P$ [RY86]	PSPACE-complet [FJ13]	$\in$ PSPACE-complet [RY86]

Tableau 1 – Complexité du problème d'accessibilité et de couverture pour les d-VASS

Modèle	Complexité
CPN	P-complet [FH13]
$\mathbb{Z}$-VASS	NP-complet [HH14]

Tableau 2 – Complexité du problème d'accessibilité pour les réseaux de Petri continus (CPN) et pour les VASS sans gardes, c'est-à-dire sans l'interdiction de nombre négatifs dans les compteurs, avec opération de remise à zéro ($\mathbb{Z}$-VASS).

3 Systèmes d'addition de vecteurs continus

Dans cette section seront définis deux modèles de systèmes d'addition de vecteurs continus, c'est-à-dire permettant le franchissement fractionnaire plutôt qu'entier des transitions du système. Ceux-ci ont des comportements, de même que des ensembles d'accessibilité, très différents. D'un point de vue notationnel, chaque transition sera accompagnée par un facteur $\alpha \in (0, 1]$.

Le premier modèle, le *Système d'addition de vecteurs continu avec états multiples* (CVASS_M), peut être dans plusieurs états simultanément. Ainsi, en empruntant une transition $(p, z, q) \in \delta$ selon un facteur α, une fraction α de l'état changera de p vers q. Cette définition donne une caractéristique de fluidité pour les états. Des procédures de conversions seront présentées et l'équivalence avec les réseaux de Petri continus sera explicitée. La motivation derrière ces résultats réside dans le fait qu'il existe déjà un algorithme pouvant résoudre le problème d'accessibilité pour un CPN en temps polynomial.

Le deuxième modèle, le *Système d'addition de vecteurs continu avec états uniques* (CVASS_U), change entièrement d'état à chaque transition empruntée et ce indépendamment du coefficient α. Il peut, entre autre, passer à n'importe quel état subséquent avec des coefficients α qui tendent vers 0. Par son caractère très permissif, son comportement diffère grandement des VASS et des CVASS_M.

3.1 Systèmes d'addition de vecteurs continus avec états multiples (CVASS_M)

Un CVASS_M possède les mêmes données qu'un VASS (ensemble d'états, ensemble de transitions et dimension) mais possède un comportement différent. Chaque transition est accompagnée d'un coefficient $\alpha \in (0, 1]$. On pondère par α le vecteur qui sera additionné aux compteurs de la configuration courante de

même que le changement d'état.

Définition 3.1. Une *configuration d'états* κ est un vecteur de $\kappa \in [0,1]^Q$ où $\kappa[q]$ correspond à la proportion de l'état q dans laquelle le système est. Une configuration d'états possède la caractéristique suivante :

$$\sum_{q \in Q} \kappa[q] = 1$$

Nous allons introduire une nouvelle notation qui nous sera utile plus loin dans ce document. e_q est un vecteur de taille $|Q|$ tel que $e_q[q] = 1$ et $\forall i \in Q/\{q\}, e_q[i] = 0$.

Définition 3.2. Une *configuration d'un d-CVASS*$_M$ est une paire (κ_i, x_i) où κ_i est une configuration d'états et $x_i \in \mathbb{R}^d_{\geq 0}$.

Définition 3.3. Une *exécution d'un CVASS*$_M$ est une liste de configurations telle que la différence entre la configuration (κ_{i+1}, x_{i+1}) et la configuration (κ_i, x_i) correspond à une fraction α_i d'une transition δ_i :

$$\kappa_{i+1} = \kappa_i - \alpha e_p + \alpha e_q$$
$$x_{i+1} - x_i = \alpha u$$

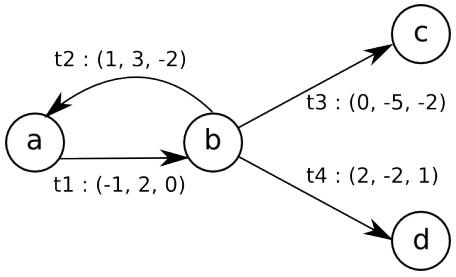
où $(p, u, q) \in \delta$

Notons qu'une exécution ne doit jamais faire tomber sous zéro un compteur.

Définition 3.4. Une *séquence d'activation d'un CVASS*$_M$ est une liste de paires $\sigma \in ((0,1] \times \delta)^*$.

Exemple 3.5. *Soit $V = (Q, \delta)$ un CVASS$_M$ possédant les mêmes informations que la figure 2.3 et x_0 une configuration initiale tels que :*

$$\begin{aligned} Q &= \{a, b, c, d\} \\ \delta &= \{t_1, t_2, t_3, t_4)\} \\ \kappa &= (\kappa[a], \kappa[b], \kappa[c], \kappa[d]) \\ x_0 &= (e_b, (0, 0, 2)) \\ &= ((0, 1, 0, 0), (0, 0, 2)) \end{aligned}$$



L'exécution σ de la séquence d'activation π est valide :

$$\begin{aligned}\pi =& 0.5t_2 0.5t_4 0.5t_1 0.5t_4\\ \sigma =& ((0,1,0,0),(0,0,2))((0.5,0.5,0,0),(0.5,1.5,1))\\ & ((0.5,0,0,0.5),(1.5,0.5,1.5))((0,0.5,0,0.5),(1,1.5,1.5))\\ & ((0,0,0,1),(2,0.5,2))\end{aligned}$$

3.2 Équivalence entre CVASS$_M$ et CPN

3.2.1 Conversion de CVASS$_M$ vers CPN

On peut simuler un CVASS$_M$ à l'aide d'un CPN de la même manière que l'on simule un VASS à partir d'un PN. Pour tout d-CVASS$_M$ $V = (Q, \delta)$, on définit le réseau $C(V) = (P, T, Pre, Post)$ tel que :

1. L'ensemble P a un compteur pour chaque état de V ainsi que d compteurs pour le marquage de V. Nous ferons référence au premier groupe comme l'ensemble P_Q et au deuxième groupe comme l'ensemble P_d. Ainsi, $P = P_Q \cup P_d$.
2. L'ensemble T a une transition t_k pour chaque transition $k \in \delta$ de V.
3. Pour toute transition $k = (p, z, q) \in \delta$, on a :

$$Pre[, t_k] = \begin{pmatrix} e_p \\ -\min(z, 0) \end{pmatrix}$$
$$Post[, t_k] = \begin{pmatrix} e_q \\ \max(z, 0) \end{pmatrix}$$

Soit $k = (p, z, q) \in \delta$, on remarque que dans les compteurs $i \in P_Q$ il y a dans la matrice Pre un seul compteur non-nul correspondant à $Pre[p, t_k] = 1$ et un seul compteur non-nul dans la matrice Post correspondant à $Post[q, t_k] = 1$. On remarque aussi la correspondance entre z et les compteurs $i \in P_d$ tels que $Pre[i, t_k] = -\min(z[i], 0)$ et $Post[i, t_k] = \max(z[i], 0)$.

