T.E.R. : analyse de l'article *The power of Well-Structured Transition Systems*

Blot Elliott - Lasjaunias Mathieu - Al Najjar Yacine

Encadré par M. Zeitoun

14 novembre 2016

Table des matières

Table des figures

Introduction

Initialement prévu comme une analyse de l'article [Schmitz and Schnoebelen, 2014], le sujet de ce T.E.R. a finalement dévié quelque peu et se résume à la compréhension de plusieurs parties des articles [Schmitz and Schnoebelen, 2014] et [Finkel and Schoebelen, 2001] ainsi qu'une implémentation d'un algorithme en Python. Il nous a semblé intéressant de regarder plus en détail quelques résultats théoriques sur les beau pré-ordres qui sont à la base des systèmes de transitions bien structurés (les objets d'études centraux des deux articles susmentionnés, entre autres).

Les systèmes de transitions bien structurés (en anglais *"Well-Structured Transition Systems"* ou *WSTS*) comptent parmi les différentes familles de modèles de calcul, au même titre que les automates finis (finite state automata ou FSA) et les machines de Turing. On peut voir sur la figure ci-dessous un exemple simple d'automates finis.

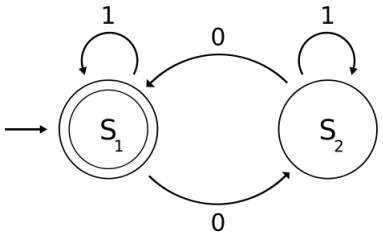

FIGURE 1 – Exemple d'automate vérifiant si le nombre de 0 est paire dans un nombre en binaire

Les automates à nombres d'états fini sont des modèles de calcul simples : un mot est lu lettre par lettre en suivant les transitions étiquetées par les lettres lues et ce mot est accepté si et seulement si on arrive dans un état final. Sur la Figure 1 l'état initial est S_1 (pointé par une flèche venant de nulle part) et l'état final est aussi S_1 (symbolisé par un sommet doublement cerclé). Les algorithmes représentés par des automates finis sont facilement analysables contrairement aux machines de Turing avec lesquelles la majorité des problèmes intéressants sont indécidables (d'après le théorème de Rice qui stipule que toute propriété

non-trivial des langages semi-décidables est indécidable).

Parmi ces problèmes indécidables, l'un des plus fondamental est certainement le problème que nous appellerons Terminaison et dont un énoncé plus formel est donné ci-dessous. Empiriquement il consiste à savoir si un programme s'arrête ou pas étant donné une configuration d'entrée.

— *Terminaison* :
ENTRÉE : Une configuration s, un système de transition
PROBLÈME : Est ce que tout calcul à partir de s est de longueur finie ?

Ce problèmes, restreint aux systèmes de transition engendrés par des automates finis, est décidable. En effet le nombre de configurations d'un automate fini est fini (d'où le terme d'automates $finis$, abréviation d'automates à états finis). Donc on peut construire entièrement le graphe orienté des configurations (à partir d'une configuration d'entrée) et il est possible de voir si un calcul boucle ou termine. Pour les machines de Turing on peut cependant être confronté à une situation intermédiaire : le graphe des configurations peut être infini.

Les *WSTS* ont justement été introduits pour combler ce gouffre i.e. pouvoir étudier des modèles de calcul avec un nombre de configurations infini mais qui grâce à leur structure particulière admettent des algorithmes décisionnels pour répondre à ce type de problème.Par exemple le problème ci-dessus a été prouvé semi-décidable mais indécidable lorsque la question est posé pour les systèmes qui sont des machines de Turing (d'après le théorème de Rice) tandis qu'il s'est révélé être décidable sur *WSTS* comme nous le verrons par la suite.

Concrètement les *WSTS* sont constitués d'un ensemble d'états, équipé d'une relation de beau quasi-ordre (*well quasi-order* ou *wqo*) avec laquelle la relation de transition est compatible. Cette structure particulière permet d'obtenir des résultats génériques de décidabilité pour plusieurs problèmes, notamment *Terminaison* et un autre problème important appelé *Couverture* que nous étudierons en détail plus loin.

En premier lieu nous allons définir en détail ce qu'est un *WSTS*, puis nous présenterons les quelques *WSTS* les plus connus et étudiés à ce jour. Ensuite nous étudierons en détail deux des principaux problèmes et verrons les différences en terme de décidabilité, par rapport aux machines de Turing.

Première partie

Le modèle de calcul

Chapitre 1

Un peu de théorie des beaux pré-ordre (wqo)

Avant de parler de ce qu'est concrètement un *WSTS* dans le deuxième chapitre, nous allons devoir introduire trois notions importantes, nécessaires pour la définition de *WSTS* : la notion de *système de transiton*, la propriété de *monotonicité* d'un système de transition, et la notion de *well-quasi-order*(ou *wqo*). Dans le présent chapitre nous commencerons par étudier cette dernière notion en développant une partie de la théorie des wqo.

1.1 Beau pré-ordre (wqo)

Nous rappelons qu'un ensemble quasiment ordonné ou encore un quasi-ordre (qo) est un ensemble S munit d'une relation $\leqslant$ ayant les propriétés suivantes :

1. transitivité $\overset{\text{def}}{\Leftrightarrow} \forall x,y,z,\ (x \leqslant y \text{ et } y \leqslant z) \Rightarrow x \leqslant z$
2. réflexivité $\overset{\text{def}}{\Leftrightarrow} \forall x, x \leqslant x)$

La définition de beau quasi-ordre (wqo) fait intervenir la notion de *bonne* suite. Étant donné un ensemble S, on appelle une *bonne* suite, une suite $s_0, s_1, ...$ telle qu'il existe $s_i \leqslant s_j$ pour un certain couple d'indices $i < j$. Une suite ne satisfaisant pas cette propriété est naturellement qualifiée de *mauvaise* suite.

Définition 1. *Un quasi-ordre* $(S, \leqslant)$ *est* bon (well) *si toute suite infinie* $s_0, s_1, ...$ *à valeurs dans* S *est une bonne suite. Autrement dit toute mauvaise suite est finie. Nous appellerons un tel bon pré-ordre un* wqo *pour plus de commodité.*

Exemple 2. $(\mathbb{N}, \leqslant)$ *est un* wqo.

Exemple 3. $(P, |)$ *avec* P *l'ensemble des nombres premiers et* $|$ *la division* ($a|b \Leftrightarrow a$ *divise* b)*, n'est pas un* wqo *car tous les éléments de* P *sont incomparables deux à deux, donc à fortiori il n'existe aucun couple d'indices* $i < j$ *tel que le*

i-ème nombre premier est inférieur (au sens de la divisibilité) au j-ème nombre premier.

Exemple 4. $(\mathbb{Z}, \leqslant)$ *n'est pas un* wqo*, car on peut obtenir une sous-suite strictement décroissante, et donc où il n'existe pas de couple d'indices* $i < j$ *tel que* $s_i \leqslant s_j$*.*

Une caractérisation équivalente d'un wqo est donné par la proposition suivante qui donne une autre caractérisation pour une bonne suite **infinie** :

Proposition 5. *Soit* $(S, \leqslant)$ *un* wqo*. Une suite infinie* $(s_i)_{i\in\mathbb{N}}$ *à valeurs dans* S *est une* bonne *suite si et seulement si elle contient une sous-suite infinie croissante.*

Démonstration. L'équivalence se décompose en les 2 implications suivantes :

— ($\Leftarrow$) trivial.

— ($\Rightarrow$) Soit $(s_i)_{i\in\mathbb{N}}$ une bonne suite infinie et considérons l'ensemble

$$E := \{i \in \mathbb{N} \mid \forall j > i,\ s_i \not\leqslant s_j\}.$$

Si E était infini $(s_i)_{i\in E}$ serait une mauvaise suite infinie, par définition de E, ce qui est impossible dans le wqo S. Donc E est fini et il existe $i_0 > \max(E)$. Comme $i_0 \notin E$, il existe également $i_1 > i_0$ tel que $s_{i_0} \leqslant s_{i_1}$ et ainsi de suite. On construit de cette manière une sous-suite infinie croissante.