Voici une procédure qui construit en s'appliquant sur le développement précédent un CPN $C(V)$ à partir d'un CVASS$_M$ V :

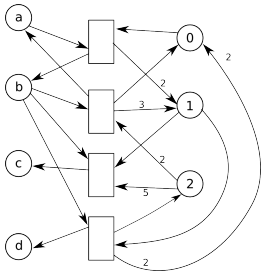
Algorithme 1. Conversion d'un CVASS$_M$ en CPN
Entrée : (Q, δ) un d-CVASS$_M$;
Sortie : Un CPN avec $d + |Q|$ compteurs ;
$P \leftarrow \emptyset, T \leftarrow \emptyset$;
$Pre \leftarrow$ matrix(default_value = 0, size = $|Q| + d \times |\delta|$) ;
$Post \leftarrow$ matrix(default_value = 0, size = $|Q| + d \times |\delta|$) ;
for $j \in Q$ **do**
  $P \leftarrow P \cup p_j$;

end for
for $i \in range(1, d)$ **do**
$P \leftarrow P \cup p_i$;
end for
$k \leftarrow 0$;
for $(i, z, j) \in \delta$ **do**
$T \leftarrow T \cup k$;
$Pre[p_i, k] = 1$;
$Post[p_j, k] = 1$;
for $w \in range(1, d)$ **do**
if $z[w] > 0$ **then**
$Post[w, k] = z[w]$;
end if
if $z[w] < 0$ **then**
$Pre[w, k] = -z[w]$;
end if
end for
$k++$;
end for
return $(P, T, Pre, Post)$;

Exemple 3.6. *La figure 3.7 illustre le CPN obtenu de la conversion du CVASS* $V = (Q, \delta)$ *suivant (figure 2.3) :*

$$\begin{aligned} Q &= \{a, b, c, d\} \\ \delta &= \{(a, (-1, 2, 0), b), (b, (1, 3, -2), a), \\ &\quad (b, (0, -5, -2), c), (b, (2, -2, 1), d)\} \end{aligned}$$

Figure 3.7. *Le CPN* $C(V)$ *résultant de la conversion de la figure 2.3 à travers l'algorithme 1.*



$$P = \{a, b, c, d, 0, 1, 2\}$$
$$T = \{0, 1, 2, 3\}$$
$$Pre = \begin{pmatrix} 1 & 0 & 0 & 0 \\ 0 & 1 & 1 & 1 \\ 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 5 & 2 \\ 0 & 2 & 2 & 0 \end{pmatrix} \qquad Post = \begin{pmatrix} 0 & 1 & 0 & 0 \\ 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \\ 0 & 1 & 0 & 2 \\ 2 & 0 & 0 & 0 \\ 0 & 3 & 0 & 1 \end{pmatrix}$$

3.2.2 Conversion d'exécution de CVASS$_M$ vers CPN

Proposition 3.8. *Soit un CVASS$_M$ V et $\alpha \in (0, 1]$, αk peut être activé à partir de la configuration (κ, x) si et seulement si αt_k peut être activé dans le CPN $C(V)$ à partir du marquage κx . Autrement dit,*

$$\kappa x - \alpha Pre[, t_k] \geq \vec{0} \iff \kappa - \alpha e_p \geq \vec{0} \wedge x + \alpha(\min(z, \vec{0})) \geq \vec{0}$$

Preuve. En utilisant les définitions, la preuve est plutôt directe :

$$\begin{aligned} & \kappa x - \alpha Pre[, t_k] \geq \vec{0} \\ \iff & \kappa x - \alpha \begin{pmatrix} e_p \\ -\min(z, \vec{0}) \end{pmatrix} \geq \vec{0} \\ \iff & \kappa - \alpha e_p \geq \vec{0} \\ & \wedge\ x - \alpha(-\min(z, \vec{0})) \geq \vec{0} \end{aligned}$$

□

Proposition 3.9. *Soit $V = (Q, \delta)$, un CVASS$_M$, $C(V) = (P, T, Pre, Post)$, $\alpha \in (0, 1]$ et $k \in \delta$. Alors $(\kappa, x) \overset{\alpha k}{\to} (\kappa', x')$ dans $V \iff \kappa x \overset{\alpha t_k}{\to} \kappa' x'$ dans $C(V)$.*

Preuve. Nous savons déjà que la transition t_k peut s'activer si et seulement si la transition k peut s'activer. Il est donc plutôt simple de montrer le parallèle entre l'effet des transitions.

$$\begin{aligned} k'x' &= kx + \alpha t_k \\ &= kx - \alpha Pre[, t_k] + \alpha Post[, t_k] \\ &= kx - \alpha \begin{pmatrix} e_p \\ -min(z, 0) \end{pmatrix} + \alpha \begin{pmatrix} e_q \\ max(z, 0) \end{pmatrix} \\ \iff \kappa' &= \kappa - \alpha e_p + \alpha e_q \\ \wedge\ x' &= x - \alpha(-\min(z, 0)) + \alpha \max(z, 0) \\ \iff \kappa' &= \kappa - \alpha e_p + \alpha e_q \\ \wedge\ x' &= x + \alpha z \end{aligned}$$

□

Corollaire 3.10. *Avec les données de la proposition 3.9, il existe une exécution dans V telle que $(\kappa_0, m_0) \xrightarrow{*} (\kappa_n, x_n)$ si et seulement si il existe une exécution dans $C(V)$ telle que $\kappa_0 m_0 \xrightarrow{*} \kappa_n x_n$*

Puisqu'on sait maintenant simuler un CVASS$_M$ par un CPN, et que la conversion s'effectue en temps polynomial, alors on conclut que

Corollaire 3.11. *Accès($CVASS_M$) $\leq_m^p$ Accès(CPN).*

3.2.3 Conversion de CPN vers CVASS$_M$

Inversement, il est possible de construire un CVASS$_M$ $V(C) = (Q, \delta)$ pour simuler un CPN $C = (P, T, Pre, Post)$. Pour ce faire, il faut définir $V(C)$ tel que :

1. $Q = T \cup \{q\}$
2. La dimension de $V(C)$ est des $|P|$
3. Pour toute transition $i \in T$, il y aura la transition δ_{qi} et la transition δ_{iq} dans δ. Celles-ci sont définies de la manière suivante :

$$\delta_{ai} = (q, -Pre[, i], i)$$
$$\delta_{ia} = (i, Post[, i], q)$$

   Ainsi, nous définissons δ_{qi} comme étant l'unique transition de q vers i et δ_{iq} comme étant l'unique transition de i vers q. Il n'y a aucune autre transition dans $V(C)$.

Voici une procédure pour convertir un CPN en CVASS$_M$:

Algorithme 2. Convertit un CPN en CVASS$_M$
Entrée : Un CPN $(P, T, Pre, Post)$;
Sortie : Un $CVASS_M$ (Q, δ) ;
$Q \leftarrow \{q\}, \delta \leftarrow \emptyset$;
for $i \in T$ **do**
  $Q \leftarrow Q \cup i$;
  $\delta \leftarrow \delta \cup (q, -Pre[, i]), i)$;
  $\delta \leftarrow \delta \cup (i, Post[, i]), q)$;
end for
return (Q, δ)

Le coeur de la simulation du CPN par le CVASS$_M$ sera de maintenir une correspondance entre un marquage m du CPN C et une configuration (e_q, m) du CVASS$_M$ $V(C)$.