□

Il existe un moyen très efficace de construire de nouveaux *wqo* à partir de *wqo* initiaux, comme le prouve le lemme suivant :

Lemme 1. *Soient* $(A, \leqslant_A)$ *et* $(B, \leqslant_B)$ *deux* wqo*. Alors* $(A \times B, \leqslant_{A\times B})$ *est un* wqo *en définissant*

$$(a, b) \leqslant_{A\times B} (a', b') \Leftrightarrow \begin{cases} a \leqslant_A a' \\ b \leqslant_B b' \end{cases}$$

Démonstration. On considère une suite infinie quelconque $x_1, x_2, x_3, \ldots$ sur $A \times B$ avec, pour tout i, $x_i = (a_i, b_i)$ et on va utiliser deux fois la proposition 5 :

Comme (a_i) est infinie d'un wqo, c'est une bonne suite et il existe $I \subset \mathbb{N}$ avec I infini tel qu'on peut extraire $(x_i)_{i\in I} = x_{i_1}, x_{i_2}, x_{i_3}, \ldots$ une sous-suite infinie de $(x_i)_{i\in\mathbb{N}}$ qui est croissante pour la première composante. Dans un deuxième temps comme la suite formé des éléments de la deuxième composante de cette sous-suite est une bonne suite (car B est un wqo et la sous-suite est infinie) on peut extraire à nouveau une sous-suite infinie de $(x_i)_{i\in I}$ qui est croissante pour cette deuxième composante. La sous-suite résultante de ces deux extractions successives est bien une sous-suite infinie croissante de la suite initiale $(x_i)_{i\in\mathbb{N}}$.

Ainsi, comme cette dernière était choisie arbitrairement, toute suite infinie à valeurs dans $A \times B$ est une bonne suite pour le pré-ordre $\leqslant_{A\times B}$ ce qui montre, selon la définition 1, que $(A \times B, \leqslant_{A\times B})$ est un *wqo*.

□

Lemme 2 (Lemme de Dickson)**.** $(\mathbb{N}^k, \leqslant_\times)$ *est un* wqo.

Démonstration. Ceci est une conséquence immédiate du lemme 1, par récurrence en utilisant $A = \mathbb{N}^{k-1}$ et $B = \mathbb{N}$.

□

1.2 Caractérisation supplémentaire des wqo

Une question que l'on peut se poser est de savoir si dans un pré-ordre qui n'est pas un wqo on ne pourrait pas avoir autre chose que des antichaînes infinies ou des suites infinies strictement décroissantes, par exemple une hybridation des deux. La réponse à cette question est négative comme l'indique la proposition suivante. Celle-ci nous éloigne quelque peu de notre objet d'étude initial mais fait partie des classiques de la théorie des wqo et de ce que nous avons choisi d'étudier durant ce TER.

Proposition 6. $(S, \leqslant)$ *est un* wqo *si et seulement si il n'existe ni d'antichaîne infinie (i.e. dont tous les éléments sont incomparables deux à deux), ni de sous-suite infinie strictement décroissante.*

Démonstration. On décompose naturellement l'équivalence en deux parties :

- ($\Rightarrow$) est trivial par définition d'un *wqo*.

- ($\Leftarrow$) : On suppose que $(S, \leqslant)$ n'est pas un wqo et on veut montrer qu'il existe une antichaîne infinie ou une suite strictement décroissante infinie. Soit $(s_i)_{i\in\mathbb{N}}$ une mauvaise suite infinie, c'est à dire $\forall i, \forall j, i < j \Rightarrow s_i \not\leqslant s_j$, et soit $E = \{i \in \mathbb{N} \mid \forall j > i,\ s_i \text{ n'est pas comparable avec } s_j\}$.

  - Si E est infini, alors $(s_i)_{i\in E}$ est une antichaîne infinie.
  - Si E est finie, on prend $i_0 > max(E)$ comme $i_0 \notin E$ il existe $i_1 > i_0$ tel que s_{i_0} et s_{i_1} sont comparables et comme $s_{i_0} \not\leqslant s_{i_1}$ on a forcément $s_{i_0} > s_{i_1}$. De même on peut trouver $i_2 > i_1$ tel que l'on ait $s_{i_0} \not\leqslant s_{i_1} \not\leqslant s_{i_2}$ (car $i_1 > max(E)$ aussi) et en continuant ainsi de suite on obtient une suite infinie strictement décroissante.

□

Une preuve plus graphique de ($\Leftarrow$) peut être donné par la théorie de Ramsey :

Preuve avec le théorème de Ramsey

On rappelle que pour un graphe non orienté $G = (V, E)$ et $W \subseteq V$ le graphe H obtenu en conservant uniquement les sommets de W et les arêtes reliant ces sommets est appelé graphe *induit* de G par W. On appelle *clique* de G un graphe complet induit de G.

Quand on colorie les arêtes d'un graphe G, on associe une couleur à chacune des arêtes de E. Une clique monochromatique est une clique dont les arêtes sont toutes coloriées avec la même couleur.

On rappelle également que, par le **principe des tiroirs**, si on colorie un ensemble infini d'objets avec un nombre fini de couleurs alors il existe un sous-ensemble infini d'objets coloriés avec la même couleur. Enfin un graphe *dénombrable* est un graphe dont l'ensemble des sommets est dénombrable.

Théorème 7. *Théorème de Ramsey (version simplifiée) :*
Si on colorie les arêtes d'un graphe complet infini dénombrable avec un nombre fini de couleurs alors il existe une clique infinie monochromatique.

Démonstration. On pose $G = (V, E)$ avec $V \simeq \mathbb{N}$ et G complet.

Soit $\mathcal{C}$ une coloration des arêtes E de G toujours par un nombre fini k de couleurs et soit v_0 un sommet arbitraire fixé. Cette coloration $\mathcal{C}$ des arêtes induit naturellement (par le biais de v_0) une coloration $\mathcal{C}'$ des **sommets** de $G \backslash v_0$, la $\mathcal{C}'$-couleur d'un sommet $v \neq v_0$ étant la $\mathcal{C}$-couleur de l'arête (v, v_0).

Comme $\mathcal{C}'$ est une coloration finie d'un ensemble infinie, il existe $W_0 \subset V \backslash \{v_0\}$ tels que les sommets de W_0 soient de la même $\mathcal{C}'$-couleur et ce avec W_0 infini. En revenant à la coloration initiale $\mathcal{C}$ des **arêtes**, W_0 et v_0 définissent donc une étoile monochromatique infinie $K_{1,\omega}$ (centrée en v_0 et dont chaque extrémité est un sommet de W_0).

Maintenant on peut réitérer à l'infini ce raisonnement :

— prendre $v_i \in W_{i-1}$
— prendre $W_i \subset W_{i-1} \backslash \{v_i\}$ tel que l'étoile centrée en v_i et d'extrémités les sommets de W_i soit monochrome pour la coloration $\mathcal{C}$ des arêtes.

Par ce procédé on obtient une suite infini $(v_0, v_1, \ldots)$ telle que pour tout indice i fixé, les paires de sommets (v_i, v_j) sont reliées par un arc de même couleur, pour tout $j > i$. Concrètement chaque élément de cette suite infinie est le centre d'une étoile monochromatique dont les extrémités sont tous les éléments suivants dans la suite.

Par le **principe des tiroirs** il existe une sous-suite infinie $(v'_i)_{i \in \mathbb{N}}$ de $(v_i)_{i \in \mathbb{N}}$ telle que les étoiles de (v'_i) soient toutes d'une même couleur. Alors le graphe induit par cet ensemble de sommets forme une clique infinie monochromatique. □

On peut à présent prouver l'affirmation suivante : Dans un ensemble muni d'un *quasi ordre* $(S, \leqslant)$, si il n'y a ni antichaîne infinie ni suite strictement décroissante infinie, alors $S, \leqslant)$ est un *wqo*.

En effet, supposons que $(S, \leqslant)$ n'est pas un wqo. Alors il existe $(s_i)_{i \in \mathbb{N}}$ une suite infinie mauvaise d'éléments de S. Il suffit alors de voir ses éléments comme des sommets et pour toute paire d'indice (i, j) avec $i < j$ de mettre une arête entre s_i et s_j et de la colorier de la façon suivante :

— si $s_i \leqslant s_j$ alors mettre une arête verte.
— si $s_i > s_j$ alors mettre une arête rouge.
— si s_i et s_j sont incomparables, alors mettre une arête noire.

Le théorème de Ramsey affirme qu'il existe nécessairement une clique infinie monochromatique verte, rouge ou noire. En fait il n'y a aucune arête verte sinon $(s_i)_{i \in \mathbb{N}}$ serait une bonne suite. Donc on en déduit qu'il existe nécessairement une clique infinie rouge ou une clique infinie noire c'est à dire une suite infinie strictement décroissante ou une antichaîne infinie.

Chapitre 2

Systèmes de transitions bien structurés (WSTS)

Maintenant que nous avons expliqué la théorie sous-jacente des wqo, nous allons pouvoir définir les objets centraux de ce papier : les *Systèmes de transitions bien structurés*, que l'on appelle *WSTS* dans la suite. Mais avant, il nous reste à définir plus formellement la notion de système de transitions ainsi que la propriété de monotonicité qui caractérise les WSTS. En deuxième partie de chapitre nous présenterons enfin quelques exemples complets de systèmes bien structurés.

2.1 Systèmes de transitions et monotonicité

2.1.1 Systèmes de transitions

Définition 8. *On appelle* système de transition $\mathcal{T} = (S, \rightarrow)$ *un couple, avec* S *un ensemble de configurations (qu'on appellera aussi états), et* $\rightarrow$ *une relation de transition (formellement on a* $\rightarrow \subseteq S \times S$*).*

Exemple 9. *On peut imaginer un programme "compteur" qui ne contient qu'une variable* x*.* x *part de 0 et à tout moment il effectue une des 2 opérations suivantes :*

$$x = x + 1 \text{ ou, si } x > 0, x = x - 1$$

La sémantique de ce programme peut alors être représentée par :

$$0 \leftrightarrow 1 \leftrightarrow 2 \leftrightarrow 3 \ldots$$

Dans la suite nous nous limiterons à étudier des systèmes de transitions avec S dénombrable. On définit un calcul démarrant à s_1 et finissant par s_t de la manière suivante : c'est une suite de configurations $s_1, s_2, \ldots, s_t$ telle que

$s_1 \to s_2 \to \ldots \to s_t$. De la même manière qu'il nous arrivera parfois d'intervertir état et configuration, dans la suite nous emploierons également parfois le terme d'exécution à la place de calcul.

Les systèmes de transitions que nous allons étudier plus en détail sont des systèmes *ordonné* c'est-à-dire muni d'une relation de quasi-ordre i.e. une relation *réflexive* et *transitive* $\leqslant \subseteq S \times S$. On abrégera le terme quasi-ordre par *qo*, (conformément aux notations dans l'article [Schmitz and Schnoebelen, 2014]).

2.1.2 Monotonicité

Définition 10. *Un système de transitions quasi-ordonné* $S = (S, \to, \leqslant)$ *est* monotone *si pour tout* $s_1, s_2, t_1 \in S$ *on a l'implication suivante :*

Si $(s_1 \to s_2$ *et* $s_1 \leqslant t_1)$ ***alors*** $\exists t_2 \in S$ *tel que* $(t_1 \to t_2$ *et* $s_2 \leqslant t_2)$

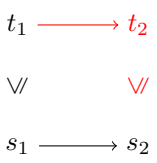

FIGURE 2.1 – Représentation graphique de la monotonicité.

Exemple 11. *L'exemple 9 du compteur est monotone car pour tout* $s_i, s_j \in S$ *si* $s_i \leqslant s_j$ *alors* $s_i + 1 \leqslant s_j + 1$.

On appelle parfois cette propriété particulière la *compatibilité par le haut* (pour plus de précision). Il arrive également que la relation de transition soit qualifiée de *croissante*.

Le lemme suivant se révélera fondamental pour prouver la décidabilité de Terminaison.

Lemme 3 (Simulation par au dessus)**.** *Soit* $\mathcal{S}$ *un système de transitions quasi-ordonné monotone avec une exécution* $c = s_0 \to \ldots \to s_n$ *et une configuration* t_0 *telle que* $s_0 \leqslant t_0$*. Alors il existe une exécution* $d = t_0 \to \ldots \to t_n$ *(de même longueur) telle que* $\forall i \in [0, n], s_i \leqslant t_i$*. On dit alors que* d ***simule*** c *(par au dessus).*

Démonstration. La construction de proche en proche de d est immédiate d'après la monotonicité. □

2.1.3 Définition de WSTS

À présent que nous avons défini les wqo et la propriété de monotonicité, nous allons pouvoir introduire la définition formelle de WSTS qui s'appuie ces notions.

Définition 12. *Un* système de transition quasi-ordonné $\mathcal{S} = (S, \to, \leqslant)$ *est un* WSTS *si*

— $(S, \leqslant)$ *est un wqo.*

— $(S, \leqslant)$ *est monotone (voir Définition 10).*

En reprenant l'exemple 9 du compteur sur $\mathbb{N}$, on peut voir sans effort que ce système de transition appartient à la grande famille des WSTS.

2.2 Exemples de WSTS

Afin de mieux comprendre comment fonctionnent les WSTS, nous allons maintenant introduire plusieurs exemples qu'on rencontre fréquemment dans la littérature.

2.2.1 VAS (Vector Addition Systems)

Le VAS (ou système d'addition de vecteur) est un système de transition qui fonctionne de la façon suivante : Une configuration est représentée par un vecteur de dimension n, $(i_1, i_2, \ldots, i_n)$ avec chaque composante i un entier naturel. Chaque transition est également un vecteur de taille n, où chaque composante est un entier relatif.

On commence dans une configuration initial, et on change de configuration en effectuant la somme d'un vecteur transition et d'un vecteur d'état, sachant qu'un état du système doit toujours avoir toutes ses composantes positives ou nulles.

Le système s'arrête lorsqu'on ne peut plus effectuer aucune transition sans violer cette règle.

Il existe une relation d'ordre naturelle pour ce modèle de calcul :
Soit $u = (u_1, u_2, \ldots, u_n)$ et $v = (v_1, v_2, \ldots, v_n)$. $u \leqslant v$ lorsqu'on a $u_i \leqslant v_i$ pour tout $0 \leqslant i \leqslant n$.

Exemple 13. *Syntaxe : vecteur initial : (3,3) transitions : (1,2) (-1,2) (2,-3)*
Sémantique :

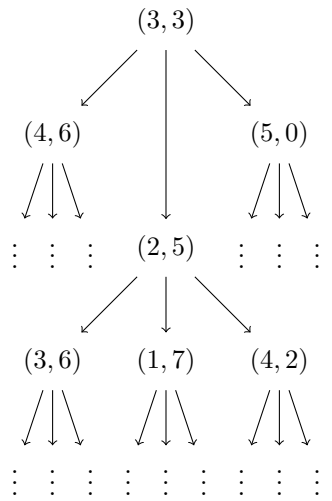

FIGURE 2.2 – Exemple de VAS de dimension 2.