Exemple 3.12. *La figure 3.14 illustre le $CVASS_M$ obtenu suite à la conversion du réseau de Petri continu de la figure 3.13.*

Figure 3.13. *Un réseau de Petri continu.*

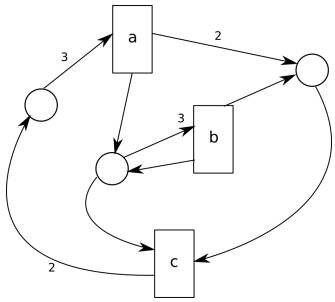
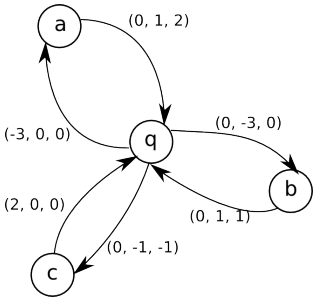
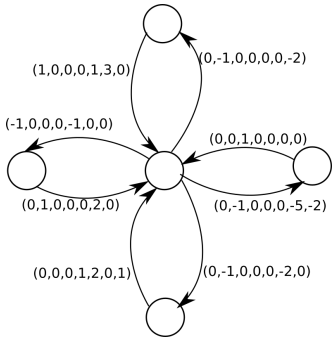


Figure 3.14. *Le $CVASS_M$ résultant de la conversion du réseau de Petri de la figure 3.13.*



Exemple 3.15. *(Remarque) Pour bien illustrer l'augmentation de la dimension (i.e. taille du marquage) dans la conversion de $CVASS_M$ vers CPN, nous allons, à l'aide de la procédure CPN vers $CVASS_M$ reconvertir le résultat de l'exemple 3.6. Cela est illustré à la figure 3.16.*

Figure 3.16. *La reconversion en VASS du réseau de Petri de la figure 3.7.*



On remarque que, au coût d'une augmentation de la dimension, tout $CVASS_M$ peut être mis dans la forme avec un état central d'où entre et sortent tous les arcs du graphe du $CVASS_M$.

3.2.4 Conversion d'exécution de CPN vers CVASS$_M$

Définition 3.17. Dans une exécution d'un CVASS$_M$ $V(C)$ obtenu du CPN C par la méthode de la section précédente, une transition de la forme $\alpha\delta_{ia}$ est dite *non-couplée* si elle n'est pas précédée immédiatement par une transition $\beta\delta_{ai}$ où $\beta \geq \alpha$. Une transition de la forme $\beta\delta_{ai}$ est dite *non-couplée* si elle n'est pas suivie immédiatement de $\alpha\delta_{ia}$ où $\beta \leq \alpha$.

Définition 3.18. Une exécution d'un CVASS$_M$ $V(C)$ est dite *désordonnée* si elle contient au moins une transition non-couplée. Sinon, elle est dite *ordonnée*.

Proposition 3.19. *Soit C un CPN, $V(C)$ un CVASS$_M$ comme ci-dessus et $\alpha \in (0,1]$. Alors pour tout $x, x' \in \mathbb{R}^P_{\geq 0}$, $x \xrightarrow{\alpha t_i} x'$ dans $C \iff (e_a, x) \xrightarrow{\alpha\delta_{ai}\alpha\delta_a} (e_a, x')$ dans $V(C)$.*

Preuve. Nous savons que dans le cas de $V(C)$, la configuration d'états n'est jamais un facteur contraignant car :

$$\begin{aligned} & x - \alpha Pre[,t_i] && \geq \vec{0} \\ \iff & e_a - \alpha e_a + \alpha e_i && \geq \vec{0} \\ & \wedge\, x + \alpha z_{ai} && \geq \vec{0} \end{aligned}$$

De plus, l'effet des transitions sur x est le même dans C et dans $V(C)$:

$$\begin{aligned} x' &= x + \alpha t_i \\ &= x + \alpha(-Pre[,t_i] + Post[,t_i]) \\ &= x + \alpha(-Pre[,t_i]) + \alpha(Post[,t_i]) \\ &= x + \alpha z_{ai} + \alpha z_{ia} \end{aligned}$$

où $\delta_{ai} = (a, z_{ai}, i)$ et $\delta_{ia} = (i, z_{ia}, a)$ □

Corollaire 3.20. *Soit un CPN $C = (P, T, Pre, Post)$ et le CVASS$_M$ $V(C) = (Q, \delta)$. Il existe une exécution de C $m_0 \xrightarrow{*} m$ si et seulement si il existe une exécution ordonnée de $V(C)$ telle que $(e_a, m_0) \xrightarrow{*} (e_a, m)$.*

Proposition 3.21. *Soit une exécution σ de $(e_a, m_0) \xrightarrow{\pi} (e_a, m)$ telle que π possède $x > 0$ transitions δ_{ai} et δ_{ia} non-couplées, alors il existe une exécution σ' d'une séquence d'activation π' qui mène au même marquage final où π' possède $y \leq x - 1$ transitions δ_{ai} et δ_{ia} non-couplées ainsi que le même nombre de transitions $j \neq i$ non-couplées.*

Preuve. Nous pouvons décomposer π de la façon suivante :

$$\pi = \sigma_0 \beta\delta_{ai} \sigma_1 \alpha\delta_{ia} \sigma_2$$

où σ_2 ne contient aucune transition δ_{ia} non-couplée et σ_1 ne contient aucun δ_{ai} non-couplé.

En premier lieu, si $\beta\delta_{ai}$ est suivi immédiatement par $\gamma\delta_{ia}$, on sait que $\beta > \gamma$ sinon δ_{ai} serait couplé. Si c'est le cas, considérons la séquence d'activation équivalente $\pi_1 = \sigma_0(\beta-\gamma)\delta_{ai}\gamma\delta_{ai}\gamma\delta_{ia}\sigma_1\alpha\delta_{ia}\sigma_2$ où on sépare $\beta\delta_{ai}$ en $(\beta-\gamma)\delta_{ai}\gamma\delta_{ai}$. Continuons la preuve avec π_1 à la place de π. Sinon, considérons $\gamma = 0$.

À ce point, on distingue deux cas :

Cas 1 : $\beta - \gamma \geq \alpha$
Dans ce cas, nous posons $\pi' = \sigma_0(\beta-\gamma)\delta_{ai}\alpha\delta_{ia}\gamma\delta_{ai}\gamma\delta_{ia}\sigma_1\sigma_2$. On remarque que la transition $\alpha\delta_{ia}$ qui était non-couplée est maintenant couplée. Le déplacement de cette transition n'invalide pas la séquence car aucun δ_{ai} non-couplé n'était dans σ_1.

Cas 2 : $\beta - \gamma < \alpha$
Dans ce cas-ci, il est préalablement nécessaire de séparer $\alpha\delta_{ia}$ en $(\alpha - (\beta - \gamma))\delta_{ia}(\beta-\gamma)\delta_{ia}$. Par la suite, nous posons $\pi' = \sigma_0(\beta-\gamma)\delta_{ai}(\beta-\gamma)\delta_{ia}\gamma\delta_{ai}\gamma\delta_{ia}\sigma_1(\alpha-(\beta-\gamma))\delta_{ia}\sigma_2$. On remarque que la transition $\beta\delta_{ai}$ qui était non-couplée est maintenant couplée. Le déplacement de cette transition n'invalide pas la séquence car aucun δ_{ai} non-couplé n'était dans σ_1.