Lemme 4. *VAS est un WSTS.*

Démonstration. d'après 2, $(\mathbb{N}^k, \leqslant_\times)$ est un *wqo*, il ne reste donc qu'à démontrer que le système de transitions est monotone. Soit $s = (s_1, s_2, ..., s_n)$, $t = (t_1, t_2, ..., t_n)$ deux configuration telle que $s \leqslant t$, et une transition $v = (v_1, v_2, ..., v_n)$,. On applique la transition v sur s et t, on a alors $s' = (s_1+v_1, s_2+v_2, ..., s_n+v_n)$ et $t' = (t_1 + v_1, t_2 + v_2, ..., t_n + v_n)$. Or on travaille sur des entiers, donc pour tout $0 \leqslant i \leqslant n$, vu que $s_i \leqslant t_i$, on a $s_i + v_i \leqslant t_i + v_i$, donc le système de transitions et *monotone* et le VAS est un *WSTS* □

2.2.2 Réseaux de Petri

Les réseau de Petri sont essentiellement des VAS et peuvent être vus comme tels. La manière dont leur syntaxe est représenté apportent cependant un éclairage nouveau et permet de voir pourquoi ce type de WSTS est utilisé pour modéliser toute sorte de protocoles de communication ainsi que des programmes accédant à des ressources de façon concurrente.

Formellement la syntaxe d'un réseau de Petri est la suivante. Un réseau $N = \{P, T, F\}$ a un ensemble fini *d'états* P, n ensemble fini de *transition* T, et une *matrice d'écoulement* $F : (P \times T \cup T \times P) \to \mathbb{N}$. Une configuration est définie par les placements de jetons dans les états, par exemple dans la figure 2.3, la configuration se note $\{p_1, p_1, p_2, p_3\}$, ou pour simplifier, $\{p_1^2, p_2, p_3\}$.

Pour changer de configuration en passant par une transition t_i, il faut que tout état entrant en t_i ait au moins un jetons, et si on l'applique, alors tout état sortant de t_i obtient un jeton supplémentaire. Par exemple sur la figure 2.3 on peut obtenir $\{p_1, p_2, p_3^3\}$ en passant par t_1, ou bien $\{p_1^2, p_2, p_4\}$ en passant par t_2, mais on ne peut pas passer par t_3 car il n'y a pas de jetons dans l'état p_4.

Ce système est équivalent au VAS, en effet, en ordonnant les emplacements, on peut par exemple noter la configuration $\{p_1^2, p_2, p_4\}$, $(2, 1, 0, 1)$, et une transition en vecteur de dimension $n = nombred'etats$ dans $\{-1, 0, 1\}^n$(par exemple la transition t_1 de l'exemple 14 peut se noter $(-1, -1, 1, 0)$)

L'ordre naturel pour ce système de transition est l'inclusion, de même que pour le *VAS* (2.2.3), où $M \subseteq M'$ si chaque état du réseau a au moins autant de jetons. Par exemple $\{p_1, p_2, p_3\} \subseteq \{p_1^2, p_2, p_3\}$ mais $\{p_1, p_2, p_4\} \not\subseteq \{p_1^2, p_2, p_3\}$.

Exemple 14. *Configuration actuelle :* $(2, 1, 1, 0)$ *(ou* $\{p_1^2, p_2, p_3\}$*)*
Transitions :
$t_1 = (-1, -1, 1, 0)$
$t_2 = (1, 0, -1, 1)$
$t_3 = (0, 1, 1, -1)$

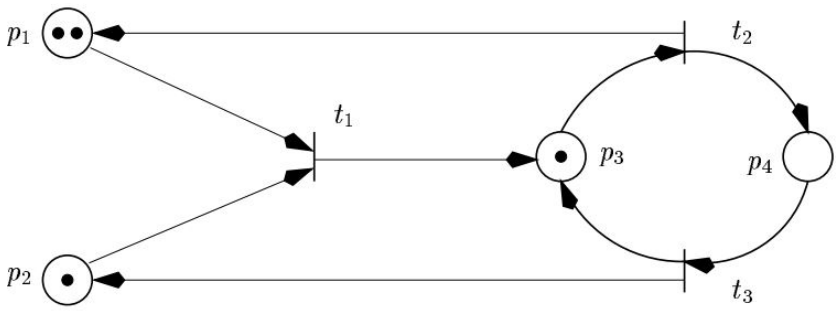

FIGURE 2.3 – Exemple de reseau de petri

Lemme 5. *Réseau de Petri est un WSTS.*

Démonstration. Vu que Q est fini, $(\mathbb{N}^Q, \subseteq)$ est un *wqo* d'après 2. Montrons que le systeme est *monotone*.

Soit s_1 et c_1 deux configuration avec $s_1 \subseteq c_1$. On applique un transition t_i sur s_1, on obtient s_2. Comme $s_1 \subseteq c_1$, on peut appliquer t_i sur c_1 pour obtenir c_2. Pour chaque transition on n'ajoute(ou retire) un jeton exactement au mêmes états, ce qui fait que chaque état de c_2 a autant ou plus de jetons que celui de s_2. On a donc $s_2 \subseteq c_2$ et le système est *monotone*. □

2.2.3 Broatcast protocols

Les protocoles de Broadcast sont d'autres modèles de calculs utilisés pour modéliser des protocoles de communication.

Syntaxe :

Un protocole de Broadcast est défini par un triplet $B = (Q, M, R)$, où
— Q est un ensemble fini d'*emplacements*.
— M est un ensemble de *messages*.

— R un ensemble fini de *règles* qui sont dans le cas présent des triplets (q, op, q') dans $Q \times Op \times Q$ qui décrivent des *opérations* possibles depuis q vers q'. Ces opérations sont de plusieurs sortes :

rendez-vous Notés $r!$ et $r?$, ce sont respectivement les opérations d'envoi et de réception d'un message r.

broadcast Notés $b!!$ et $b??$, ce sont respectivement les opérations d'envoi et de réception d'un message de rendez-vous b.

spawn Noté sp(p), c'est l'opération de création d'un nouveau processus qui va s'exécuter à l'emplacement p.

Bien évidemment on note $q \xrightarrow{op} q'$ si $(q, op, q') \in R$.

Sur le figure 2.4 on peut voir un exemple de protocole de Broadcast.

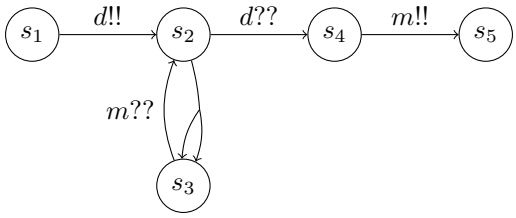

FIGURE 2.4 – Exemple de broadcast protocol.

Exemple 15. *Cet exemple est tiré de l'article [Schmitz and Schnoebelen, 2014].*

La définition formelle du protocole de Broadcast contraint le système à avoir exactement une transition $\xrightarrow{m?}$ pour toute transition $\xrightarrow{m!}$. On peut éventuellement introduire des règles de la forme $q \xrightarrow{m} q'$ qui permettent, à la réception d'un message m, d'envoyer 1 processus de q vers q'. En tout cas toute lettre étiquetant une opération ne doit servir que pour un seul type d'opération.

Sémantique :

On exprime la sémantique d'un protocole de Broadcast par un système de transition $\mathcal{S} = (S, \to)$ où les états du systèmes sont appelés configurations ou également marquages et sont des multi-ensembles d'emplacements de Q.

Une configuration s indique l'emplacement (ou l'état) de chaque processus. $s = \{q_1, ..., q_n\}$, où plusieurs processus peuvent être dans les mêmes états ce qui fait donc de chaque configuration un multi-set. Pour plus de simplicité, lorsqu'on a une configuration de la forme $s = \{q, q, q, q', q', q', q'\}$ par exemple, on la notera $s = \{q^3, q'^4\}$.