□

Corollaire 3.22. *Soit une exécution de* $(e_a, m_0) \xrightarrow{\pi} (e_a, m)$ *où* π *est désordonnée pour* x *transitions différentes, il existe une exécution* $(e_a, m_0) \xrightarrow{\pi'} (e_a, m)$ *équivalente où* π' *est ordonnée pour toutes ses transitions.*

En prenant le corollaire 3.20 et le corollaire 3.22, on obtient :

Corollaire 3.23. *Soit un CPN* $C = (P, T, Pre, Post)$ *et le CVASS*$_M$ $V(C) = (Q, \delta)$. *Il existe une exécution de* C $m_0 \xrightarrow{*} m$ *si et seulement si il existe une exécution de* $V(C)$ *telle que* $(e_a, m_0) \xrightarrow{*} (e_a, m)$.

On sait maintenant qu'il est possible de simuler un CPN par un CVASS$_M$.

Corollaire 3.24. *Accès(CPN)* $\leq_m^p$ *Accès(CVASS*$_M$*).*

Du corollaire 3.11 et 3.24 nous pouvons conclure que

$$\text{Accès}(CPN) \equiv_m^p \text{Accès}(\text{CVASS}_M)$$

3.2.5 Quel est l'intérêt des CVASS$_M$?

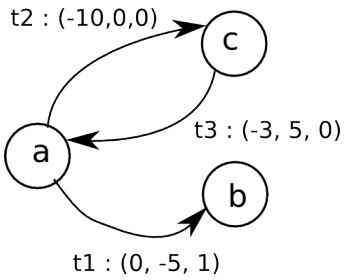
Les CVASS$_M$ sont une variante des VASS avec des contraintes relaxées. Il est simple de démontrer que $(p, v) \xrightarrow{*} (q, u)$ dans un VASS $\Rightarrow (e_p, v) \xrightarrow{*} (e_q, u)$ dans un CVASS$_M$.

3.3 Systèmes d'addition de vecteurs continus avec états uniques (CVASS$_U$)

À la différence des CVASS$_M$, nous définissons la sémantique du CVASS$_U$ de telle sorte que lorsqu'on active une transition selon un coefficient α, l'état change entièrement et non en partie. Cela nous libère de la nécessité d'utiliser le concept de configuration d'états défini précédemment. Un seul état sera encodé dans la configuration.

Définition 3.25. Une configuration de d-CVASS$_U$ possède la même définition que celle d'un VASS, soit une paire avec comme premier élément un état et comme deuxième élément un vecteur $m \in \mathbb{R}^d_{\geq 0}$. Une séquence d'activation d'un CVASS$_U$ possède la même définition que celle d'un CVASS$_M$.

Exemple 3.26. *Soit le CVASS$_U$ suivant et le marquage initial* $(a, (4, 0, 0))$.



L'exécution $\sigma : (a, (4, 0, 0)) \xrightarrow{\pi} (b, (0, 0, 1))$ *est une exécution valide du précédent CVASS$_U$ où*

$$\pi = 0.1t_2 1.0t_3 1.0t_1$$
$$\sigma = (a, (4, 0, 0))(c, (3, 0, 0))(a, (0, 5, 0))(b, (0, 0, 1)).$$

Exemple 3.27. *Soit le VASS de la figure 2.3, interprété comme un CVASS$_U$, et le marquage initial* $(b, (1, 0, 3))$. *La séquence d'activation* π *est un chemin valide pour le CVASS$_U$ partant de* $(b, (1, 0, 3))$ *et arrivant à* $(d, (1, 3, 2.25))$ *où*

$$\pi = 0.5t_2 1t_1 0.25t_4$$
$$\sigma = (b, (1, 0, 3))(a, (1.5, 1.5, 2))(b, (0.5, 3.5, 2))(d, (1, 3, 2.25)).$$

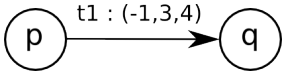
3.3.1 Constat sur le comportement des CVASS$_U$

Le comportement d'un CVASS$_U$ est parfois contre-intuitif et curieux. Il est possible d'activer une transition avec un coefficient α qui tend vers 0 pour vu qu'il n'y ait pas 0 dans un compteur qui doit être décrémenté. Dans ce cas, on passe au prochain état avec un impact quasi-nul sur les compteurs.

Dans un CVASS$_U$ qui contient un cycle franchissable au moins une fois, s'il y a un arc du cycle qui est strictement positif, il est possible d'emprunter le cycle à répétition et d'augmenter arbitrairement le marquage.

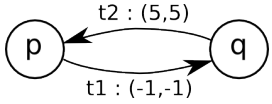
Exemple 3.28. *Soit le CVASS$_U$ représenté par la figure 3.29. Soit* $\epsilon \in (0,1]$, *alors* $(p,(1,0,5)) \xrightarrow{\epsilon t_1} (q,(1-\epsilon, 0+\epsilon, 5+5\epsilon))$. *Donc, si* ϵ *tend vers 0, alors on peut s'approcher arbitrairement de* $(q,(1,0,5))$.

Figure 3.29.



Exemple 3.30. *Soient le CVASS$_U$ représenté par la figure 3.31,* $\epsilon \in (0,1]$, $c \in \mathbb{N}_{>0}$ *et* $\sigma = (\frac{\epsilon}{c-1} t_1 \frac{1}{5} t_2)^{c-1}$, *alors* $(e_p,(1,1)) \xrightarrow{\sigma} (e_p,(c-\epsilon, c-\epsilon))$.

Figure 3.31. *Un CVASS$_U$ contenant un cycle.*



3.3.2 Quel est l'intérêt des CVASS$_U$?

Encore une fois, s'il n'existe pas de chemin $(q,u) \xrightarrow{*} (p,v)$ dans un CVASS$_U$, on sait qu'il n'y a pas de chemin de (q,u) à (p,v) dans le VASS correspondant et si $(q,u) \xrightarrow{*} (p,v)$ dans un VASS, alors le même chemin existe dans le CVASS$_U$ correspondant. Cette observation fait des CVASS$_U$ une relaxation des VASS.

Plusieurs questions peuvent se poser pour ce modèle :

1. Existe-t-il un modèle équivalent dans un différent type de système, par exemple un réseau de Petri ?
2. Peut-on caractériser les ensembles d'accessibilité des CVASS$_U$?
3. Est-ce que le modèle est trop permissif ? Permet-il d'atteindre tellement plus de configuration que les VASS, que très rarement un test d'accessibilité nous apportera de l'information ?

4 Implémentation d'un algorithme de résolution du problème d'accessibilité dans les CPN

Le module développé consiste en un algorithme permettant de résoudre le problème d'accessibilité pour les CPN. Les liens entre VASS et CVASS$_M$ étant présentés dans la précédente section, nous ne discuterons que des détails de l'implémentation de l'algorithme dans cette section. L'algorithme implémenté est tiré de [FH13]. Une représentation des réseaux de Petri ainsi que leurs manipulations présentées dans la section 2 ont été développées en Python à l'aide de la librairie *Numpy*.