Il y a différentes façons de changer de configuration :

— *rendez-vous* : si $q_1 \xrightarrow{m!} q_1'$ et $q_2 \xrightarrow{m?} q_2'$ avec $m \in M$, alors $s + \{q_1, q_2\} \to s + \{q_1', q_2'\}$ pour tout s dans $\mathbb{N}^Q$

— *broadcast* : si $q_0 \xrightarrow{m!!} q_0'$ et $q_i \xrightarrow{m??} q_i'$ pour tout $1 \leqslant i \leqslant k$, alors $s + \{q_0, q_1, \ldots, q_k\} \to s + \{q_0', q_1', \ldots, q_k'\}$ pour tout s dans $\mathbb{N}^Q$.

— *spawn*(création de nouveau processus) : si $q \xrightarrow{sp(p)} q'$, alors $s + \{q\} \to s + \{q', p\}$ pour tout s dans $\mathbb{N}^Q$.

Exemple 16. *On peut par exemple obtenir, sur l'exemple de la figure 2.4, la suite de configurations suivante :*

$$\{s_1, s_2^2, s_4\} \xrightarrow{s_3} \{s_1, s_2, s_3^2, s_4\} \xrightarrow{s_3} \{s_1, s_3^4, s_4\} \xrightarrow{m} \{s_1, s_2^4, s_5\} \xrightarrow{d} \{s_2, s_4^4, s_5\}$$

En ordonnant linéairement les emplacements (ce qui a été fait implicitement) on peut aussi noter la chaîne précédente de cette manière :

$$(1,2,0,1,0) \xrightarrow{s_3} (1,1,2,1,0) \xrightarrow{s_3} (1,0,4,1,0) \xrightarrow{m} (1,4,0,0,1) \xrightarrow{d} (0,1,0,4,1)$$

Proposition 17. *Les protocoles de broadcast sont des WSTS.*

Démonstration. Il s'agit de montrer que tout protocole de Broadcast est muni d'un wqo et que le système de transition a la propriété de compatibilité par le haut. Un ordre naturel pour ce système est l'inclusion :

$$c \subseteq c' \overset{\text{def}}{\Leftrightarrow} \forall q \in Q, c(q) \leqslant c'(q)$$

Par exemple, $\{q, q, q'\} \subseteq \{q, q, q, q'\}$ mais $\{q, q', q'\} \not\subseteq \{q, q, q'\}$ (avec $q \neq q'$). $(\mathbb{N}^Q, \subseteq)$ est un *wqo* isomorphe à $(\mathbb{N}^k, \leqslant_\times)$ avec $k = \sharp Q$ fini, d'après le lemme de Dickson (voir lemme 2). Il ne reste donc qu'à démontrer que le système est *monotone*.

Soit une transition $c_1 \to c_2$ et $c_1 \subseteq d_1$. Sans perte de généralité posons $d_1 = c_1 + \{q\}$. Il y a trois cas possibles pour le type de la transition $c_1 \to c_2$:

1. Soit $c_1 \to c_2$ est une transition *rendez-vous* avec, par exemple :
   — $q_1 \xrightarrow{m!} q_1'$
   — $q_2 \xrightarrow{m?} q_2'$
   auquel cas $c_2 = c_1 - \{q_1, q_2\} + \{q_1', q_2'\}$ et en prenant $d_2 := d_1 - \{q_1, q_2\} + \{q_1', q_2'\}$ alors $d_1 \to d_2$ est aussi une transition *rendez-vous*.

2. Soit $c_1 \to c_2$ est une transition *spawn* avec par exemple :
   —
   $$q \xrightarrow{sp(p)} q'$$
   auquel cas $c_2 = c_1 - \{q\} + \{q', p\}$ et en prenant $d_2 := d_1 - \{q\} + \{q', p\}$ on peut faire une relation *spawn* de $d_1 \to d_2$.

3. Soit $c_1 \to c_2$ est une transition *broadcast* avec par exemple :
   — $q_0 \xrightarrow{m!!} q_0'$

— $q_i \xrightarrow{m??} q'_i$ pour tout $1 \leqslant i \leqslant k$
auquel cas $c_2 = c_1 - \{q_0, \ldots, q_k\} + \{q'_0, \ldots, q'_k\}$ et en prenant $d_2 := d_1 - \{q_0, \ldots, q_k\} + \{q'_0, \ldots, q'_k\}$ alors $d_1 \to d_2$ est aussi une transition *rendez-vous*.

Dans les trois cas énoncés ci-dessus on a bien que $c_2 \subseteq d_2$.

□

Deuxième partie

Décidabilité de Terminaison et Coverability

Chapitre 3

Le problème Terminaison

Comme annoncé plus haut, nous allons maintenant nous intéresser de plus près au problème Terminaison et voir pourquoi il est en fait décidable avec un WSTS. On rappelle l'énoncé du problème :

Définition 18. Terminaison *:*
ENTRÉE : Un système de transition et une configuration d'entrée s
PROBLÈME : Toute exécution depuis s termine-t-elle (i.e. est-elle de longueur finie) ?

Ce problème n'est pas à confondre avec le suivant plus général appelé $Terminaison\ structurelle$ ou encore $Terminaison\ uniforme$:

Définition 19. *Étant donné un système de transition, toute exécution du modèle de calcul correspondant termine-t-elle ?*

$Terminaison\ structurelle$ a lui été prouvé indécidable et nous ne l'évoquerons plus dans la suite.

3.1 Décidabilité de Terminaison

Dans le but de mettre en évidence la décidabilité de Terminaison nous devons à présent introduire le lemme suivant.

Lemme 6 (temoin fini de non terminaison)**.** *Soit $\mathcal{S} = (S, \leqslant, \rightarrow)$ un WSTS et $s_0 \in S$.*
Il existe une exécution infinie de $\mathcal{S}$ depuis s_0 si et seulement si il existe une exécution finie de $\mathcal{S}$ depuis s_0 qui est une bonne suite.

Démonstration. 2 sens à prouver :

— ($\Rightarrow$) Toute exécution infinie $s_0 \rightarrow s_1 \rightarrow \ldots$ forme une bonne suite car l'ordre est un wqo, donc il existe $i, j \in \mathbb{N}$ tel que $s_i \leqslant s_j$ et alors $s_0 \rightarrow ... \rightarrow s_j$ forme une bonne suite **finie**.

— ($\Leftarrow$) Soit une bonne exécution finie $s_0 \to \ldots \to s_i \to \ldots \to s_j$ avec $s_i \leqslant s_j$. D'après la monotonicité de $\mathcal{S}$ et plus précisément le lemme 3 il existe une exécution $s_j \to \ldots \to s_{2j-i}$ qui simule par au dessus $s_i \to \ldots \to s_j$. Mais alors $s_0 \to \ldots \to s_j \to \ldots \to s_{2j-i}$, obtenu en accolant ces 2 exécutions, forme une exécution plus longue et on a $s_i \leqslant s_j \leqslant s_{2j-i}$ donc on peut encore ralonger cette nouvelle exécution en accolant en suffixe au choix soit une exécution simulant $s_j \to \ldots \to s_{2j-i}$ soit même une exécution simulant $s_i \to \ldots \to s_{2j-i}$. Ainsi de suite on peut toujours rajouter un suffixe à l'exécution finie de départ donnant naissance à une exécution infinie.

□

Ainsi l'existence d'une bonne suite finie est un témoin de non terminaison et cela montre tout presque immédiatement que Terminaison est un problème décidable.

Théorème 20. *Terminaison est décidable pour les WSTS.*

Pour aborder rigoureusement la preuve de ce résultat, nous allons encore puiser dans la théorie des graphes élémentaire :

Un nœud est dit $infini$ s'il a un nombre infini de successeurs. Un arbre est dit *à branchement fini* s'il ne contient aucun noeud infini.

Lemme 7 (Lemme de König)**.** *Tout arbre infini à branchement fini a une branche infinie.*

Démonstration. Supposons qu'on ait un arbre infini de racine r_0 à branchement fini. Comme r_0 n'a qu'un nombre fini de successeurs immédiats, l'un d'entre eux au moins, notons le r_1, est la racine d'un sous-arbre infini (sinon l'arbre ne serait pas infini). De même l'un des successeurs immédiat de r_1, notons le r_2, est racine d'un sous-arbre infini. On peut définir ainsi un nombre infini de nœuds $r_i, \forall i \in \mathbb{N}$ qui forment une branche infinie. □

Preuve de la décidabilité de la terminaison. On parcourt l'arbre des configurations (en profondeur par exemple). il y a deux cas de figure :

1. L'arbre est fini.
2. Sinon il existe une branche infinie d'après le lemme de König (et l'hypothèse que l'arbre des configurations du système est à branchement fini).

— Terminaison est trivialement semi-décidable. En effet un semi-algorithme consiste à parcourir l'arbre des configurations et ce semi-algorithme termine si et seulement l'arbre est fini auquel cas la réponse à la question est positive : toute calcul depuis s est de longueur fini.

— D'autre part le problème complémentaire, à savoir s'il existe un calcul commençant par s et ne terminant jamais, est également semi-décidable d'après le lemme 6 : il suffit de rechercher les bonnes suites parmi les

exécutions possibles. Dès qu'on trouve une exécution $s \to \ldots \to s_i \to \ldots \to s_j$ avec $s_i \leqslant s_j$ on s'arrête dans l'état acceptant : il existe une exécution commençant par s et de longueur infini. En effet, grâce à la monotonicité du WSTS, si on a $s_i \leqslant s_j$ et $s_i \to s_j$, alors il existe s_k tel que $s_j \to s_k$ et $s_j \leqslant s_k$, et ainsi de suite infiniment. De plus, on en peut pas avoir ni d'antichaine infini, ni de sous suite strictement décroissante infini, et donc si on ne tombe pas dans le cas précèdent, alors le sous-arbre est fini.

Vu que le problème de terminaison pour les WSTS est semi-décidable et que son inverse l'est aussi, alors le problème de terminaison est décidable sur les WSTS □

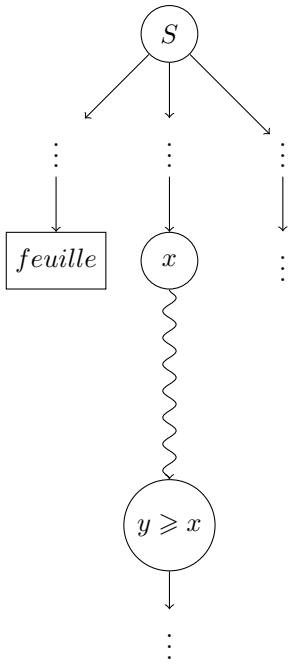

FIGURE 3.1 – Exemple de branche infini

Un algorithme pour la terminaison est donc le suivant :
On effectue un parcourt en profondeur du graphe des configuration depuis s. Si la configuration est calculé supérieure à une de ses ancêtres, alors on retourne que le WSTS en question ne termine pas depuis s. Sinon on a parcourut le graphe en entier (grâce à l'hypothèse de branchement fini) et on retourne naturellement que le système est terminable. Un parcours en largeur d'abord fonctionne tout autant mais gaspillerait de la mémoire pour rien étant donné qu'il faut comparer chaque nouvelle configuration possible par tous ses ancêtres dans l'arbre.

3.2 Exemples et implémentation

Afin de bien sentir comment fonctionne cet algorithme, observons comment il s'exécute sur quelques exemples simples qui rendent plus ou moins compte de la situation générale.

3.2.1 Exemples sur des VAS de dimension 2

Sur les deux exemples qui suivent, l'algorithme présenté dans la section précédente pour répondre au problème Terminaison répond respectivement "oui" et "non" :

Exemple 21. *vecteur initial : (2,2)*
transitions : (-2,2),(1,-3),(-1,-1)

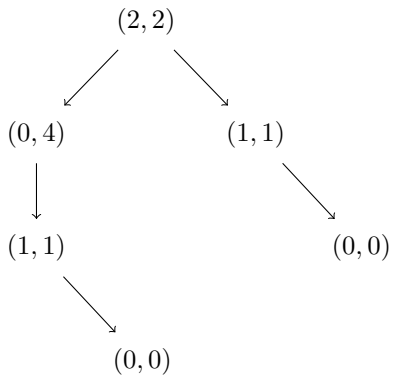

FIGURE 3.2 – Arbre des configurations avec que des branches finis

Dans cet exemple on peut déplier l'arbre des configurations car il est fini, donc le VAS de l'exemple 21 est terminable.

Exemple 22. *vecteur initial : (2,2)*
transitions : (-1,3),(1,-2),(-1,-1)

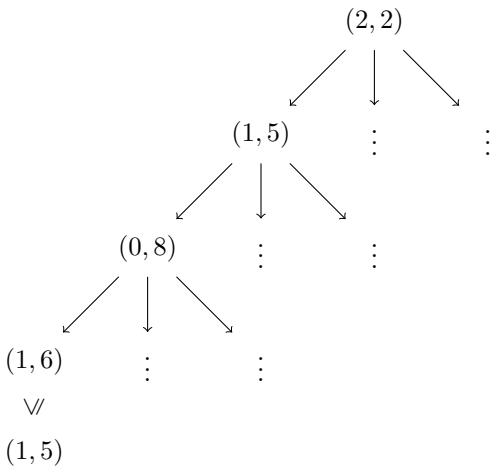

FIGURE 3.3 – Arbre des configurations avec une branche infini

Dans l'exemple précèdent on voit qu'on peut obtenir une configuration supérieur à une configuration précédente, donc on peut arrêter de parcourir le graphe des configuration, car on sait déjà que ce VAS n'est pas terminable.

3.2.2 Implémentation en Python

Voici le code en Python correspondant à l'algorithme qui décide le problème Terminaison d'un VAS :

Listing 3.1 – Code python d'un VAS de dimension 2

```
1  #entree : I etat initial I=[vecteur] ,
2  #L liste des transitions L = [[trans1],[trans2],...]
3
4  #exemple
5  I = [3,3]
6  L = [[-1,2],[2,-3]]
7
8  def vas2(P,L):
9      c=0            #numero de transition
10     tmp =[0,0]     #vecteur de travail
11     while c < len(L):
12         #test si positif
13         if(P[-1][0] + L[c][0] >= 0 ):
14             if(P[-1][1] + L[c][1] >= 0):
15                 tmp[0] = P[-1][0] + L[c][0]
16                 tmp[1] = P[-1][1] + L[c][1]
17                 #ajout a la pile des ancetre
```

```
18                  P = P + [tmp]
19                  #test si "plus grand" qu un ancetre
20                  for i in range(len(P) - 1):
21                      if(P[i][0] <= tmp[0]):
22                          if(P[i][1] <= tmp[1]):
23                              print(P)
24                              return False
25                  #si on sait que non terminable
26                  #on stop l'algorithme
27                  if(vas2(P,L) == False):
28                      return False
29                  #sinon on essaye avec une autre transition
30                  P.pop()
31          c += 1
32      return True
33
34  def main(I,L):
35      P=[I]  #Pile des ancetres
36      if(vas2(P,L)):
37          print(P)
38          print("le systeme est terminable")
39          return
40      print("le systeme n'est pas terminable")
41      return
```

Le programme utilise le parcourt en profondeur afin de ne conserver en mémoire que les ancêtres du sommet actuel. Tout d'abord, on commence par enregistrer le vecteur initial dans P, la pile des ancêtre. Ensuite, la fonction $vas2()$ va vérifier si on peut appliquer la première transition sans briser les règles du VAS. Si on peut l'appliquer, alors on l'ajoute a la pile et on vérifie si la nouvelle configuration est supérieur à une précédente dans la pile(si c'est le cas on arrête l'algorithme et on retourne "False"), puis on relance $vas2()$ avec la nouvelle pile. Sinon on essaye une autre transition, et si il n'y a plus de transition possible, on retire la dernière configuration de la pile. Lorsque la pile est vide, on a terminé le parcourt de l'arbre, et donc le VAS est terminable, on retourne "True".