4.1 Algorithmes de Fraca et Haddad

L'algorithme pour l'accessibilité des CPN de Fraca et Haddad se présente en deux parties. La première est l'algorithme `Fireable` qui permet de déterminer si toutes les transitions d'un ensemble peuvent être activées à partir d'un marquage donné. La deuxième partie est l'algorithme `Reachable` qui utilise `Fireable` et résoud le problème d'accessibilité pour les CPN.

L'algorithme `Fireable` illustré à la figure 4.1 permet de vérifier en temps polynomial s'il est possible, étant donné une configuration initiale m_0, de déclencher un sous-ensemble $T' \subseteq T$ de transitions d'un réseau de Petri continu $N = (P, T, Pre, Post)$. En cas d'échec, l'algorithme retourne le plus grand ensemble $T'' \subset T'$ de transitions qui peuvent être activées à partir de m_0.

La fonction `maxFs` est définie comme étant l'appel de la fonction `Fireable` avec $T' = T$. C'est celle-ci qui sera appelée dans l'algorithme de résolution du problème d'accessibilité.

L'algorithme `Reachable` illustré à la figure 4.2 représente le test d'accessibilité en soi. Celui-ci reçoit en entrée un réseau de Petri N, un marquage initial m_0, un marquage final m ainsi qu'une valeur booléenne indiquant si l'on souhaite tester la *lim*-accessibilité plutôt que l'accessibilité. Cette dernière valeur est par défaut fausse. Lorsque le marquage m est inaccessible à partir de m_0, l'algorithme retourne la valeur booléenne fausse. Lorsque qu'il existe une exécution $m_0 \xrightarrow{\sigma} m$, l'algorithme retourne $\vec{\sigma}$, l'image de Parikh de σ. L'image de Parikh $\vec{\sigma}$ d'une séquence d'activation σ d'un CPN est un vecteur de taille $|T|$ où $\forall\ t \in T,\ \vec{\sigma}[t] = \sum_{\alpha t \in \sigma} \alpha$.

Pour les preuves relatives à ces algorithmes, veuillez consulter [FH13].

Figure 4.1. *Fireable tiré de [FH13].*

Algorithm 1: Decision algorithm for membership of $FS(\mathcal{N}, \boldsymbol{m}_0)$

`Fireable`$(\langle \mathcal{N}, \boldsymbol{m}_0 \rangle, T')$: status
Input: a CPN system $\langle \mathcal{N}, \boldsymbol{m}_0 \rangle$, a subset of transitions T'
Output: the membership status of T' w.r.t. $FS(\mathcal{N}, \boldsymbol{m}_0)$
Output: in the negative case the maximal firing set included in T'
Data: *new*: boolean; P': subset of places; T'': subset of transitions

```
1 T'' ← ∅; P' ← [[m_0]]
2 while T'' ≠ T' do
3 |   new ← false
4 |   for t ∈ T' \ T'' do
5 |   |   if •t ⊆ P' then T'' ← T'' ∪ {t}; P' ← P' ∪ t•; new ← true
6 |   end
7 |   if not new then return (false, T'')
8 end
9 return true
```

Figure 4.2. *Reachable tiré de [FH13].*

Algorithm 2: Decision algorithm for reachability

```
   Reachable(⟨𝒩, m₀⟩, m): status
   Input: a CPN system ⟨𝒩, m₀⟩, a marking m
   Output: the reachability status of m
   Output: the Parikh image of a witness in the positive case
   Data: nbsol: integer; v, sol: vectors; T': subset of transitions
 1 if m = m₀ then return (true,0)
 2 T' ← T
 3 while T' ≠ ∅ do
 4 |   nbsol ← 0; sol ← 0
 5 |   for t ∈ T' do
 6 |   |   solve ∃?v  v ≥ 0 ∧ v[t] > 0 ∧ C_{P×T'} v = m − m₀
 7 |   |   if ∃v then nbsol ← nbsol + 1; sol ← sol + v
 8 |   end
 9 |   if nbsol = 0 then return false else sol ← (1/nbsol) sol
10 |   T' ← ⟦sol⟧
11 |   T' ← T' ∩ maxFS(𝒩_{T'}, m₀[•T'•])
12 |   T' ← T' ∩ maxFS(𝒩⁻¹_{T'}, m[•T'•]) /* deleted for lim-reachability    */
13 |   if T' = ⟦sol⟧ then return (true,sol)
14 end
15 return false
```

Remarquons l'unique inégalité stricte dans le système d'inégalités à résoudre à la ligne 6. Celle-ci nous obligera à faire plus que rentrer des données brutes dans un solveur puisque peu ou pas de solveurs peuvent gérer les inégalités strictes.

4.2 Choix du langage, librairies et dépendances

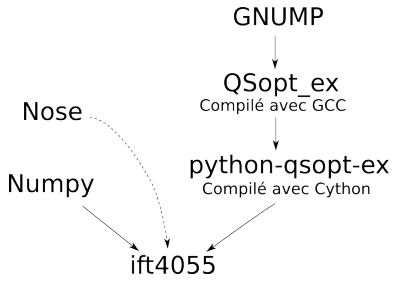
4.2.1 Python

Le langage choisi pour l'implémentation de cet algorithme est Python. Python est un langage de haut niveau implémentant des éléments de la programmation orientée objet, de la programmation fonctionnelle et de la programmation impérative. Dans les grandes lignes, Python est typé dynamiquement et sa mémoire est gérée automatiquement à l'aide d'un mécanisme de comptage de références et d'un ramasse-miettes pour la destruction des cycles. Simple et élégant, la lisibilité du code est au coeur de la philosophie du langage. La version utilisée pour le développement du projet est la 3.4.2.

4.2.2 Librairies et dépendances

Numpy :	Librairie de calcul numérique pour Python
QSopt-ex :	Librairie basée sur QSopt qui résoud le simplexe de manière exacte.
Cython :	Librairie de liaisons C/C++ pour Python
python-qsopt-ex :	Liaison existante entre QSopt-ex et Python développé en Cython par jonls.
GNUMP :	Le GNU Multiple Precision Arithmetic Library [1] nous intéresse notamment pour les fonctions et représentations des nombres rationnels.
Nose :	Module pour développer et exécuter des tests en Python.

Figure 4.3. *Schéma des dépendances du module.*



4.2.3 QSopt_ex

Basé sur QSopt mais utilisant la librairie GNU MP Bignum, QSopt_ex [2] est un solveur exact, c'est-à-dire un solveur qui ne retourne jamais de solution inexacte. Plutôt que d'effectuer tous ses calculs dans $\mathbb{Q}$, QSopt_ex favorise les calculs en arithmétique en virgule flottante puis vérifie sa solution dans $\mathbb{Q}$. En cas d'échec, QSopt_ex augmente la précision et poursuit la résolution à l'aide de l'information acquise, plus précisément à l'aide de la base obtenue par le simplexe. QSopt_ex commence les calculs en 64 bits, puis double à 128 bits et, pour toutes les itérations suivantes, augmente la précision d'un facteur d'environ 1.5 de manière à ce que le nombre soit un multiple de 32. Si la solution trouvée ne passe pas les tests de vérification après que la précision ait atteint 3264 bits, la résolution sera complétée entièrement dans $\mathbb{Q}$. [3]