Listing 3.2 – Application du programme sur l'exemple 21

```
1  >>> import vas2
2  >>> I = [2,2]
3  >>> L = [[-2,2],[1,-3],[-1,-1]]
4  >>> vas2.main(I,L)
5  le systeme est terminable
```

Listing 3.3 – Application du programme sur l'exemple 22

```
1  >>> import vas2
```

```
>>> I = [2,2]
>>> L = [[-1,3],[1,-2],[-1,-1]]
>>> vas2.main(I, L)
[[2, 2], [1, 5], [0, 8], [1, 6]]
"le systeme n'est pas terminable"
```

Chapitre 4

Le problème Couverture

Voici maintenant le deuxième problème qui intervient frequemment quand on traite des WSTS : le problème *Couverture*.

Etant donné un système de transition ordonné $\mathcal{S}$ ainsi qu'un état de départ $x_{init} \in S$ et un état cible $t \in S$, la question est de savoir s'il existe une exécution depuis x_{init} qui à un certain moment couvre t, i.e. s'il existe x_n atteignable depuis x_{init} avec $x \geq t$. On appelle une telle exécution $x_0 \to x_1 \to \cdots \to x_n$ avec $x_n \geq t$ une *exécution couvrante* (pour l'état t).

Dans la suite plutôt que de manipuler directement des exécutions couvrantes on va utiliser des objets appelés *pseudo-exécutions*. Une pseudo-exécution est une suite $s_0, \ldots, s_n$ telle que pour tout $i = 1, \ldots, n$, l'état s_{i-1} couvre s_i **en un seul pas de calcul**, c'est à dire que $s_{i-1} \to t_i$ pour un certain $t_i \geq s_i$. Il est clair que toute exécution est également une pseudo-exécution mais qu'il n'y a pas forcément l'implication inverse. Cependant l'intérêt d'une pseudo-exécution $s_0, \ldots, s_n$ avec $s_n \geq t$ est qu'elle témoigne de l'existence d'exécutions couvrantes depuis n'importe quel $s_{init} \geq s_0$:

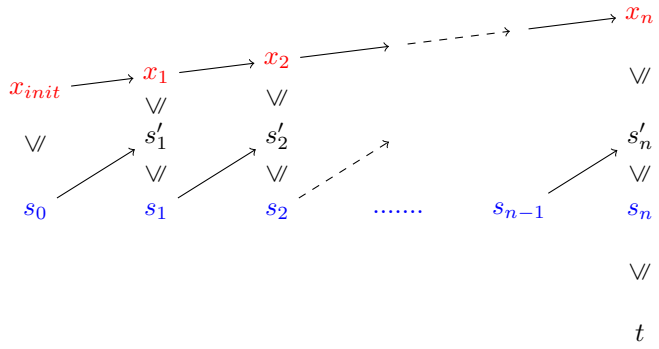

FIGURE 4.1 – Pseudo-exécution couvrante ⇒ Exécution couvrante.

Une pseudo-exécution $s_0, \ldots, s_n$ est dite *minimal* si pour tout $i = 1, \ldots, n$, l'état s_{i-1} est minimal parmi tous les configurations depuis lesquelles s_i peut être couverte en une étape (on dit que s_{i-1} est un pseudo-prédécesseur *minimal* de s_i).

Dernier point de vocabulaire : une suite $s_0, s_1, \ldots, s_n$ telle que la suite *inverse* $s_n, s_{n-1}, \ldots, s_0$ est mauvaise est une suite qualifiée de "revbad" (pour "bad reverse sequence").

Lemme 8 (témoin minimal pour Couverture)**.** *Si un système de transition ordonné bien structuré (WSTS) $\mathcal{S}$ admet une exécution couvrante (pour t) depuis s_{init}, il admet en particulier une pseudo-exécution minimal $s_0, \ldots, s_n$ avec*

- $s_0 \leq s_{init}$
- $s_n = t$
- $s_0, \ldots, s_n$ *revbad*

Démonstration. Supposons que $s_{init} = s_0 \to s_1 \to \cdots \to s_n$ est une exécution couvrante. Si on prend t au lieu de s_n on obtient clairement une pseudo-exécution terminant par t. Dans la suite on va montrer que si cette pseudo-exécution n'est pas minimal ou non revbad alors il existe une pseudo-exécution plus petite dans deux sens différents : plus courte ou plus "basse".

- Supposons que $s_n, s_{n-1}, \ldots, s_0$ n'est pas une suite mauvaise c'est à dire qu'il existe $0 \leq i < j \leq n$ avec $s_i \geq s_j$. Alors la suite $s_0, s_1, \ldots, s_{i-1}, s_j, s_{j+1}, \ldots, s_n$ est aussi une pseudo-exécution, plus courte.

- Supposons $s_0, \ldots, s_n$ n'est pas minimal il existe un certain s_{i-1} qui n'est pas minimal parmi les pseudo-prédecesseurs de s_i. On peut alors le remplacer par un autre pseudo-prédecesseur $s'_{i-1} \leq s_{i-1}$ minimal, ceci d'après la remarque suivante :

  Remarque 23. *Si $(S, \leq)$ est un wqo, n'importe quel sous-ensemble non vide de $\mathcal{S}$ contient au moins un élément minimal.*

  Alors $s_0, \ldots, s_{i-2}, s'_{i-1}, s_i, \ldots, s_n$ est encore une pseudo-exécution.

On peut répéter ce genre de raccourcissements et d'abaissements jusqu'à obtenir une pseudo-exécution qui est à la fois minimal et revbad (en plus de satisfaire les conditions $s_0 \leq s_{init}$ and $s_n = t$)

□

A présent, pour transformer le lemme 8 en un algorithme pour répondre à *Couverture*, on peut faire un raisonnement très similaire à celui fait pour le problème *Terminaison* : En appelant $MinPPre$ la fonction qui à un état donné associe l'ensemble de ses pseudo-prédécesseurs minimaux et en supposant que

cette fonction est calculable (en plus de l'hypothèse sur la calculabilité de l'ordre) on peut exhiber cet algorithme.

Lemme 9. *En posant $s \equiv s' \Leftrightarrow s \leq s' \leq s$, pour n'importe quel sous-ensemble d'un WQO il n'y a qu'un nombre fini d'éléments minimaux à cette équivalence près.*

A cause du lemme de Kőnig (voir lemme 7), l'ensemble de tous les pseudo-exécutions minimales et revbad **finissant par** t est fini : D'une part la minimalité des pseudo-exécutions assure, grâce au lemme 9, le branchement fini de l'arbre. D'autre part le caractère revbad assure que les branches ont toutes une longueur finie.

On peut donc construire progressivement cet ensemble de pseudo-exécutions en partant de la fin (depuis t) et en appliquant la fonction $MinPPre$ récursivement. En notant $MinPPre^*(t)$ l'ensemble de tous les configurations obtenus, on peut chercher progressivement s_{init} est plus grand que l'un d'entre eux.