2. Page officielle : `http://www.math.uwaterloo.ca/~bico/qsopt/ex`
3. Pour plus de précision, voir [Esp06] p.39

Figure 4.4. *Algorithme à la base de QSopt_ex, tiré de [ACD06]. Une version plus détaillée est disponible dans [Esp06] p.39 .*

Algorithm 1. `Exact_LP_Solver`
Require: $c \in \mathbb{Q}^n$, $b \in \mathbb{Q}^m$, $A \in \mathbb{Q}^{m \times n}$
1: Start with the best native floating-point precision p (number of bits for floating-point representation).
2: Set $\mathcal{B} \leftarrow \emptyset$.
3: Compute approximations $\bar{c}, \bar{b}, \bar{A}$ of the original input in the current floating-point precision.
4: Solve $\min\{\bar{c}x : \bar{A}x \leq \bar{b}\}$ using $\mathcal{B}$ (if it is not empty) as the starting basis.
5: $\mathcal{B} \leftarrow$ ending basis of the simplex algorithm.
6: Test result in rational arithmetic.
7: **if** Test fails **then**
8: Increase precision p
9: **goto** step 3
10: **end if**
11: **return** x^*

QSopt_ex est le solveur utilisé pour résoudre le système d'inégalités contenu dans l'algorithme à implémenter. La librairie QSopt_ex a été modernisée avec autotools par jonls [4] et c'est cette version qui est utilisée dans le projet.

4.2.4 python-qsopt-ex

La librairie de liaison Cython pour QSopt_ex, python-qsopt-ex, a été développé par jonls. Il est possible de récuperer la liaison adaptée pour Python ≥ 3 sur mon compte GitHub [5] ou par courriel à antakivi@iro.umontreal.ca.

4.2.5 Nose

La librairie pour le développement de tests Nose [6] n'est pas nécessaire pour faire fonctionner le programme mais est nécessaire pour effectuer les tests.

Une fois le module installé, on peut passer les tests avec la commande :

```
nosetests3 test/
```

4.3 Détails de l'implémentation

4.3.1 Encodage des réseaux de Petri

Plus tôt dans ce document, nous avons défini un réseau de Petri comme étant constitué de quatre composantes (P, T, Pre, $Post$). Dans l'implémentation, les matrices Pre et $Post$ sont encodées comme une seule matrice de taille $|P| \times |Q|$

4. `https://github.com/jonls/qsopt-ex`
5. `https://github.com/DjAntaki/python-qsoptex`
6. Page officielle : `https://nose.readthedocs.org`

où chaque élément est un tuple $(pre_{ij}, post_{ij})$ constitué de deux entiers non signés.

Afin de garder la structure aussi légère que possible, nous considérons l'ensemble P et l'ensemble T comme étant spécifiés implicitement par la taille des matrices Pre et $Post$ ($|P| \times |Q|$). Ainsi, l'index d'une colonne sert d'identifiant à la transition qu'elle représente et l'index d'une ligne sert à identifier une place.

Les réseaux de Petri sont des instances de `numpy.ndarray` qu'il est possible d'instancier avec la fonction `numpy.matrix()`. Le paramètre `dtype` sert à déterminer le ou les types de données ('uint' :'unsigned integer', 'int64' :'64-bits signed integer',...).

Exemple 4.5. *Initialisation d'un réseau de Petri et accès à ses paramètres.*

```
>>> net = np.matrix(
...      [[(2, 0), (0, 0), (3, 8), (1, 2), (0,37)],
...       [(0, 3), (1, 3), (5, 0), (5, 2), (23,0)]],
...       dtype=[('pre', 'uint'), ('post', 'uint')])
>>> net['pre']
matrix([[ 2,  0,  3,  1,  0],
        [ 0,  1,  5,  5, 23]], dtype=uint64)
>>> net['post']
matrix([[ 0,  0,  8,  2, 37],
        [ 3,  3,  0,  2,  0]], dtype=uint64)
>>> net.shape
(2, 5)
```

4.3.2 Manipulation des réseaux de Petri

Voici les fonctions de bases pour manipuler les réseaux de Petri :

- preset(net, $S \subseteq T$, $place$) :
  Soit net un réseau de Petri, S une liste d'indices ordonnée représentant un sous-ensemble des transitions de net et $place$ une valeur booléenne. La fonction retourne le preset de l'ensemble de transitions (resp. places si $place ==$ True) en entrée.
- postset(net, $S \subseteq T$, $place$) :
  Soit net un réseau de Petri, S une liste d'indices ordonnée représentant un sous-ensemble des transitions de net et $place$ une valeur booléenne. La fonction retourne le postset de l'ensemble de transitions (resp. places si $place ==$ True) en entrée.
- subnet(net, $S \subseteq T$, $subplace$) :
  Soit net un réseau de Petri, S une liste d'indices ordonnée représentant un sous-ensemble des transitions de net et $place$ une booléenne. La fonction

retourne un réseau de Petri qui n'a conservé que les transitions de S. Si le paramètre *subplace* == True, on retire du réseau retourné toute place qui n'est pas dans le preset ou dans le postset des transitions de S. On retournera dans ce cas, une paire constituée du sous-réseau et de l'ensemble des places conservées.

- ◇ reverse_net(*net*)
  Retourne le réseau de Petri renversé du réseau en entrée.
- ◇ incidence(*net*)
  Retourne la matrice d'incidence du réseau de Petri *net*. Celle-ci a comme éléments des entiers signés.

Exemple 4.6. *Exemple de manipulations des réseaux de Petri.*

```
>>> net
matrix([[(2, 0), (0, 0), (3, 8), (1, 2), (0,37)],
        [(0, 3), (1, 3), (5, 0), (5, 2), (23,0)]],
        dtype=[('pre', 'uint'), ('post', 'uint')])
>>> incidence(net)
matrix([[ -2,  0,  5,  1,  37],
        [ 3,  2,  -5,  -3, -23]], dtype=int64)
>>> reversed_net(net)
matrix([[(0, 2), (0, 0), (8, 3), (2, 1), (37,0)],
           [(3, 0), (3, 1), (0, 5), (2, 5), (0,23)]],
           dtype=[('pre', 'uint'), ('post', 'uint')])
>>> preset(net,[4])
array([1])
>>> preset(net,[0], place=True)
array([2, 3, 4])
>>> preset(net,[1], place=True)
array([0, 1, 3])
>>> preset(net,[0,1], place=True)
array([0, 1, 2, 3, 4])
>>> subnet(net,[0,2,3])
matrix([[(2, 0), (3, 8), (1, 2)],
        [(0, 3), (5, 0), (5, 2)]],
       dtype=[('pre', '<u8'), ('post', '<u8')])
>>> subnet(net,[1],subplaces=True)
(matrix([[(1, 3)]],
dtype=[('pre', '<u8'), ('post', '<u8')]), array([1]))
```

4.3.3 Solveur

Peu de solveurs de problèmes de programmation linéaire travaillent dans les rationnels et aucun trouvé ne prend en charge une contrainte d'inégalité stricte sur une seule variable. Nous avons donc choisi d'utiliser QSopt_ex, une

librairie qui applique l'algorithme du simplexe et qui retourne une solution dans les rationnels. Nous avons utilisé un peu d'astuce pour appliquer la contrainte stricte. L'objectif est de trouver une solution au problème initial suivant :

$$\textbf{Résoudre } \exists ?v \textbf{ tel que } v \geq 0 \wedge v[t] > 0 \wedge C_{P\times T'}v = m - m_0$$