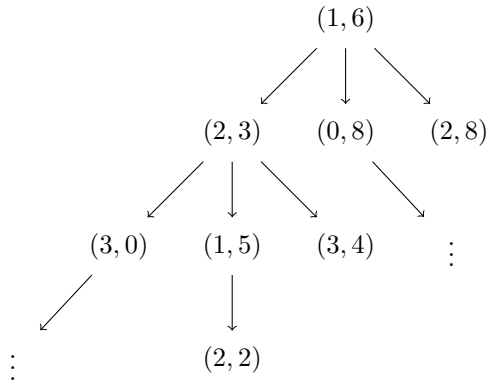

FIGURE 4.2 – Exemple de recherche d'exécution à partir de $(2, 2)$ couvrant $(1, 6)$ sur l'exemple 22

On voit qu'on réussi a obtenir une configuration inférieur ou égale $(2, 2)$ (ici elle est égale),donc on a trouver une exécution de $(2, 2)$ qui couvre $(1, 6)$ en appliquant les mêmes transitions que $(2, 2) \to (1, 5) \to (2, 3) \to (1, 6)$.

Conclusion

La recherche dans le domaine des WSTS est encore très récente. Finkel [Finkel, 1990] a été l'un des premiers à donner la définition actuelle de WSTS, en particulier en extrapolant à partir de réseaux de Petri. Puis des exemples de plus en plus élaborés et des algorithmes ont été développé pour répondre à des questions spécifiques. L'algorithme pour Terminaison est notamment apparu suite à l'article fondateur de Karp et Miller ([Karp and Miller, 1969]).

Nous avons vu dans ce TER que les WSTS forme une famille générique de systèmes de transitions pour lesquelles les problèmes Terminaison est décidable lorsqu'il est restreint à ces modèles de calcul. Nous avons également exhibé un algorithme générique pour répondre au problème Couverture. En fait on peut également s'intéresser à d'autres problèmes importants en vérification logicielle et plus ou moins faciles à traiter comme *Reachability*, *Boundedness* et *Place-Boudedness* pour citer les plus classiques.

On peut également s'interroger sur la complexité des algorithmes pour Terminaison et Couverture. En réalité même si ces problèmes sont décidables pour les WSTS, il est montré dans notre article de référence ([Schmitz and Schnoebelen, 2014]) que la complexité dans le pire des cas et le meilleur des cas de ces algorithmes est énorme si bien qu'en pratique il semble possible de résoudre ces problèmes uniquement sur de petites entrées (des VAS de faible dimension par exemple). Plus précisément il est montré que la complexité de Terminaison et Couverture est au delà de la classe *élémentaire*, et que la complexité dans le meilleur des cas des algorithmes est sensiblement identique à celle dans le pire des cas.

Pour arriver à un tel résultat, les auteurs sont partis du constat suivant : on peut imaginer un WSTS dont l'exécution termine à tous les coups (peu importe la configuration d'entrée) et qui admet une exécution de longueur monstrueuse. Reprenons par exemple le protocole de Broadcast de la Figure 4.3, que nous avons reproduit ici pour plus de commodité.

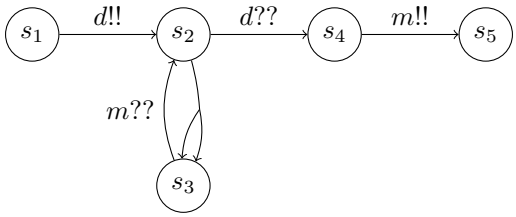

FIGURE 4.3 – Exemple de broadcast protocol.

Ce protocole de Broadcast admet une exécution de taille non élémentaire que nous pouvons construire de la façon suivante. En reprenant les notations de VAS "paramétré" du chapitre précédent ainsi qu'en ignorant le nombre de processus dans l'emplacement s_5 et enfin en abrégeant s_3 par s on a :

$$(0, n, 0, 1) \xrightarrow{s^n} (0, 0, 2n, 1) \xrightarrow{m} (0, 2n, 0, 0)$$

Une telle séquence de message double le nombre de processus en s_2 et supprime 1 processus de l'emplacement s_4. En itérant de telles séquences tant qu'il y a un processus en s_4 avant de "diffuser" le message de broadcast d cela donne donc la séquence :

$$(1, 2^0, 0, n) \xrightarrow{s^{2^0} m} (1, 2^1, 0, n-1) \xrightarrow{s^{2^1} m} (1, 2^2, 0, n-2) \to \dots$$

$$\dots \to (1, 2^{n-1}, 0, 1) \xrightarrow{s^{2^{n-1}} m} (1, 2^n, 0, 0) \xrightarrow{d} (0, 2^0, 0, 2^n)$$

qui implémente une exponentiation du nombre de processus en s_3 tout en supprimant 1 processus en s_1. En répétant ce genre de séquence on obtient donc la séquence suivante :

$$(n, 1, 0, 1) \to^* (0, 1, 0, \text{tower}(n))$$

où $tower()$ est la fonction définie de la manière suivante :

$$\text{tower}(0) \stackrel{\text{def}}{=} 1 \text{ et } \text{tower}(n+1) \stackrel{\text{def}}{=} 2^{\text{tower}(n)}$$

Donc même s'il termine toujours (ce qui reste à prouver) le broadcast protocol de Figure 4.3 peut engendrer une mauvaise exécution de longueur non élémentaire. Cela donne une borne inférieure à la complexité dans le meilleur des cas étant donné que l'algorithme pour Terminaison doit parcourir toutes les mauvaises exécutions.

On peut se demander comment situer plus finement la complexité des deux algorithmes de la section précédente. En réexaminant les arguments développés précédemment on peut observer que le point critique est la longueur de la plus longue **mauvaise suite**. En effet les arbres explorés dans le cas de la résolution des problèmes Terminaison et Couverture sont d'autant plus grand que les exécutions (ou pseudo-exécutions) mauvaises sont longues. La définition de wqo permet de dire que les mauvaises suites sont finis mais ne donne pas d'information sur leur longueur maximum. En observant le WSTS $(\mathbb{N}^Q, \subseteq)$ avec $Q = \{p, q\}$ et la suite suivante :

$$\{p^n\}, \{p^{n-1}\}, \ldots, \{p\}, \{q^m\}, \{q^{m-1}\}, \ldots, \{q\}, \emptyset \tag{4.1}$$

On voit qu'on peut facilement obtenir des mauvaises suites en utilisant deux genres d'astuces : partir d'une configuration de départ arbitrairement grande et à un moment dans la suite faire un "saut" arbitrairement grand.

Que ce genre de "sauts arbitrairement haut" apparaissent dans une branche de l'arbre fini construit par l'un des algorithmes précédents semble peu probable et on peut montrer formellement que ce n'est effectivement pas le cas en utilisant des fonctions contrôlant la longueur des mauvaises suites ainsi qu'un peu de théorie des ordinaux pour définir des classes de fonctions à croissance très rapides, et des classes de complexité correspondantes.

Remerciements

Nous tenons à remercier particulièrement M. Zeitoun pour ses explications claires et le temps important qu'il nous a accordé pour ce travail encadré de recherche.

Bibliographie

[Esparza et al., 1999] Esparza, J., Finkel, A., and Mayr, R. (1999). On the verification of broadcast protocols. In *Logic in Computer Science, 1999. Proceedings. 14th Symposium on*, pages 352–359.

[Finkel, 1990] Finkel, A. (1990). Reduction and covering of infinite reachability trees. *Information and Computation*, 89(2) :144 – 179.

[Finkel and Schoebelen, 2001] Finkel, A. and Schoebelen, P. (2001). Well-structured transition systems everywhere ! *Theoretical Computer Science*.

[Karp and Miller, 1969] Karp, R. M. and Miller, R. E. (1969). Parallel program schemata. *Journal of Computer and System Sciences*, 3(2) :147 – 195.

[Schmitz and Schnoebelen, 2014] Schmitz, S. and Schnoebelen, P. (2014). The power of well-structured systems. *CoRR*, abs/1402.2908.