Puisque notre objectif est de trouver une solution **faisable** et qu'il n'y a aucun critère d'optimalité, nous pouvons simuler la contrainte stricte à l'aide d'un objectif. Celui-ci visera à maximiser la variable qui est soumise à l'inégalité stricte. L'approche implémentée est la suivante :

1. Faire le simplexe en excluant la contrainte d'inégalité stricte en ajoutant l'objectif qui vise à maximiser t :

$$\textbf{Maximiser } t$$
$$\textbf{Sujet à } C_{P\times T'}v = m - m_0 \wedge v \geq 0$$

   - Si le simplexe répond que le système sans la contrainte stricte est infaisable, on conclut l'infaisabilité du système avec la contrainte stricte et on arrête la procédure.
   - Si le simplexe donne une solution optimale, on vérifie que $c[t] \neq 0$. Si la condition est vérifiée, on sait que le système avec l'inégalité stricte est satisfaisable et que la solution retournée est une solution valide. Dans ce cas, on arrête la procédure et retourne la solution obtenue. Si la condition n'est pas respectée, on sait que le problème avec la contrainte stricte est insatisfaisable. On arrête la procédure et retourne le statut "Infaisable".
   - Si le simplexe répond que le problème est non borné, on sait qu'il existe une solution au système avec l'inégalité stricte mais on ne sait pas laquelle (QSopt_ex ne retourne aucune base lorsque la solution est non bornée). Dans ce cas, on passe à l'étape 2.

2. Dans le cas où on sait qu'il existe une solution mais qu'on ne sait pas laquelle, nous faisons arbitrairement une "coupe" dans le problème en rajoutant la contrainte $v[t] \geq i > 0$ et adaptons l'objectif pour minimiser. Dans l'implémentation, $i = 1$.

$$\textbf{Minimiser } t$$
$$\textbf{Sujet à } C_{P\times T'}v = m - m_0 \wedge v \geq 0 \wedge v[t] \geq i$$

   Étant donné le résultat de l'étape 1, ce système possède toujours une solution.

Cette approche qui requiert un maximum de deux résolutions du simplexe est construite de manière à éliminer tous les cas infaisables à la première résolution. Alternativement, il serait possible d'appliquer l'approche suivante :

1. Nous faisons arbitrairement une "coupe" dans le problème initial en rajoutant la contrainte $v[t] \geq i > 0$ et définissons l'objectif comme étant de minimiser $v[t]$.

$$\textbf{Minimiser } t$$
$$\textbf{Sujet à } C_{P\times T'}v = m - m_0 \wedge v \geq 0 \wedge v[t] \geq i$$

   - Si une solution est trouvée, on arrête la procédure et retourne cette solution.
   - Si la résolution retourne le statut "Infaisable", on passe à l'étape suivante.

2. Résoudre le système suivant :

$$\textbf{Maximiser } t$$
$$\textbf{Sujet à } C_{P\times T'}v = m - m_0 \wedge v \geq 0 \wedge v[t] \leq i$$

   - Si une solution est trouvée, on retourne cette solution.
   - Si la résolution retourne le statut "Infaisable", on sait que le problème initial n'est pas satisfiable.

Certains cas faisables s'arrêteraient après une seule résolution. Si on réduit la coupe arbitraire i, on réduit le nombre de cas faisables qui se rendent à la deuxième résolution. Par contre, les cas infaisables feraient deux résolutions au lieu d'une avant de conclure.

4.3.4 Programme

Voici des extraits du code source du projet.

Figure 4.7. *Implémentation des algorithmes Fireable et maxFS.*

```
def fireable(net, m, t1):
    assert net.shape[0] == m.shape[0]
    t2,p = np.empty((0,)), m.nonzero()[0]
    while np.setdiff1d(t1,t2).size != 0:
        new = False
        for t in np.setdiff1d(t1,t2) :
            if all(np.in1d(preset(net, [t]),p,
                assume_unique=True)) :
                t2, = np.union1d(t2,[t])
                p = np.union1d(p, postset(net,[t]))
                new = True
        if not new : return (False, t2)
    return (True,t2)

def maxFS(net, m):
    return fireable(net,m,np.array(range(0,net.shape
        [1])))[1]
```

Figure 4.8. *Implémentation de l'algorithme Reachable.*

```
def reachable(net, m0, m, limreach=False):
    n1, n2   = net.shape
    assert len(m) == n1 and n1 == len(m0)

    if np.array_equiv(m,m0) :
        return [0 for x in range(0,n2)]

    #initially, t1 represents all the transitions of
       the Petri net system
    t1 = np.array(range(0,n2))
    b_eq = np.array(m - m0)

    while t1.size != 0:
        l = t1.size
        nbsol, sol = 0,np.zeros(l,dtype=Fraction)
        A_eq = incident(subnet(net,t1))

        for t in t1:
            objective = [minus_one_if_equal(t,x) for x
                in range(0,l)]
            #Call qsopt_ex to get the job done.
            result = solve_qsopt(objective, A_eq, b_eq
                , t)
            if result is not None :
                nbsol += 1
                sol += result

        if nbsol == 0 :
            #No solution, there is no combination of
               transitions that equals to m-m0.
            return False

        else :
            # in numpy, sol *= 1/nbsol cast sol in
               float and we do not want that to happen
               .
            y = Fraction(1,nbsol)
            sol = [x * y for x in list(sol)]
            sol = np.array(sol,dtype=Fraction)

        t1 = sol.nonzero()[0]
        sub, subplaces = subnet(net, t1, True)
        t1 = np.intersect1d(t1, maxFS(sub, m0.take(
           subplaces)),assume_unique=True)
```

```
        if not limreach:
            t1 = np.intersect1d(t1, maxFS(reversed_net
                (sub), m.take(subplaces)),assume_unique
                =True)

        if np.array_equiv(t1,sol.nonzero()) :
            #Found a solution. yay.
            return sol

    return False
```

4.4 Installation du module

4.4.1 Installation de Numpy, GNUMP, Cython et Nose

Avec un gestionnaire de paquets sous Linux, la commande pour télécharger est la suivante :

```
apt-get install numpy cython gnump nosestests3
```

4.4.2 Installation de QSopt_ex

Il faut s'assurer d'avoir GnuMP d'installé avant la compilation de QSopt_ex car il est nécessaire à son fonctionnement. Avec la version de la librairie QSopt_ex de jonls [7], l'installation se simplifie à

```
./bootstrap
make
sudo make install
```

Une fois installé, les liaisons Cython peuvent être construites et installées

```
python3 setup.py build
python3 setup.py install --user
```

Ce qui permet par la suite d'utiliser la librarie QSopt_ex en Python à l'aide de la commande

```
import qsoptex
```

5 Conclusion

5.1 Résumé

Voici un bref retour sur le contenu de ce document produit dans le cadre du cours IFT4055 - Projet informatique honor.

7. https://github.com/jonls/qsopt-ex

À la section 3, nous avons défini deux nouveaux modèles de VASS continus. Les CVASS_M ont été prouvés équivalent aux CPN. Les CVASS_U sont une nouvelle relaxation des VASS. Quelques traits de leur comportement ont été soulignés mais les CVASS_U n'ont pas fait l'objet d'une analyse approfondie.

À la section 4, l'algorithme de [FH13] a été présenté et l'implémentation en Python fut détaillée de même qu'une représentation légère des réseaux de Petri à l'aide de *Numpy*. L'utilisation d'un solveur exact de simplexe pour résoudre le système d'inégalités linéaires avec une inégalité stricte sur une variable a été explicitée. La suite du développement de ce module sera de mettre en place des solveurs alternatifs et de comparer leur efficacité.

5.2 Questions ouvertes et travaux futurs

5.2.1 CVASS_U

Le problème d'accès dans les CVASS_U reste inexploré dans le cadre de ce travail. Cette relaxation, à priori, ne ressemble à aucun modèle connu. Si nous parvenions à caractériser leur ensemble d'accessibilité d'une certaine façon, il nous serait peut-être possible de résoudre le problème d'accessibilité. Si nous étions capables de le résoudre de manière efficace, il pourrait être utile de créer un module à part entière pour traiter les CVASS_U.

5.2.2 Utilisation d'une fonction de rappel dans un solveur

Le simplexe est un algorithme d'optimisation et nous cherchons une solution faisable pour un système donné. L'optimalité de la solution ne nous importe pas. Il serait donc intéressant d'avoir une fonction de rappel, c'est-à-dire qui est appelée après chaque itération du simplexe, et qui arrêterait le simplexe durant sa phase d'optimisation dès que la contrainte de positivité stricte est respectée. À défaut d'arrêter le simplexe, une fonction qui permettrait de stocker la première solution qui respecte la contrainte serait suffisante et nous permettrait de ne pas effectuer plus d'une résolution de simplexe dans le problème tel qu'expliqué à la section 4.

5.2.3 Utilisation d'un solveur avec arithmétique en virgule flottante

Il devrait être possible d'utiliser un solveur avec arithmétique en virgule flottante et d'inférer (ou tenter d'inférer), à partir de la base du simplexe [Mon09] ou à partir d'une heuristique, une solution dans les rationnels. Cette solution pourrait par la suite être vérifiée et, si elle est correcte, nous épargner d'utiliser un solveur exact. Une version qui infère le nombre rationnel à partir d'une heuristique très limitée à été implémentée à l'aide de Scipy mais, faute de temps, celle-ci n'a pas pu être testée assez exhaustivement pour confirmer qu'elle réduisait le temps d'exécution. Celle-ci a été retirée de la version actuelle du module.

5.2.4 Évaluation comparative de multiples solveurs

Il est relativement simple de changer le solveur utilisé dans le programme. Il est moins simple de trouver un solveur capable de résoudre le problème de façon exacte dans les rationnels. Avec plusieurs solveurs, il sera possible de comparer leur efficacité. Il serait par ailleurs possible de comparer l'efficacité des différentes approches de résolution à travers QSopt_ex.

5.2.5 Modules suivants

Comme indiqué dans la première section du document, le développement du module s'inscrit dans une perspective plus large soit le développement d'un outil permettant la résolution partielle du problème d'accessibilité des VASS, notamment en prouvant la non accessibilité en traitant itérativement des relaxations du système. Les prochains modules devraient être des algorithmes résolvant le problème d'accessibilité ou le problème de couverture pour d'autres relaxations des VASS. Les $\mathbb{Z}$-VASS sont de bons candidats potentiels pour le prochain module.

Remerciements

Merci à Guillaume Poirier-Morency pour son aide avec les différentes librairies. Merci à Michael Blondin pour le soutien et la révision de ce rapport. Merci à Pierre Mckenzie de m'avoir proposé de participer à ce projet.

Références

[ACDE06] David L. APPLEGATE, William COOK, Sanjeeb DASH et Daniel G. ESPINOZA : Exact solutions to linear programing problems, 2006.

[Esp06] Daniel G. Espinoza : On Linear Programming, Integer Programming and Cutting Planes. Thèse de Doctorat, Georgia Institute of Technology, 2006.

[FJ13] John Fearnley et Marcin Jurdzinski : Reachability in Two-Clock Timed Automata Is PSPACE-Complete. *Automata, Languages, and Programming - 40th International Colloquium*, pages 212-223, 2013.

[MO12] Diego C. B. Oliveira et David Monniaux : Experimenting on the feasability of using a floating-point simplex in an SMT solver. *PAAR-2012. Third Workshop on Practical Aspects of Automated Reasoning*, p.19-28, 2012.

[Pet62] Carl Adam Petri : Kommunikation mit Automaten. Thèse de Doctorat, Université de Hamburg, 1962.

[HP79] John E. Hopcroft et Jean-Jacques Pansiot : On Reachability problem for 5-dimensional vector addition systems. *Theoretical Computer Science*, 8 :135-159, 1979.

[KM69] Richard M. Karp and Raymond E. Miller : Parrallel program schemata. *Journal of Computer and System Sciences*, pages 147-195, 1969.

[LS15] Jérôme Leroux et Sylvain Schmitz : Reachability in vector addition systems demystified. *LICS*, 2015.

[RY86] Louis E. Rosier et Hsu-Chun Yen : A multiparameter analysis of boundedness problem for vector addition systems. *Journal of Computer and System Sciences*, pages 105-135, 1986.

[RAC78] Charles Rackoff : The covering and boundedness problems for vector addition systems. *Theoretical Computer Science*, 6 :223-231, 1978.

[BFG+14] Michael Blondin, Alain Finkel, Stefan Göller, Christoph Haase et Pierre McKenzie : Reachability in two dimensionnal vector addition systems with states is PSPACE-Complete. *LICS*, 2015.

[May81] Ernst W. Mayr : An algorithm for the general petri net reachability problem, *STOC*, pages 238-246, 1981.

[Ler12] Jérôme Leroux : Vector addition systems reachability problem (a simpler solution). *Turing 100 The Alan Turing Centenary*, pages 214-228, 2012.

[HKOW09] Christoph Haase, Stephan Kreutzez, Jöel Ouaknine et James Worrell : Reachability in succint automata and parametric one-counter automata. *CONCUR*, page 369-383, 2009.

[HH14] Christoph Haase et Simon Halfon : Integer vector addition systems with states. *Reachability Problems*, pages 369-383, 2009.

[FH13] Estìbaliz Fraca et Serge Haddad : Complexity analysis of continuous petri nets. *Petri Nets*, pages 170-189, 2013.

[JRS03] Jorge Júlvez, Laura Recalde et Manuel Silva : On Reachabilty in Autonomous Continuous Petri Net Systems. *Applications and Theory of Petri Nets*, 2003.

[Blo14] Michael Blondin : Algorithmique et complexité des systèmes à compteurs. Rapport pré-doctoral, Université de Montréal et ENS Cachan, 2014.

[Reu89] Christophe Reutenauer : Aspects mathématiques des réseaux de Pétri. *Masson*, Paris, 1989.

[PS82] Christos H. Papadimitriou and Kenneth Steiglitz : Combinatorial Optimization, Algorithms and Complexity. *Prentice-Hall*, 1982.